Lecture Notes in Computer Science 2017

Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Tokyo

Samuel P. Midkiff José E. Moreira Manish Gupta
Siddhartha Chatterjee Jeanne Ferrante Jan Prins
William Pugh Chau-Wen Tseng (Eds.)

Languages and Compilers for Parallel Computing

13th International Workshop, LCPC 2000
Yorktown Heights, NY, USA, August 10-12, 2000
Revised Papers

Springer

Volume Editors

Samuel P. Midkiff
José E. Moreira
Manish Gupta
Siddhartha Chatterjee
IBM T.J. Watson Research Center
P.O. Box 218, Yorktown Heights, NY 10598, USA
E-mail: {smidkiff,jmoreira,mgupta,sc}@us.ibm.com

Jeanne Ferrante
University of California at San Diego, Computer Science and Engineering
9500 Gilman Drive, La Jolla, CA 92093-0114, USA
E-mail: ferrante@cs.ucsd.edu

Jan Prins
University of North Carolina, Department of Computer Science
Chapel Hill, NC 27599-3175, USA
E-mail: prins@unc.edu

William Pugh
Chau-Wen Tseng
University of Maryland, Department of Computer Science
College Park, MD 20742, USA
E-mail: {pugh,tseng}@cs.umd.edu

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Languages and compilers for parallel computing : 13th international workshop ; revised papers / LCPC 2000, Yorktown Heights, NY, USA, August 10 - 12, 2000. Samuel P. Midkiff ... (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Tokyo : Springer, 2002
(Lecture notes in computer science ; Vol. 2017)
ISBN 3-540-42862-3

CR Subject Classification (1998): D.3, D.1.3, F.1.2, B.2.1, C.2

ISSN 0302-9743
ISBN 3-540-42862-3 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

Typesetting: Camera-ready by author, data conversion by Boller Mediendesign
Printed on acid-free paper SPIN: 10782214 06/3142 5 4 3 2 1 0

Foreword

This volume contains the papers presented at the 13th International Workshop on Languages and Compilers for Parallel Computing. It also contains extended abstracts of submissions that were accepted as posters. The workshop was held at the IBM T. J. Watson Research Center in Yorktown Heights, New York. As in previous years, the workshop focused on issues in optimizing compilers, languages, and software environments for high performance computing. This continues a trend in which languages, compilers, and software environments for high performance computing, and not strictly parallel computing, has been the organizing topic. As in past years, participants came from Asia, North America, and Europe.

This workshop reflected the work of many people. In particular, the members of the steering committee, David Padua, Alex Nicolau, Utpal Banerjee, and David Gelernter, have been instrumental in maintaining the focus and quality of the workshop since it was first held in 1988 in Urbana-Champaign. The assistance of the other members of the program committee – Larry Carter, Sid Chatterjee, Jeanne Ferrante, Jans Prins, Bill Pugh, and Chau-wen Tseng – was crucial. The infrastructure at the IBM T. J. Watson Research Center provided trouble-free logistical support. The IBM T. J. Watson Research Center also provided financial support by underwriting much of the expense of the workshop. Appreciation must also be extended to Marc Snir and Pratap Pattnaik of the IBM T. J. Watson Research Center for their support.

Finally, we would like to thank the referees who spent countless hours assisting the program committee members in evaluating the quality of the submissions: Scott B. Baden, Jean-Francois Collard, Val Donaldson, Rudolf Eigenmann, Stephen Fink, Kang Su Gatlin, Michael Hind, Francois Irigoin, Pramod G. Joisha, Gabriele Keller, Wolf Pfannenstiel, Lawrence Rauchweger, Martin Simons, D. B. Skillicorn, Hong Tang, and Hao Yu.

January 2001

Manish Gupta
Sam Midkiff
José Moreira

Organization

The 13th annual International Workshop on Languages and Compilers for High Performance Computing (LCPC 2000) was organized and sponsored by the IBM T. J Watson Research Center, Yorktown Heights, New York

Steering Committee

Utpal Banerjee	*Intel Corporation*
David Gelernter	*Yale University*
Alex Nicolau	*University of California at Irvine*
David A. Padua	*University of Illinois at Urbana-Champaign*

Program Committee

Siddhartha Chatterjee	*University of North Carolina at Chapel Hill*
Larry Carter	*University of California at San Diego*
Jeanne Ferrante	*University of California at San Diego*
Manish Gupta	*IBM T. J. Watson Research Center*
Sam Midkiff	*IBM T. J. Watson Research Center*
José Moreira	*IBM T. J. Watson Research Center*
Jans Prins	*University of North Carolina at Chapel Hill*
Bill Pugh	*University of Maryland*
Chau-Wen Tseng	*University of Maryland*

Sponsoring Institutions

The IBM T. J. Watson Research Center, Yorktown Heights, New York

Table of Contents

Presented Papers

Posters

Accurate Shape Analysis for Recursive Data Structures*

Francisco Corbera, Rafael Asenjo, and Emilio Zapata

Dept. Computer Architecture, University of Málaga, Spain
{corbera,asenjo,ezapata}@ac.uma.es

Abstract. Automatic parallelization of codes which use dynamic data structures is still a challenge. One of the first steps in such parallelization is the automatic detection of the dynamic data structure used in the code. In this paper we describe the framework and the compiler we have implemented to capture complex data structures generated, traversed, and modified in C codes. Our method assigns a *Reduced Set of Reference Shape Graphs* (RSRSG) to each sentence to approximate the shape of the data structure after the execution of such a sentence. With the properties and operations that define the behavior of our RSRSG, the method can accurately detect complex recursive data structures such as a doubly linked list of pointers to trees where the leaves point to additional lists. Other experiments are carried out with real codes to validate the capabilities of our compiler.

1 Introduction

For complex and time-consuming applications, parallel programming is a must. Automatic parallelizing compilers are designed with the aim of dramatically reducing the time needed to develop a parallel program by generating a parallel version from a sequential code without special annotations. There are several well-known research groups involved in the development and improvement of parallel compilers, such as Polaris, PFA, Parafrase, SUIF, etc. We have noted that the detection step of current parallelizing compilers does a pretty good job when dealing with regular or numeric codes. However, they cannot manage irregular codes or symbolic ones, which are mainly based on complex data structures which use pointers in many cases. Actually, data dependence analysis is quite well known for array-based codes even when complex array access functions are present [5]. On the other hand, much less work has been done to successfully determine the data dependencies of code sections using dynamic data structures based on pointers. Nevertheless, this is a problem that cannot be avoided due to the increasing use of dynamic structures and memory pointer references.

* This work was supported by the Ministry of Education and Science (CICYT) of Spain (TIC96-1125-C03), by the European Union (BRITE-EURAM III BE95-1564), by APART: Automatic Performance Analysis: Resources and Tools, EU Esprit IV Working Group No. 29488

S.P. Midkiff et al. (Eds.): LCPC 2000, LNCS 2017, pp. 1–15, 2001.

With this motivation, our goal is to propose and implement new techniques that can be included in compilers to allow the automatic parallelization of real codes based on dynamic data structures. From this goal we have selected the shape analysis subproblem, which aims at estimating at compile time the shape the data will take at run time. Given this information, a subsequent analysis would detect whether or not certain sections of the code can be parallelized because they access independent data regions.

There are several ways this problem can be approached, but we focus in the graph-based methods in which the "storage chunks" are represented by nodes, and edges are used to represent references between them [2], [8], [9]. In a previous work [3], we combined and extended several ideas from these previous graph-based methods, for example, allowing more than a summary node per graph among other extensions. However, we keep the restriction of one graph per sentence in the code. This way, since each sentence of the code can be reached after following several paths in the control flow, the associated graph should approximate all the possible memory configurations arising after the execution of this sentence. This restriction leads to memory and time saving, but at the same time it significantly reduces the accuracy of the method. In this work, we have changed our previous direction by selecting a tradeoff solution: we consider several graphs with more than a summary node, while fulfilling some rules to avoid an explosion in the number of graphs and nodes in each graph.

Among the first relevant studies which allowed several graphs were those developed by Jones et al. [7] and Horwitz et al. [6]. These approaches are based on a "k-limited" approximation in which all nodes beyond a k selectors path are joined in a summary node. The main drawback to these methods is that the node analysis beyond the "k-limit" is very inexact and therefore they are unable to capture complex data structures. A more recent work that also allows several graphs and summary nodes is the one presented by Sagiv et al.[10]. They propose a parametric framework based on a 3-valued logic. To describe the memory configuration they use 3-valued structures defined by several predicates. These predicates determine the accuracy of the method. As far as we know the currently proposed predicates do not suffice to deal with the complex data structures that we handle in this paper.

With this in mind, our proposal is based on approximating all the possible memory configurations that can arise after the execution of a sentence by a set of graphs: the *Reduced Set of Reference Shape Graphs* (RSRSG). We see that each RSRSG is a collection of *Reference Shape Graphs* (RSG) each one containing several non-compatible nodes. Finally, each node represents one or several memory locations. Compatible nodes are "summarized" into a single one. Two nodes are compatible if they share the same reference properties. With this framework we can achieve accurate results without excessive compilation time. Besides this, we cover situations that were previously unsolved, such as detection of complex structures (lists of trees, lists of lists, etc.) and structure permutation, as we will see in this article.

The rest of the paper is organized as follows. Section 2 briefly describes the whole framework, introducing the key ideas of the method and presenting the data structure example that will help in understanding node properties and operations with graphs. These properties are described in Sect. 3 where we show how the RSG can accurately approximate a memory configuration. The analysis method have been implemented in a compiler which is experimentally validated, in Sect. 4, by analyzing several C codes based on complex data structures. Finally, we summarize the main contributions and future work in Sect. 5.

2 Method Overview

Basically, our method is based on approximating all possible memory configurations that can appear after the execution of a sentence in the code. Note that due to the control flow of the program, a sentence could be reached by following several paths in the control flow. Each "control path" has an associated memory configuration which is modified by each sentence in the path. Therefore, a single sentence in the code modifies all the memory configurations associated with all the control paths reaching this sentence. Each memory configuration is approximated by a graph we call *Reference Shape Graphs* (RSG). So, taking all this into account, we conclude that each sentence in the code will have a set of RSGs associated with it. This set of RSGs will describe the shape of the data structure after the execution of this sentence.

The calculation of this set of graphs is carried out by the **symbolic execution** of the program over the graphs. In this way, each program sentence transforms the graphs to reflect the changes in the memory configurations derived from the sentence execution. The RSGs are graphs in which nodes represent memory locations which have similar reference patterns. Therefore, a single node can safely and accurately represents several memory locations (if they are similarly referenced) without losing their essential characteristics.

To determine whether or not two memory locations should be represented by a single node, each one is annotated with a set of properties. Now, two different memory locations will be "summarized" in a single node if they fulfill the same properties. Note that the node inherits the properties of the memory locations represented by this node. Besides this, two nodes can be also summarized if they represent "summarizable" memory locations. This way, a possibly unlimited memory configuration can be represented by a limited size RSG, because the number of different nodes is limited by the number of properties of each node. These properties are related to the reference pattern used to access the memory locations represented by the node. Hence the name *Reference Shape Graph*.

As we have said, all possible memory configurations which may arise after the execution of a sentence are approximated by a set of RSGs. We call this set *Reduced Set of Reference Shape Graphs* (RSRSG), since not all the different RSGs arising in each sentence will be kept. On the contrary, several RSGs related to different memory configurations will be fused when they represent memory locations with similar reference patterns. There are also several properties related

to the RSGs, and two RSGs should share these properties to be joined. Therefore, besides the number of nodes in an RSG, the number of different RSGs associated with a sentence are limited too. This union of RSGs greatly reduces the number of RSGs and leads to a practicable analysis.

The symbolic execution of the code consists in the abstract interpretation of each sentence in the code. This abstract interpretation is carried out iteratively for each sentence until we reach a fixed point in which the resulting RSRSG associated with the sentence does not change any more [4]. This way, for each sentence that modifies dynamic structures, we have to define the abstract semantics which describes how these sentences modify the RSRSG. We consider six simple instructions that deal with pointers: $x = NULL$, $x = malloc$, $x = y$, $x \rightarrow sel = NULL$, $x \rightarrow sel = y$, and $x = y \rightarrow sel$. More complex pointer instructions can be built upon these simple ones and temporal variables.

The output RSRSG resulting from the abstract interpretation of a sentence over an input RSRSG_i is generated by applying the abstract interpretation to each $rsg_i \in \mathrm{RSRSG}_i$. After the abstract interpretation of the sentence over the $rsg_i \in \mathrm{RSRSG}_i$ we obtain a set of output rsg_o. As we said, we cannot keep all the rsg_o arising from the abstract interpretation. On the contrary, each rsg_o will be compressed, which means the summarization of compatible nodes in the rsg_o. Furthermore, some of the rsg_os can be fused in a single RSG if they represent similar memory configurations. This operation greatly reduces the number of RSGs in the resulting RSRSG. In the worst case, the sequence of operations that the compiler carries out in order to symbolically execute a sentence are: graph division, graph prune, sentence symbolic execution (RSG modification), RSG compression and RSG union to build the final RSRSG. Due to space constraints we cannot formally describe this operations neither the abstract semantics carried out by the compiler. However, in order to provide an overview of our method we present a data structure example which will be refered to during the framework and operations description.

2.1 Working Example

The data structure, presented in Fig. 1 (a), is a doubly linked list of pointers to trees. Besides this, the leaves of the trees have pointers to doubly linked lists. The pointer variable S points to the first element of the doubly linked list (header list). Each item in this list has three pointers: nxt, prv, and $tree$. This $tree$ selector points to the root of a binary tree in which each element has the lft and rgh selectors. Finally, the leaves of the trees point to additional doubly linked lists. All the trees pointed to by the header list are independent and do not share any element. In the same way, the lists pointed to by the leaves of the same tree or different trees are also independent.

This data structure is built by a C code which traverses the elements of the header list with two pointers and eventually can permute two trees. Our compiler has analyzed this code obtaining an RSRSG for each sentence in the program. Figure 1 (b) shows a compact represetation of the RSRSG obtained for the last sentence of the code after the compiler analysis.

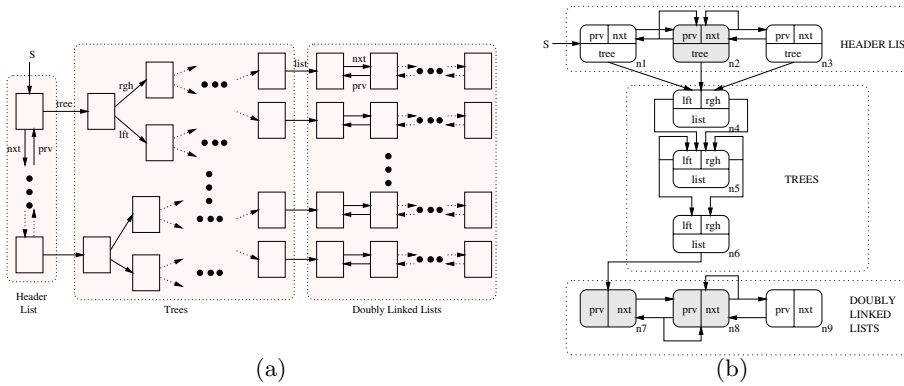


Fig. 1. A complex data structure and compact representation of the resulting RSRSG.

As we will see in the next sections, from the RSRSG represented in Fig. 1 (b) we can infer the actual properties of the real data structure: the trees and lists do not share elements and therefore they can be traversed in parallel. These results and other examples of real codes (sparse matrix-vector multiplication, sparse LU factorization and Barnes-Hut N-body simulation) with different data structures are presented in Sect. 4. But first, we describe our framework with a formal description of the RSGs in the next section.

3 Reference Shape Graph

First, we need to present the notation used to describe the different memory configurations that may arise when executing a program.

Definition 1 *We call a collection of dynamic structures a memory configuration. These structures comprise several memory chunks, that we call memory locations, which are linked by references. Inside these memory locations there is room for data and for pointers to other memory locations. These pointers are called selectors.*

We represent the memory configurations with the tuple $M = (L, P, S, PS, LS)$ *where:* **L** *is the set of memory locations;* **P** *is the set of pointer variables (pvars) used in the program;* **S** *is the set of selectors declared in the data structures;* **PS** *is the set of references from pvars to memory locations, of the type* $< pvar, l >$*, with* $pvar \in P$ *and* $l \in L$*; and* **LS** *is the set of links between memory locations, of the form* $< l_1, sel, l_2 >$ *where* $l_1 \in L$ *references* $l_2 \in L$ *by selector* $sel \in S$*.*

We will use L(m), P(m), S(m), PS(m), and LS(m) to refer to each one of these sets for a particular memory configuration, m. □

Therefore, we can assume that the RSRSG of a program sentence is an approximation of the memory configuration, M, after the execution of this sen-

tence. But let us first describe the *RSGs* now that we know how a memory configuration is defined.

Definition 2 *An RSG is a graph represented by the tuple* $RSG = (N, P, S, PL, NL)$ *where:* **N***: is the set of nodes. Each node can represent several memory locations if they fulfill certain common properties;* **P***: is the set of pointer variables (pvars) used in the program;* **S***: is the set of declared selectors;* **PL***: is the set of references from pvars to nodes, of the type* $< pvar, n >$ *with* $pvar \in P$ *and* $n \in N$*; and* **NL***: is the set of links between nodes, of the type* $< n_1, sel, n_2 >$ *where* $n_1 \in N$ *references* $n_2 \in N$ *by selector* $sel \in S$*.*

We will use N(rsg), P(rsg), S(rsg), PL(rsg), and NL(rsg) to refer to each one of these sets for a particular RSG, rsg. □

To obtain the RSG which approximates the memory configuration, M, an abstraction function is used, $F : M \rightarrow RSG$. This function maps memory locations into nodes and references to memory locations into references to nodes at the same time. In other words, F, translates the memory domain into the graph domain. This, function F comprises three functions: $F_n : L \rightarrow N$ takes care of mapping memory locations into nodes; $F_p : PS \rightarrow PL$ maps references from pvars to memory locations into references from pvars to nodes, and $F_l : LS \rightarrow NL$ maps links between locations into links between nodes.

It is easy to see that: $F_p(< pvar, l >) =< pvar, n >$ iif $F_n(l) = n$ and $F_l(< l_1, sel, l_2 >) =< n_1, sel, n_2 >$ iif $F_n(l_1) = n_1 \wedge F_n(l_2) = n_2$ which means that translating references to locations into references to nodes is trivial after mapping locations into nodes. This translates almost all the complexity involved in function F to function F_n which actually maps locations into nodes.

Now we focus on F_n which extracts some important properties from a memory location and, depending on these, this location is translated into a node. Besides this, if several memory locations share the same properties then this function maps all of them into the same node of the *RSG*. Due to this dependence on the location properties, the F_n function will be described during the presentation of the different *properties* which characterize each node. These properties are: Type, Structure, Simple Paths, Reference pattern, Share information, and Cycle links. These are now described.

3.1 Type

This property tries to extract information from the code text. The idea is that two pointers of different types should not point to the same memory location (for example, a pointer to a graph and another to a list). Also, the memory location pointed to by a graph pointer should be considered as a graph. This way, we assign the `TYPE` property, of a memory location l, from the type of the pointer variable used when that memory location l is created (by malloc or in the declarative section of the code).

Therefore, two memory locations, l_1 and l_2, can be mapped into the same node if they share the same `TYPE` value: If $F_n(l_1) = F_n(l_2) = n$ then $\texttt{TYPE}(l_1) =$

TYPE(l_2), where the TYPE() function returns the TYPE value. Note that we can use the same property name and function, TYPE(), for both memory locations and nodes. Clearly, TYPE(n) = TYPE(l) when $F_n(l) = n$.

This property leads to the situation where, for the data structure presented in Fig. 1 (a), the nodes representing list memory locations will not be summarized with those nodes representing tree locations, as we can see in Fig. 1 (b).

3.2 Structure

As we have just seen, the TYPE property keeps two different nodes for two memory locations of different types. However, we also want to avoid the summarization of two nodes which represent memory locations of the same type but which do not share any element. That is, they are non-connected components. This behavior is achieved by the use of the STRUCTURE property, which takes the same value for all memory locations (and nodes) belonging to the same connected component. Again, two memory locations can be represented by the same node if they have the same STRUCTURE value: If $F_n(l_a) = F_n(l_b) = n$ then STRUCTURE(l_a) = STRUCTURE(l_b)

This leads to the condition that two locations must fulfill in order to share the same STRUCTURE value: STRUCTURE(l_a) = STRUCTURE(l_b) = val iif $\exists l_1, ..., l_i | (< l_a, sel_1, l_1 >, < l_1, sel_2, l_2 >, ..., < l_i, sel_{i+1}, l_b > \in LS) \vee (< l_b, sel_1, l_1 >, < l_1, sel_2, l_2 >, ..., < l_i, sel_{i+1}, l_a > \in LS)$, which means that two memory locations, l_a and l_b, have the same STRUCTURE value if there is a path from l_a to l_b (first part of the previous equation) or from l_b to l_a (second part of the equation). In the same way we can define STRUCTURE(n).

3.3 Simple Paths

The SPATH property further restricts the summarizing of nodes. *Simple paths* denominates the access path from a pointer variable (pvar) to a location or node. An example of a simple path is $p \rightarrow s$ in which the pvar p points to the location s. In this example the simple path for s is $< p >$. The use of the simple path avoids the summarization of nodes which are directly pointed to by the pvars (actually, these nodes are the entry points to the data structure). We define the SPATH property for a memory location $l \in L(m)$ as SPATH(l) = $\{p_1, .., p_n\}$ where $< p_i, l > \in PS(m)$. This property is similarly defined for the RSG domain. Now, two memory locations are represented by the same node if they have the same SPATH (If $F_n(l_1) = F_n(l_2)$ then SPATH(l_1) = SPATH(l_2)).

3.4 Reference Patterns

This new property is introduced to classify and represent by different nodes the memory locations with different reference patterns. We understand by reference pattern the type of selectors which point to a certain memory location and which point from this memory location to others.

This is particularly useful for keeping singular memory locations of the data structure in separate nodes. For example, the head/tail and the rest of the elements of a single linked list are two kinds of memory locations. These will be represented by different nodes, because the head location is not referenced by other list entries and the tail location does not reference any other list location. The same would also happen for more complex data structures built upon more simple structures (such as lists of lists, trees of lists, etc.). For example, in Fig. 1 (a), the root of one of the trees is referenced by the header list and the leaves do not point to tree items but to a doubly linked list. Thanks to the reference patterns, the method results in the RSRSG of Fig. 1 (b), where the root of the tree, the leaves, and the other tree items are clearly identified.

In order to obtain this behavior, we define two sets `SELINset` and `SELOUTset` which contain the set of input and output selectors for a location: $\texttt{SELINset}(l_1) = \{sel_i \in S | \exists l_2 \in L, < l_2, sel_i, l_1 > \in LS\}$ and $\texttt{SELOUTset}(l_1) = \{sel_i \in S | \exists l_2 \in L, < l_1, sel_i, l_2 > \in LS\}$, where we see that sel_i is in the $\texttt{SELINset}(l_1)$ if l_1 is referenced from somewhere by selector sel_i, or sel_i is in $\texttt{SELOUTset}(l_1)$ if $l_1.sel_i$ points to somewhere outside.

3.5 Share Information

This is a key property for informing the compiler about the potential parallelism exhibited by the analyzed data structure. Actually, the share information can tell whether at least one of the locations represented by a node is referenced more than once from other memory locations. That is, a shared node represents memory locations which can be accessed from several places and this may prevent the parallelization of the code section which traverses these memory locations. From another point of view, this property helps us to determine if a cycle in the RSG graph is representing cycles in the data structure approximated by this RSG or not.

Due to the relevance of this property, we use two kinds of attributes for each node: (i) $\texttt{SHARED}(n)$ with $n \in N(rsg)$, is a Boolean function which returns "true" if any of the locations, l_1, represented by n are referenced by other locations, l_2 and l_3, by different selectors, sel_i and sel_j. Therefore, this `SHARED` function tells us whether there may be a cycle in the data structure represented by the RSG or not. If $\texttt{SHARED}(n)$ is 0, we know that even if we reach the node n by sel_1 and later by sel_2, we are actually reaching two different memory locations represented by the same n node, and therefore there is no cycle in the approximated data structure. (ii) $\texttt{SHSEL}(n, sel)$ with $n \in N(rsg)$ and $sel \in S$, is a Boolean function which returns "true" if any of the memory locations, l_1, represented by n can be referenced more than once by selector sel from other locations, l_2 and l_3. This way, with the `SHSEL` function, we can distinguish two different situations that can be represented by an RSG with a node, n, and a selector, sel, pointing to itself. If $\texttt{SHSEL}(n, sel) = 0$ we know that this node is representing an acyclic unbounded data structure (the size is not known at compile time). For example, in a list, all the elements of the list (locations) are represented by the same node, n, but following selector sel we always reach a different memory location. On the

other hand, if $\texttt{SHSEL}(n, sel) = 1$, for the same list example, by following selector sel we can reach an already visited location, which means that there are cycles in the data structure.

Let's illustrate these SHARED and SHSEL properties using the compact representation of the RSRSG presented in Fig. 1 (b). In this Fig., shaded nodes have the SHARED property set to true. For example, in the header list the middle node n_2 is shared, $\texttt{SHARED}(n_2)$=1, because n_2 is referenced by selectors nxt and prv. However, the $\texttt{SHSEL}(n_2, nxt)$=$\texttt{SHSEL}(n_2, prv)$ =0 which means that by following selector nxt or prv it is not possible to reach an already visited memory location. Actually, in this example, there are no selectors with the SHSEL property set to true. So, the same happens for node n_8 which represents the middle items of the doubly linked lists.

We can also see in Fig. 1 (b), that node n_4 is not shared, which states that, in fact, from memory locations represented by n_1, n_2, and n_3 we can reach different trees which do not share any elements (as we see in Fig. 1 (a)). Finally, node n_7 is shared because it is pointed to by selectors $list$ and prv. However, due to $\texttt{SHSEL}(n_7, list)$=0 we can ensure that two different leaves of the trees will never point to the same doubly linked list.

3.6 Cycle Links

The goal of this property is to increase the accuracy of the data structure representation by avoiding unnecessary edges that can appear during the RSG updating process.

The cycle links of a node, n, are defined as the set of pairs of references $< sel_i, sel_j >$ such that when starting at node n and consecutively following selectors sel_i and sel_j, the n node is reached again. More precisely, for $n \in N(rsg)$ we define: $\texttt{CYCLELINKS}(n) = \{< sel_i, sel_j > | sel_i, sel_j \in S\}$, such that if $< sel_i, sel_j > \in \texttt{CYCLELINKS}(n)$ then: $\forall l_i, F_n(l_i) = n$, if $< l_i, sel_i, l_j > \in LS$ then $\exists < l_j, sel_j, l_i > \in LS$.

This CYCLELINKS set maintains similar information to that of "identity paths" in the Abstract Storage Graph (ASG) [8], which is very useful for dealing with doubly-linked structures. For example, in the data structure presented in Fig. 1 (a), the elements in the middle of the doubly linked lists have two cycle links: $< nxt, prv >$ and $< prv, nxt >$, due to starting at a list item and consecutively following selectors nxt and prv (or prv and nxt) the starting item is reached. Note, that this does not apply to the first or last element of the doubly linked list. This property is captured in the RSRSG shown in Fig. 1 (b) where we see three nodes for the double linked lists (one for the first element of the list, another for the last element, and another between them to represent the middle items in the list). This middle node, n_8, is annotated by our compiler with $\texttt{CYCLELINKS}(n_8) = \{< nxt, prv >, < prv, nxt >\}$.

We conclude here that the CYCLELINKS property is used during the pruning process which take place after the node materialization and RSG modification. So, in contrast with the other five properties described in previous subsections,

the CYCLELINKS property does not prevent the summarization of two nodes with do not share the CYCLELINKS sets and therefore do not affect the F_n function.

3.7 Compression of Graphs

After the symbolic execution of a sentence over an input RSRSG, the resulting RSRSG may contain RSGs with redundant information, which can be removed due to node summarization or compression of the RSG.

In order to do this, after the symbolic execution of a sentence, the method applies the COMPRESS function over the just modified RSGs. This COMPRESS function first call to the boolean C_NODES_RSG one, which identifies the compatible nodes that will later be summarized. This Boolean function just has to check whether or not the first five properties previously described are the same for both nodes (as we said in the previous subsection the CYCLELINKS property does not affect the compatibility of two nodes).

There is a similar function which returns true when two memory locations are compatible. With this, we can finally define F_n as the function which maps all the compatible memory locations into the same node, which happens when they have the same TYPE, STRUCTURE, SPATH and SHARED properties, and compatible reference patterns.

4 Experimental Results

All the previously mentioned operations and properties have been implemented in a compiler written in C which analyzes a C code to generate the RSRSG associated with each sentence of the code. As we said, the symbolic execution of each sentence over an RSRSG is going to generate a modified RSRSG. Before the symbolic execution of the code, the compiler can also extract some important information from the program in a previous pass. For example, a quite frequent pattern arising in C codes based on dynamic data structures is the following: `while (x != NULL) { ... }`.

In this case the compiler can assert that at the entry of the while body the pvar $x \neq NULL$. Besides this, if we have not exited the while body with a `break` sentence, we can also ensure that just after the while body the pvar $x = NULL$. This information is used to simplify the analysis and increase the accuracy of the method. More precisely, we can reduce the number of RSGs and/or reduce the complexity of this RSG by diminishing the number of memory configurations represented by each RSG. Other sentences from which we can extract useful information are IF-THEN-ELSE, FOR loops, or any conditional sentence.

The implementation of this idea has been carried out by the definition of certain *pseudoinstructions* that we call FORCE. These pseudoinstructions are inserted in the code by the first pass of the compiler and will be symbolically executed as regular sentences. Therefore, each one of these FORCE sentences has its own abstract semantics and its own associated RSRSG. The FORCE pseudoinstructions we have considered are: $\texttt{FORCE}_{[x==NULL]}(rsg)$, $\texttt{FORCE}_{[x!=NULL]}(rsg)$, $\texttt{FORCE}_{[x==y]}(rsg)$, $\texttt{FORCE}_{[x!=y]}(rsg)$, $\texttt{FORCE}_{[x\rightarrow sel==NULL]}(rsg)$.

In addition, we have found that the $\texttt{FORCE}_{[x!=NULL]}$ pseudoperation can be also placed just before the following sentences: $x \rightarrow sel = NULL$, $x \rightarrow sel = y$, $y = x \rightarrow sel$ and for any sentence with an occurrence of the type $x \rightarrow val$, where val is a non-pointer field, under the assumption that the code is correct. That is, it makes sense to assume that before the execution of all these three sentences, x is not NULL (in other cases the code would produce an error at execution time).

With the compiler described we have analyzed the code which generates, traverses, and modifies several codes: the working example presented in section 2.1, the Sparse Matrix-vector multiplication, the Sparse LU factorization and the Barnes-Hut code. All these codes were analyzed by our compiler in a Pentium III 500MHz with 128MBytes main memory. The time and memory required by the compiler are summarized in table 1. The particular aspects of these codes are described next.

Table 1. Time and space required by the compiler to analyze several codes

	Working Example	S. Matrix-vector	S. LU factorization	Barnes-Hut
Time	0'13"	0'04"	1'38"	4'04"
Space	2.7 MB	1.6 MB	24 MB	47 MB

4.1 Working Example's RSRSG

We refer in this subsection to the code that generates, traverses, and modifies the data structure presented in Fig. 1 (a). A compact representation of the resulting RSRSG for the last sentence of the code can be seen in Fig. 1 (b). Although we do not show the code due to space constraints, we have to say that this code presents an additional difficulty due to some tree permutations being carried out during data structure modification. The problem arising during structure permutation is that it is very easy to temporally assign the SHARED=true property to the root of one of the trees that we are permutating, when this root is temporally pointed to by two different locations from the header list. If this shared property remains true after the permutation we would have a shaded n_4 node in Fig. 1 (b). This would imply that two different items from the header list can point to the same tree (which would prevent the parallel execution of traversing the trees). However, this problem is solved because, after the permutation, the method reassigns false to the shared property thanks to the combination of our properties and the division of graph operations. Summarizing, after the compiler analyzes this code, the compact representation of the resulting RSRSG for the last sentence of the program (Fig. 1 (b)) accurately describes the data structure depicted in Fig. 1 (a) in the sense that: (i) The compiler successfully detects the doubly linked list which is acyclic by selectors nxt or prv and whose elements point to binary trees; (ii) As $\texttt{SHSEL}(n_4, tree)$=0, we can say that two different items of the header list cannot point to the same tree; (iii) At the same time, as no tree node (n_4, n_5 and n_6) is shared, we can say that different trees do not share items; (iv) The same happens for the doubly linked list pointed to by the

tree leaves: all the lists are independent, there are no two leaves pointing to the same list, and these lists are acyclic by selectors *nxt* or *prv*.

Besides this, our compiler has also analyzed three C program kernels which generate, traverse, and modify complex dynamic data structures which we describe next.

4.2 Sparse Matrix-Vector Multiplication

Here we deal with an irregular code which implements a sparse matrix by vector multiplication, $r = M \times v$. The sparse matrix, M, is stored in memory as a header doubly linked list with pointers to other doubly linked lists which store the matrix rows. The sparse vectors, v and r are also doubly linked lists. This can be seen in Fig. 2(a). Note that vector r grows during the multiplication process.

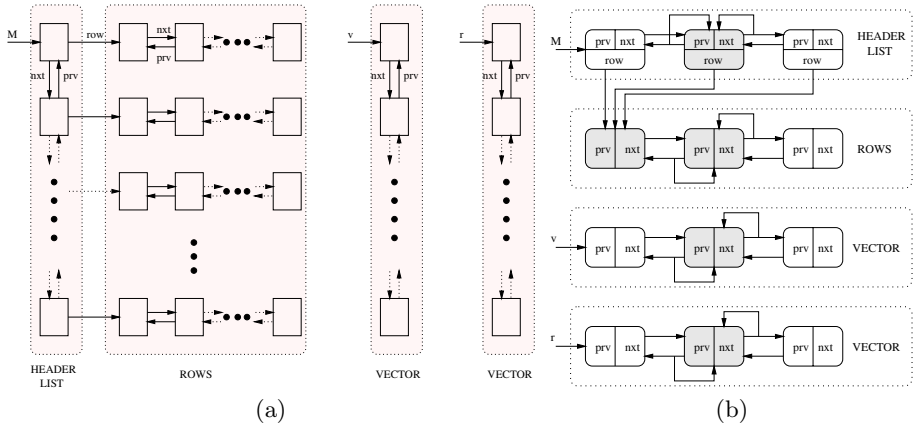


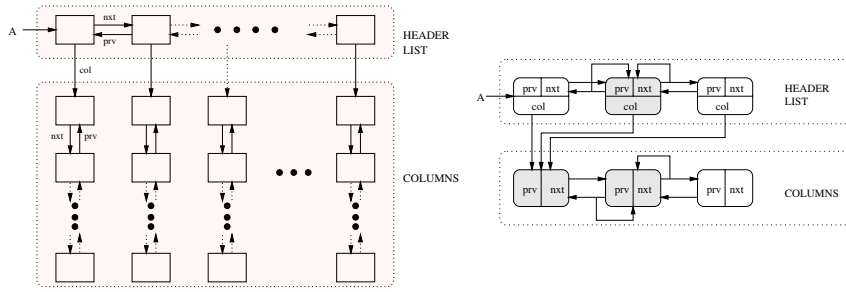
Fig. 2. Sparse matrix-vector multiplication data structure and compacted RSRSG.

After the analysis process, carried out by our compiler, the resulting RSRSG accurately represents this data structure. Actually, in Fig. 2(b) we present a compact representation of the resulting RSRSG for the last sentence of the code. First, note that the three structures involved in this code are kept in separate subgraphs. Even when the data type for vectors v and r and rows of M, is the same, the `STRUCTURE` property avoids the union of these graphs into a single one. This RSRSG states that the rows of the matrix are pointed to from different elements of the header list (there is no selector with the shared property set to true). Also, the doubly linked lists which store the rows of M and the vectors v and r are acyclic by selectors *nxt* and *prv*.

The same RSRSG is also reached just before the execution of the outermost loop which takes care of the matrix multiplication, but without the r subgraph which is generated during this multiplication.

4.3 Sparse LU Factorization

The kernel of many computer-assisted scientific applications is to solve large sparse linear systems. The code we analyze now solves non-symmetric sparse linear systems by applying the LU factorization of the sparse matrix, computed by using a general method. In particular, we have implemented an in-place code for the direct right-looking LU algorithm, where an n-by-n matrix A is factorized. The code includes a column pivoting operation (partial pivoting) to provide numerical stability and preserve sparseness. The input matrix A columns as well as the resulting in place LU columns are stored in one-dimensional doubly linked lists (see Fig. 3 (a)), to facilitate the insertion of new entries and to allow column permutations.



(a) (b)

Fig. 3. Sparse LU factorization data structure and compacted RSRSG.

After the LU code analysis we obtain the same RSRSG for the sentences just after the matrix initialization and after the LU factorization. A compact representation of this RSRSG is shown in Fig. 3(b). As we can see, variable A points to a doubly linked list, the header list and each node of this list points to a single doubly linked list which represents a matrix column. The properties of the data structure represented by this RSRSG can be inferred following the same arguments we presented in the previous subsection.

4.4 Barnes-Hut N-body Simulation

This code is based on the algorithm presented in [1] which is used in astrophysics. The data structure used in this code is based on a hierarchical octree representation of space in three dimensions. In two dimensions, a quadtree representation is used. However, due to memory constraints (the octree and the quadtree versions run out of memory in our 128MB Pentium III) we have simplified the code to use a binary tree. In Fig. 4(a) we present a schematic view of the data structure used in this code. The bodies are stored by a single linked list pointed to by the pvar *Lbodies*.

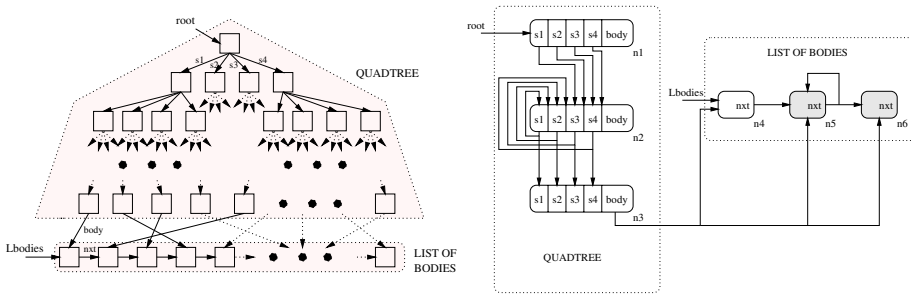


Fig. 4. Barnes-Hut data structure and compacted RSRSG.

After the code analysis, the compact representation of the RSRSG at the end of each step of the algorithm is presented in Fig. 4(b). We can see that the root of the tree is represented by node n_1, the middle elements of the tree by node n_2 and the leaves by n_3. Note that these leaves can point to any body stored in the *Lbodies* list represented by nodes n_4, n_5, and n_6. As tree nodes are not shared and selectors also have the `SHSEL` property set to false, a subsequent analysis of the code can state that the tree can be traversed and updated in parallel. This analysis can also conclude that there are no two different leaves pointing to the same body (entry in the *Lbodies* list) due to nodes n_4, n_5, and n_6 not being shared by selector *body*.

5 Conclusions and Future Work

We have developed a compiler which can analyze a C code to determine the RSRSG associated with each sentence of the code. Each RSRSG contains several RSGs, each one representing the different data structures which may arise after following different paths in the control flow graph of the code. However, several RSGs can be joined if they represent similar data structures, in this way reducing the number of RSGs associated with a sentence. Every RSG contains nodes which represent one or several memory locations. To avoid an explosion in the number of nodes, all the memory locations which are similarly referenced are represented by the same node. This reference similarity is captured by the properties we assign to the memory locations. In comparison with previous works, we have increased the number of properties assigned to each node. This leads to more nodes in the RSG because the nodes now have to fulfill more properties to be summarized. However, by avoiding the summarization of these nodes, we keep a more accurate representation of the data structure. This is a key issue when analyzing the parallelism exhibited by a code.

Our compiler symbolically executes each sentence in the code, transforming the RSGs to reflect the modifications in the data structure that are carried out due to the execution of the sentence. We have validated the compiler with several C codes which generate, traverse, and modify complex dynamic data structures,

such as a doubly linked list of pointers to trees where the leaves point to other doubly linked lists. Other structures have been also accurately identified by the compiler, even in the presence of structure permutations (for example, column permutations in the sparse LU code). As far as we know, there is no compiler achieving such successful results for these kinds of data structures appearing in real codes.

In the near future we will develop an additional compiler pass that will automatically analyze the RSRSGs and the code to determine the parallel loops of the program and allow the automatic generation of parallel code.

References

1. J. Barnes and P. Hut. *A Hierarchical O(n· log n) force calculation algorithm.* Nature v.324, December 1986.
2. D. Chase, M. Wegman and F. Zadeck. *Analysis of Pointers and Structures.* In SIGPLAN Conference on Programming Language Design and Implementation, 296-310. ACM Press, New York, 1990.
3. F. Corbera, R. Asenjo and E.L. Zapata *New shape analysis for automatic parallelization of C codes.* In ACM International Conference on Supercomputing, 220–227, Rhodes, Greece, June 1999.
4. P. Cousot and R. Cousot. *Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points.* In Proceedings of the ACM Symposium on Principles of Programming Languages. ACM Press, New York. 238–252, 1977.
5. J. Hoeflinger and Y. Paek *The Access Region Test.* In Twelfth International Workshop on Languages and Compilers for Parallel Computing (LCPC'99), The University of California, San Diego, La Jolla, CA USA, August, 1999.
6. S. Horwitz, P. Pfeiffer, and T. Reps. *Dependence Analysis for Pointer Variables.* In Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, 28-40, June 1989.
7. N. Jones and S. Muchnick. *Flow Analysis and Optimization of Lisp-like Structures.* In Program Flow Analysis: Theory and Applications, S. Muchnick and N. Jones, Englewood Cliffs, NJ: Prentice Hall, Chapter 4, 102-131, 1981.
8. J. Plevyak, A. Chien and V. Karamcheti. *Analysis of Dynamic Structures for Efficient Parallel Execution.* In Languages and Compilers for Parallel Computing, U. Banerjee, D. Gelernter, A. Nicolau and D. Padua, Eds. Lectures Notes in Computer Science, vol 768, 37-57. Berlin Heidelberg New York: Springer-Verlag 1993.
9. M. Sagiv, T. Reps and R. Wilhelm. *Solving Shape-Analysis problems in Languages with destructive updating.* ACM Transactions on Programming Languages and Systems, 20(1):1-50, January 1998.
10. M. Sagiv, T. Reps, and R. Wilhelm, *Parametric shape analysis via 3-valued logic.* In Conference Record of the Twenty-Sixth ACM Symposium on Principles of Programming Languages, San Antonio, TX, Jan. 20-22, ACM, New York, NY, 1999, pp. 105-118.

Cost Hierarchies for Abstract Parallel Machines

John O'Donnell[1], Thomas Rauber[2], and Gudula Rünger[3]

[1] Computing Science Department, University of Glasgow,
Glasgow G12 8QQ, Great Britain
jtod@dcs.gla.ac.uk
[2] Institut für Informatik, Universität Halle-Wittenberg, 06099 Halle(Saale), Germany
rauber@informatik.uni--halle.de
[3] Fakultät für Informatik, Technische Universität Chemnitz,
09107 Chemnitz, Germany
ruenger@informatik.tu-chemnitz.de

Abstract. The Abstract Parallel Machine (APM) model separates the definitions of parallel operations from the application algorithm, which defines the sequence of parallel operations to be executed. An APM contains a set of parallel operation definitions, which specify how the computation is organized into independent sites of computation and what data exchanges are required. This paper adds explicit cost models as the third component of an APM system. The costs of parallel operations can be obtained either by analyzing a parallel operation definition, or by measuring performance on a real machine. Costs with monotonicity constraints allow the cost of an algorithm to be transformed automatically as the algorithm itself is transformed.

1 Introduction

There is increasing recognition of the fundamental role that cost models play in the design of parallel programs [19]. They enable the time and space complexity of a program to be determined, as for sequential algorithms, but parallel cost models serve several additional purposes. For example, intuition is often an inadequate basis for making the right choices about organizing the parallelism and using the system's resources. Suitable high level cost models allow the programmer to assess each alternative quantitatively during the design process, improving efficiency without requiring an inordinate amount of programming time. Portability of the efficiency is one of the chief problems in parallel programming, and cost models can help here by indicating where an algorithm should be modified to make effective use of a particular machine's capabilities. Such motivations have led to a plethora of approaches to cost modeling.

APMs (abstract parallel machines [14, 15]) are an approach for describing *parallel programming models*, especially in the context of program transformation. In this approach the parallel behavior is encapsulated in a set of ParOps (parallel operations), which are analogous to combinators in data parallel programming [13] and skeletons in BMF [5, 4]. An explicit separation is made between the definitions of the ParOps and the specific parallel algorithm to be

S.P. Midkiff et al. (Eds.): LCPC 2000, LNCS 2017, pp. 16–33, 2001.

implemented. An APM consists of a set of ParOps and a coordination language; algorithms are built up from the ParOps of one APM and are expressed using a coordination language for that APM. APMs are not meant as programming languages; rather, they illustrate programming models and their relationships, and provide a basis for algorithm transformation. The relationships between different parallel operations can be clarified with a hierarchy of related APMs.

There is some notion of costs already inherent in the definition of an APM, since the parallel operation definitions state how the operation is organized into parallel sites and what communications are required. The cost of an algorithm is determined by the costs of the ParOps it uses, and the cost of a ParOp could be derived from its internal description. This would allow a derivation to be based only on the information inside the APM definition. However, this is not the only way to obtain costs, and a more general and explicit treatment of costs can be useful.

In this paper, we enrich the APM approach by adding an explicit hierarchy of cost models. Every APM can be associated with one or more cost models, reflecting the possibility that the APM could be realized on different machines. The cost models are related to each other, following the connections in the APM hierarchy. Each cost model gives the cost of every ParOp within an APM. There are several reasonable ways to assign a cost to a parallel operation: it could be inferred from the internal structure (using the organization into sites, communications and data dependencies); it could be obtained by transforming mathematically the cost of the corresponding operation in a related APM; it could be determined by measuring the real cost for a specific implementation.

The goal is to support the transformation of an algorithm from one APM to another which gives *automatically* the new costs. Such a cost transformation could be used in several ways. The costs could guide the transformation of an algorithm through the APM hierarchy, from an abstract specification to a concrete realization. If the costs for an APM were obtained by measuring performance of a real machine, then realistic cost transformations are possible although the transformation takes place at an abstract level. In some algorithm derivations, it is helpful to begin with a horizontal transformation within the same APM that increases the costs. This can happen because the reorganized algorithm may satisfy the constraints required to allow a vertical transformation to a more efficient algorithm using a different APM. In such complex program derivations it is helpful to be explicit about the costs and programming models in use at each stage; that is the purpose of the APM methodology.

The rest of the paper is organized as follows: Section 2 gives an overview of the APM approach. Section 3 introduces cost hierarchies to APM hierarchies. Sections 4 and 5 illustrate the approach by examples. Section 6 concludes.

2 Overview of APMs

Abstract Parallel Machines (APMs) have been proposed in [14] as a formal framework for the derivation of parallel algorithms using a sequence of transfor-

mation steps. The formulation of a parallel algorithm depends not only on the algorithmic-specific potential parallelism but also on the parallel programming model and the target machine to be used. Every programming model provides a specific way to exploit or express parallelism, such as data parallel models or thread parallelism, in which the algorithm has to be described. An APM describes the behavior of a parallel programming model by providing operations (or patterns of operations) to be used in a program performed in that programming model. The basic operations provided by an APM are parallel operations (ParOps) which are combined by an APM-specific coordination language (usually, e.g., including a composition function). The application is formulated for an APM with ParOps as the smallest indivisible parallel units to express a specific application algorithm. Depending on the level of abstraction, an executable program (e.g., an MPI program for distributed memory machines) or a more abstract specification (e.g., a PRAM program for a theoretical analysis) results. The APM approach comprises:

- the specification framework of ParOps defining the smallest units in a specific parallel programming model, see Section 2.1;
- APM definitions consisting of a set of ParOps and a coordination language using them, see Section 4 for an example;
- a hierarchy of APMs built up from different APMs (expressing different parallel programming models) and relations of expressiveness between them;
- the formulation of an algorithm within one specific APM, see also Section 4 for an example; and
- the transformation of an algorithm into an equivalent algorithm (e.g., an algorithm with the same result semantics), but expressed in a different way within the same APM (horizontal transformation) or in a different APM (vertical transformation), see Section 2.3.

In the following subsections, we describe the APM framework in more detail.

2.1 Framework to Define a Parallel Operation ParOp

A parallel operation ParOp is executed on a number of sites $P_1, \ldots, P_n$ (these may be virtual processors or real processors). The framework for describing a ParOp uses a local function f_i executed on site P_i using the local state s_i of P_i for $i = 1, \ldots, n$ and data provided by other sites $z_1, \ldots, z_n$ or input data $x_1, \ldots, x_r$. Data from other sites used by P_i are provided by a projection function g_i which selects data from the set V of available values, consisting of the inputs $x_1, \ldots, x_r$ and the data $z_1, \ldots, z_n$ of all other sites, see Figure 1. The result of a ParOp is a new state $s'_1, \ldots, s'_n$ and output data $y_1, \ldots, y_r$. A ParOp is formally defined by

$$\begin{aligned} \textsf{ParOp}\ ARG(s_1, \ldots, s_n)\ (x_1, \ldots, x_r) \ &= \ ((s'_1, \ldots, s'_n),\ (y_1, \ldots, y_t)) \\ \text{where} \qquad (s'_i, z_i) \ &= \ f_i(s_i,\ g_i\ (V)) \\ (y_1, \ldots, y_t) \ &= \ g\ (V) \\ V \ &= \ ((x_1, \ldots, x_r), z_1, \ldots, z_n) \end{aligned} \tag{1}$$

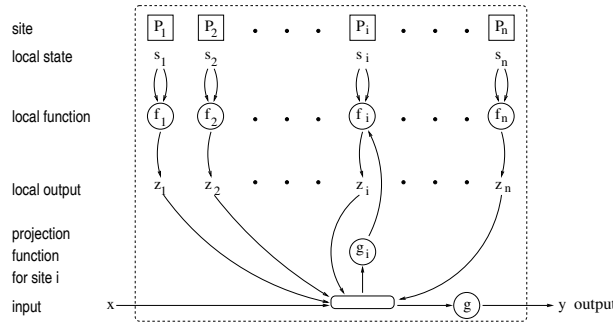


Fig. 1. Illustration of the possible data flow of a ParOp. f_i denotes the local computation of site P_i. g_i chooses values from $x, A_1, \ldots, A_n$. Only one projection function g_i is depicted so as to keep the illustration readable. The arrows do not indicate a cycle of data dependencies since the value provided by f_i need not be given back by g_i.

where ARG is a list of functions from $\{f_1, \ldots, f_n, g_0, g_1, \ldots, g_n\}$ and contains exactly those functions that are not fully defined within the body of the ParOp. The functions $f_1, \ldots, f_n, g_0, g_1, \ldots, g_n$ in the body of the ParOp definition can

- be defined as closed functions, so that the behavior of the ParOp is fully defined,
- define a class of functions, so that details have to be provided when using the ParOp in a program, or
- be left undefined, so that the entire function has to be provided when using the ParOp.

The functions that have to be provided when using the ParOp appear in the function argument list *ARG* as formal parameters in the definition (1) and as actual functions in a call of a ParOp.

The framework describes what a ParOp does, but not necessarily how it is implemented. In particular, the g_i functions imply data dependencies among the sites; these dependencies constrain the set of possible execution orders, but they may not fully define the order in an implementation. Consequently, the cost model for a ParOp may make additional assumptions about the execution order (for example, the degree of parallelism).

2.2 Using APMs in a Program

To express an application algorithm, the parallel operations defined for a specific APM are used and combined according to the coordination language. When using a ParOp in a program, one does not insert an entire ParOp definition. Instead, the operation is called along with any specific function arguments that are required. Whether function arguments are required depends on the definition of the specific ParOp.

If the function arguments f_i, g_i, are fully defined as functions in closed form, then no additional information is needed and the ParOp is called by just using its name. If one or more functions of f_i, g_i, $i = 1, \ldots, n$, are left undefined, then the call of this ParOp has to include the specific functions possibly restricted according to the class of functions allowed. A call of a ParOp has the form

$$\mathsf{ParOp} \quad ARG$$

where ARG contains exactly those functions of $(f_1, \ldots, f_n)(g_0, \ldots, g_n)$ that are needed. This might be given in the form

$$\begin{aligned} &f_{\kappa_1} = \textit{definition}, \ \ldots, \ f_{\kappa_l} = \textit{definition}, \\ &g_{\mu_1} = \textit{definition}, \ \ldots, \ g_{\mu_k} = \textit{definition}, \\ &\mathsf{ParOp}(f_{\kappa_1}, \ldots f_{\kappa_l}, g_{\mu_1}, \ldots g_{\mu_k}) \end{aligned}$$

2.3 Vertical and Horizontal Transformations between APMs

One goal of the APM approach is to model different parallel programming models within the same framework so that the relationship between two different models can be expressed. The relationship between two parallel programming models is based on the expressiveness of the APMs which is captured in the ParOps and the coordination language combining the ParOps.

We define a relation between two different APMs in terms of a transformation mapping any program for an APM M_1 onto a program for an APM M_2. The transformation is built up according to the structure of an APM program; thus it is defined on the ParOps of APM M_1 and then generalized to the entire coordination language. The transformation of a ParOp is based on its result semantics, i.e., the local data and output produced from input data at a given local state.

An APM M_1 can be simulated by another APM M_2 if for every ParOp F of M_1 there is ParOp G (or a sequence of ParOps $G_1, \ldots, G_l$) which have the same result semantics as F, i.e., starting with the same local states $s_1, \ldots, s_n$ and input data x it produces the same new local states $s'_1, \ldots, s'_n$ and output data y.

If an APM M_1 can be simulated by an APM M_2, this does not necessarily mean that M_2 can be simulated by M_1. If M_1 can be simulated by M_2, then M_1 is usually more abstract than M_2. Therefore, we arrange M_1 and M_2 in a hierarchical relationship with M_1 being the parent node of M_2. Considering an entire set of APMs, we get a tree or a forest showing a hierarchy of APMs and the relationship between them, see Figure 2.

The relationship between APMs serves as a basis for transforming an algorithm expressed on one APM to the same algorithm now expressed in the second related APM. For two related APMs M_1 and M_2 a transformation operation $T_{M_1}^{M_2}$ from M_1 to M_2 is defined according to the simulation relation, i.e., for each ParOp F of APM M_1

$$T_{M_1}^{M_2}(F) = G \quad (\text{ or } T_{M_1}^{M_2}(F) = G_1, \ldots, G_l)$$

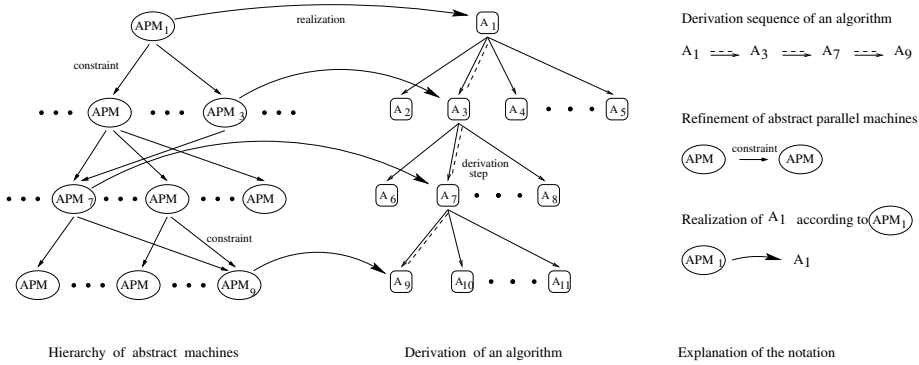


Fig. 2. Illustration of a hierarchy of abstract parallel machines and a derivation of an algorithm according to the hierarchy.

where G is the ParOp of M_2 to which F is related. In this kind of transformation step (which is called a *vertical* transformation), the program A_1 is left essentially unchanged, but it is realized on a different APM: thus (A_1, M_1) is transformed to (A_1', M_2). The operations F^j in A_1 are replaced by the transformation $T_{M_1}^{M_2}(F^j)$, so that A_2 uses the parallel operations in M_2 which realize the operations used by A_1.

There can also be a second kind of transformation (called a *horizontal* transformation) that takes place entirely within one APM M_1: (A_1, M_1) is transformed into (A_2, M_1), where a correctness-preserving transformation must be used to convert A_1 into A_2. In the context of our methodology, this means that a proof is required that for all possible inputs $X^0, \ldots, X^\kappa$ and states σ, the two versions of the algorithm must produce the same result, i.e.

$$A_1(X^0, \ldots, X^\kappa, \sigma) = A_2(X^0, \ldots, X^\kappa, \sigma).$$

There are several approaches for developing parallel programs by performing transformation steps, many of which have been pursued in a functional programming environment. Transformations based on the notion of homomorphism and the Bird-Meertens formalism are used in [10]. P3L uses a set of algorithmic skeletons like pipelines and worker farms to capture common parallel programming paradigms [3]. A parallel functional skeleton technique that emphasizes the data organization and redistributions is described in [16]. A sophisticated approach for the cost modeling of the composition of skeletons for a homomorphic skeleton system equipped with a equational transformation system is outlined in [20, 21]. The costs of the skeletons are required to be monotonic in the costs of the argument functions. The method performs a stepwise application of rewriting rules such that each application of a rewriting rule is cost-reducing. All these approaches restrict the algorithm to a single programming model, and they use the costs only to help select horizontal transformations. Vertical transformations between different programming models which could be used for the concretization of parallel programs are not supported.

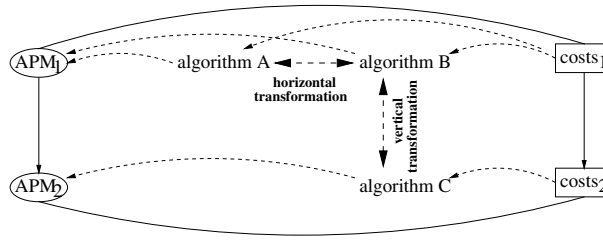


Fig. 3. Illustration of the connection between APMs, their corresponding costs and algorithms expressed using those APMs. The hierarchy contains only APM_1 and APM_2 with associated cost model costs_1 and costs_2. An algorithm A can be expressed within APM_1 and is transformed horizontally into algorithm B within the same APM which is then transformed vertically into algorithm C within APM_2.

3 Cost Hierarchies

The APM method proposed in [14] has separated the specifics of a parallel programming model from the properties of an algorithm to be expressed in the model. Since the APMs and the algorithm are expressed with a similar formalism, and the relations between APMs are specified precisely, it is possible to perform program transformations between different parallel programming models. In this section, we enrich the APM approach with a third component to capture costs, see Figure 3. The goal is to provide information that supports cost-driven transformations.

3.1 Cost Models for Leaf Machines

We consider an APM hierarchy whose leaves describe real machines with a programming interface for non-local operations. These would include, for example, a communication library for distributed memory machines (DMMs) or a coordination library for accessing the global memory of shared memory machines (SMMs) concurrently. Each operation of the real parallel machine is modeled by an operation of the corresponding leaf APM. By measuring the runtimes of the operations on the real parallel machine, the APM operations can be assigned costs that can be used to describe costs of a program. Since the execution times of many operations of the real machine depend on a number of parameters, the costs of the APM operations are described by parameterized runtime functions. For example, the costs of a broadcast operation on a DMM depend on the number p of participating processors and the size n of the message to be broadcast. Correspondingly, the costs are described by a function

$$t_{broad}(p, n) = f(p, n)$$

where f depends on the specific parallel machine and the communication library.

Usually, it is difficult to describe exactly the execution time of operations on real parallel machines. The local execution times are difficult to describe, since the processors may have a complicated internal architecture including a memory hierarchy, several functional units, and pipelines with different stages. Moreover techniques like branch prediction and out-of-order-execution may be used. The global execution times may be difficult to describe, since, for example, a distributed shared memory machine uses a physically distributed memory and emulates a shared memory by several caches, using, e.g., a directory-based cache coherence protocol. But for many (regular) applications and many DMMs, it is possible to describe the execution times of the machines accurately enough to compare different execution schemes for the same algorithm [23, 9]. The main concern of this article is not so much to give an accurate model for a specific (class of) parallel machines, but rather to extend an existing model so that it can be used at a higher programming level to compare different implementations of the same algorithm or to guide program transformations that lead to a more efficient program.

3.2 Bottom-Up Construction of a Cost Hierarchy

Based on the cost models for the leaves of an APM hierarchy, cost models for the inner nodes of an APM hierarchy can be derived step by step. We consider an APM M_1 which is the parent of an APM M_2 for which a cost measure C_2 has already been defined. At the beginning of the derivation M_2 has to be a leaf. Since M_1 is the parent of M_2, there is a transformation $T_{M_1}^{M_2}$ which assigns each parallel operation F of M_1 an equivalent sequence $G_1, \ldots, G_l$ of parallel operations, each of which has assigned a cost $C_2(G_i)$. We define a cost measure $C_{M_1 \leftarrow M_2}$ based on the cost measure C_2 for M_2 by

$$C_{M_1 \leftarrow M_2}(F) = \sum_{i=1}^{l} C_2(G_i). \tag{2}$$

A cost measure C_2 for M_2 may again be based on other cost measures, if M_2 is not a leaf. If the programmer intends to derive a parallel program for a real parallel machine R which is a leaf in the APM hierarchy, each intermediate level APM is assigned a cost measure that is based on the cost of R, i.e., the selection of the cost measure is determined by the target machine. Thus, for each path from a leaf to an inner node B there is a possibly different cost measure.

We now can define the equivalence of cost measures for an inner node M of an APM hierarchy with children M_1 and M_2. Cost measures $C_{M \leftarrow M_1}$ and $C_{M \leftarrow M_2}$ for APM M can be defined based on cost measures for C_1 and C_2 of M_1 and M_2, respectively. We call $C_{M \leftarrow M_1}$ and $C_{M \leftarrow M_2}$ *equivalent* if for arbitrary programs A_1 and A_2, the following is true:

$$\text{If } \; C_{M \leftarrow M_2}(A_1) \leq C_{M \leftarrow M_2}(A_2) \; \text{ then } \; C_{M \leftarrow M_1}(A_1) \leq C_{M \leftarrow M_1}(A_2)$$

and vice versa. If two cost measures are equivalent, then both measures can be used to derive efficient programs and it is guaranteed that both result in the

same program since both have the same notion of optimality. Note that the equivalence of two cost measures for an APM does not require that they yield the same cost value for each program.

3.3 Monotonicity of Cost Measures

The cost measure for an APM can be used to guide horizontal transformations. For this purpose, a cost measure must fulfil the property that a horizontal transformation that is useful for an APM M is also useful for all concretizations of M. This property is described more precisely by the notion of monotonicity of cost measures. We consider APMs M_2 and M_1 where M_2 is a child of M_1 in the APM hierarchy. Let A_1 and A_2 be two programs for APM M_1 where A_2 is obtained from A_1 by a horizontal transformation T_{M_1} which reduces the costs according to a cost measure C_1, i.e.,

$$C_1(A_1) \geq C_1(A_2) = C_1(T_{M_1}(A_1)).$$

Let $T_{M_1}^{M_2}$ be the vertical transformation from M_1 to M_2, i.e., the corresponding programs to A_1 and A_2 on APM M_2 are $A_1' = T_{M_1}^{M_2}(A_1)$ and $A_2' = T_{M_1}^{M_2}(A_2)$ respectively. Both these programs can be assigned costs according to a cost measure C_2 of APM M_2. The cost measures C_1 and C_2 are consistent only if the increase in efficiency that has been obtained by the transformation from A_1 to A_2 carries over to APM M_2, i.e., only if

$$C_2(T_{M_1}^{M_2}(A_1)) \geq C_2(T_{M_1}^{M_2}(A_2)).$$

This property is captured by the following definition of monotonicity. The transformation $T_{M_1}^{M_2}$ is monotonic with respect to the costs C_1 and C_2, if for arbitrary programs A_1 and A_2

$$C_1(A_1) \geq C_1(A_2) \text{ implies } C_2(T_{M_1}^{M_2}(A_1)) \geq C_2(T_{M_1}^{M_2}(A_2)).$$

The bottom-up construction of cost measures according to an APM hierarchy creates monotonic cost measures; this can be proven by a bottom-up induction over the APM tree using definition (2) of cost measures. In the next section, we describe the PRAM model with different operation sets in the APM methodology. For this example, we do not use the bottom-up cost definition but use the standard costs of the PRAM model.

3.4 Other Cost Models

One of the most popular parallel cost models is the PRAM model [8] and its extensions. Because none of the PRAM models was completely satisfactory, a number of other models have been proposed that are not based on the existence of a global memory, including BSP [22] and *logP* [6]. Both provide a cost calculus by modeling the target architecture with several parameters that capture its computation and communication performance. The supersteps in the BSP model

allow for a straightforward estimation of the runtime of complete programs [11]. Another cost modeling method is available for the skeleton approach used in the context of functional programming [18, 7]. In the skeleton approach, programs are composed of predefined building blocks (*data skeletons*) with a predefined computation and communication behavior which can be combined by *algorithmic skeletons* capturing common design principles like divide and conquer. This structured form of defining programs enables a cost modeling according to the compositional structure of the programs. A cost modeling with monads has been used in the GOLDFISH system [12]. In contrast to those approaches, APM costs allow the transformation of costs from different parallel programming models, so they can be used to guide transformations between multiple programming models.

4 APM Description of PRAMs

The PRAM (parallel random access machine) is a popular parallel programming model in theoretical computer science, widely used to design and analyze parallel algorithms for an idealized shared memory machine. A PRAM consists of a bounded set of processors and a common memory containing a potentially unlimited number of words [17]. Each processor is similar to a RAM that can access its local random access memory and the common memory. A PRAM algorithm consists of a number of *PRAM steps* which are performed in a SIMD-like fashion; i.e., all processors needed in the algorithm take part in a number of consecutive synchronized computation steps in which the same local function is performed. One PRAM step consists of three parts:

1. the processors read from the common memory;
2. the processors perform local computation with data from their local memories; and
3. the processors write results to the common memory.

Local computations differ because of the local data used and the unique identification number id_i of each processor P_i, for $i = 1, \ldots, n$ (where n is the number of processors).

4.1 An APM for PRAMs

There are many ways to describe the PRAM within the APM framework. In the PRAM model itself, the processors perform operations that cause values to be obtained from and sent to the common memory, but the behavior of the common memory itself is not modeled in detail. It is natural, therefore, to treat each PRAM processor as an APM site, and to treat the common memory as an implicit agent which is not described explicitly. The APM framework allows this abstract picture: transactions between the processors and the common memory are specified as Input/Output transactions between the APM operations and the surrounding environment.

The APM description of the PRAM model provides three ParOps for the PRAM substeps and a coordination language that groups the ParOps into steps and composes different steps while guaranteeing the synchronization between them. No other ParOps are allowed in a PRAM algorithm. The specific computations within the parallel operations needed to realize a specific algorithm are chosen by an application programmer or an algorithm designer. These local functions and the PRAM ParOps constitute a complete PRAM program.

The three ParOps of a PRAM step are READ, EXECUTE and WRITE. The state $(s_1, \ldots, s_n)$ denotes the data s_i in the local memory of processor P_i $(i = 1 \ldots n)$ involved in the PRAM program. The data from the common memory are the input $(x_1, \ldots, x_r)$ to the local computation and the data written back to the common memory are the output $(y_1, \ldots, y_r)$ of the computation.

1. In the read step, data from the common memory are provided and for each processor P_i the function g_i picks appropriate values from $(x_1, \ldots, x_r)$ which are then stored by the local behavior function $f_i = \mathsf{store}$ in the local memory, producing a new local state s'_i. There are no internal values produced in this step, so a dummy placeholder _ is used for the z_i term. The exact behavior of a specific READ operation is determined by the specific g_i functions, which the programmer supplies as an argument to READ. Thus the ParOp defined here is READ $(g_1, \ldots, g_n)$.

$$\begin{array}{rrcl} \mathsf{READ}\ (g_1, \ldots, g_n)\ (s_1, \ldots, s_n)\ (x_1, \ldots, x_r) & = & ((s'_1, \ldots, s'_n),\ (\_)) \\ \text{where} \qquad (s'_i, \_) & = & \mathsf{store}(s_i,\ g_i\ (V)) \\ V & = & ((x_1, \ldots, x_r), \_, \ldots, \_) \\ (\_, \ldots, \_) & = & g\ (V) \end{array}$$

2. In a local computation, each processor applies a local function f_i to the local data s_i in order to produce a new state s'_i. The substep for the local computation does not involve the common memory, so the input and output vectors x and y are empty. In this operation, the programmer must supply the argument function f, which determines the behaviors of the sites.

$$\begin{array}{rrcl} \mathsf{EXECUTE}\ (f, \ldots, f)\ (s_1, \ldots, s_n)\ (\_) & = & ((s'_1, \ldots, s'_n),\ ()) \\ \text{where} \qquad (s'_i, \_) & = & f(s_i, \_) \\ V & = & ((\_), \_, \ldots, \_) \\ (\_, \ldots, \_) & = & g\ (V) \end{array}$$

3. At the end of a PRAM step, data from the local states $(s_1, \ldots, s_n)$ are written back to the common memory. Each local function f_i selects data z_i from its local state s_i. From those data $(z_1, \ldots, z_n)$ the function g forms the vector $(y_1, \ldots, y_r)$, which is the data available in the common memory in the next step. If two different processors select the same variable d, then the function g_0 is capable of modelling the different write strategies corresponding to different PRAM models [17]. The programmer specifies the local fetch functions f_i to determine the values that are extracted from the local states in order to be sent to the common memory.

$$\begin{aligned}
\mathsf{WRITE}\ (f_1,\ldots,f_n)\ (s_1,\ldots,s_n)\ (_) &= ((s'_1,\ldots,s'_n),\ (y_1,\ldots,y_t)) \\
\text{where} \qquad (s'_i, z_i) &= f_i\ (s_i, _) \\
V &= ((), z_1, \ldots, z_n) \\
(y_1,\ldots,y_t) &= g\ (V)
\end{aligned}$$

The function $g((\_), A_1, \ldots, A_n) = (A_1, \ldots, A_n)$ produces an output vector with one value from each processor in the order of the processor numbers.

The PRAM steps can be combined by a sequential coordination language with for-loops and conditional statements. The next subsection gives an example.

4.2 Example Program: PRAM Multiprefix

More complex operations for PRAMs can be built up from the basic PRAM step. This section illustrates the process by defining the implementation of a multiprefix operation on a PRAM APM that lacks a built-in multi-prefix operation. Initially, the common memory contains an input array X with elements X_i, for $0 \leq i < n$; after the n sites execute a multiprefix operation with addition as the combining operation, the common memory holds a result array B, where $Y_i = \sum_{j=0}^{i} X_j$ for $0 \leq i < n$.

Several notations are used to simplify the presentation of the algorithm and to make it more readable, see Figure 4 (right). Upper case letters denote variables in the common memory, while lower case letters are used for the local site memories. The f and g functions are specified implicitly by describing abstract PRAM operations; the low level definitions of these functions are omitted. Thus the notation $\mathsf{READ}\ b_i := X_i$ means that a PRAM READ operation is performed with f and g functions defined so as to perform the parallel assignment. Similar conventions are used for the other operations, EXECUTE and WRITE. Furthermore, a constraint on the value of i in an operation means that the operation is performed only in those sites where the constraint is satisfied; other sites do nothing. (Again, the constraints are implemented through the definitions of the f and g functions.)

Figure 4 (left) gives the program realizing the multiprefix operation [2]. The algorithm first copies the array X into Y. This is done in $O(1)$ time by performing a parallel READ that stores X_i into site i, enabling the values to be stored in parallel into Y with a subsequent WRITE. Then a loop with $\log n$ steps is executed. In step j, each processor P_i (for $2^j \leq i < n$) reads the value accumulated by processor P_{i-2^j}, and it adds this to its local value of b_i. The other processors, P_i for $0 \leq i < 2^j$, leave their local b_i value unchanged. The resulting array is then stored back into Y in the common memory. All operations are executed in the READ/EXECUTE/WRITE scheme required by the PRAM model. The initialization, as well as each of the $\log n$ loop iterations, requires $O(1)$ time, so the full algorithm has time cost $O(\log n)$.

Initially: inputs are in $X_0, \ldots, X_{n-1}$.
Result: $Y_i = \sum_{j=0}^{i} X_j$ for $0 \leq i < n$

procedure parScanl $(\oplus, n, X, Y)$
  READ $b_i := X_i$ for $0 \leq i < n$
  EXECUTE _
  WRITE $Y_i := b_i$ for $0 \leq i < n$
  for $j := 0$ **to** $\log n - 1$ **do**
    READ $x_i := Y_{i-2^j}$ for $2^j \leq i < n$
    EXECUTE $b_i := x_i + b_i$ for $2^j \leq i < n$
    WRITE $Y_i := b_i$ for $0 \leq i < n$

READ $b_i := X_i$ is an abbreviation for
    READ with
    $g_i((x_0, \ldots, x_{n-1}), \_) = x_i$
    $s'_i = s_i[x_i/b_i],\ i = 0, \ldots, n-1$
EXECUTE $b_i := x_i + b_i$ abbreviates
    EXECUTE with
    $f(s_i, \_) = (s'_i, \_)$
    $s'_i = s_i[x_i + b_i/b_i]$
WRITE $Y_i := b_i$ is an abbreviation for
    WRITE with
    $f(s_i, \_) = (s'_i, b_i)$
    $g((\_), b_0, ..., b_{n-1}) = (Y_0, ..., Y_{n-1})$
    with $Y_i := b_i,\ i = 1, \ldots, n-1$

Fig. 4. Realization of a multiprefix operation expressed as an algorithm in a PRAM-APM.

4.3 PRAM* with Built-In Multiprefix

The PRAM model can be enriched with a multiprefix operation (also called parallel scan), which is considered to be *one* PRAM step. The steps of a program written in the enriched model may be either multiprefix operations or ordinary PRAM steps. The multiprefix operation with addition as a combining operation (MPADDL) has the same behavior as the *parScanl* procedure in Figure 4 (left); the only difference is that it is defined as a new PRAM ParOp, so its definition does not use any of the other ParOps READ, EXECUTE or WRITE.

$$\begin{aligned} &\text{MPADDL } (s_0, \ldots, s_{n-1})\ (X_0, \ldots, X_{n-1}) \ = \ ((s'_0, \ldots, s'_{n-1}),\ (Y_0, \ldots, Y_{n-1})) \\ &\quad \text{where} \qquad (s'_i, b_i) \ = \ f_i\ (s_i,\ g_i\ V) \\ &\qquad\qquad\quad V \ = \ ((X_0, \ldots, X_{n-1}), b_0, \ldots, b_{n-1}) \end{aligned}$$

where the functions $f_0, \ldots, f_{n-1}, g, g_0, \ldots, g_{n-1}$ are defined as follows:

$$g_i((X_0, \ldots, X_{n-1}), b_0, \ldots, b_{n-1}) = (X_0, \ldots, X_{i-1})$$

$$f_i(s_i, (X_0, \ldots, X_{i-1})) = ((s'_i, b_i) \text{ with } b_i = \sum_{j=0}^{i} X_j, i = 1, \ldots, n-1$$

$$g((X_0, \ldots, X_{n-1}), b_0, \ldots, b_{n-1}) = (Y_0, \ldots, Y_{n-1}) \text{ with } Y_i := b_i, i = 1, \ldots, n-1$$

A family of related operations can be treated as PRAM primitives in the same way. In addition to multiprefix addition from the left (MPADDL), we can also form the sums starting from the right (MPADDR). There are corresponding operations for other associative functions, such as maximum, for which we can define MPMAXL and MPMAXR. Reduction (or fold) ParOps for associative functions are also useful, such as FOLDMAX, which finds the largest of a set of values chosen from the sites. Several of these operations will be used later in an example (Section 5).

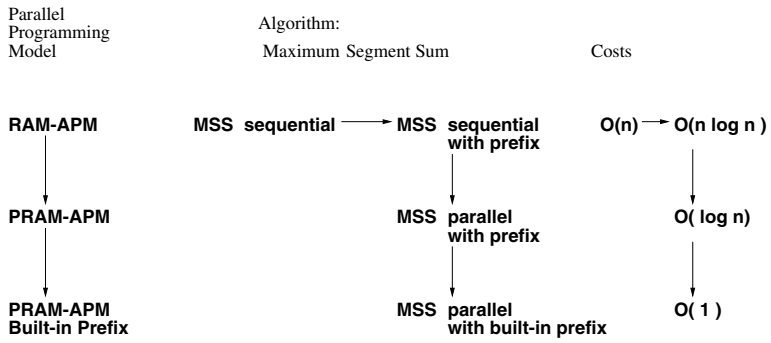


Fig. 5. Illustration of the transformation steps of the maximum segment sum.

4.4 PRAM Costs

The PRAM model defines the cost of an algorithm to be the number of PRAM steps that have to be performed by that algorithm, i.e., a PRAM step has cost 1. Thus, the cost of the multiprefix operation for the PRAM *without* a built-in multiprefix operation is the number of steps executed by the implementation in Figure 4. In the PRAM *with* built-in multiprefix, the cost for a multiprefix operation is 1. In Section 3.2, we have described a way to define costs starting from the leaves of the APM hierarchy. This is also possible for the PRAM and is useful if algorithms have to be transformed to a real machine. An example of a real machine with built-in multiprefix is the SB-PRAM [1], a multi-threaded architecture supporting multiprefix operations for integers in hardware. On this machine, each PRAM step and each multiprefix operation takes two cycles, independently from the data values contributed by the different processors. This can then be used as the cost measure of a leaf machine.

5 Example: MSS Algorithm Transformation

This section illustrates how an algorithm can be transformed within the APM framework in order to improve its efficiency. Figure 5 shows the organization of the transformation, which moves from the sequential RAM to the parallel PRAM models, and which includes both horizontal and vertical transformations.

We use as an example a version of the well-known maximum segment sum (MSS) problem similar to that presented by Akl [2]. Given a sequence of numbers $X = X_0, \ldots, X_{n-1}$, the problem is to find the largest possible sum *mss* of a contiguous segment within X. Thus we need the maximal value of $\sum_{i=u}^{v} X_i$ such that $0 \leq u \leq v < n$. Using prefix sums and maxima, the problem can be solved by the following steps:

1. For $i = 0, \ldots, n-1$ compute $S_i = \sum_{j=0}^{i} X_j$.
2. For $i = 0, \ldots, n-1$ compute $M_i = \max_{i \leq j < n} S_j$; let a_i be the value of j at which M_i is found.

3. For $i = 0, \ldots, n-1$ compute $B_i = M_i - S_i + X_i$.
4. compute $mss = \max_{0 \le i \le n-1} B_i$; if u is the index at which the maximum is found, the maximum sum subsequence extends from u to $v = a_u$.

The first version of the algorithm (Figure 6 left) is written in a conventional imperative style, except that the RAM programming model is made explicit. The RAM has only one ParOp, an EXECUTE operation, and the RAM-APM has only one site. The time of the algorithm is obtained by observing that the number of EXECUTE operations performed is $O(n)$, and each takes time $O(1)$ (this result can be obtained either by analyzing the RAM APM, or by assuming constant time for this leaf machine operation).

```
Initially: inputs are in X_0, ..., X_{n-1}.
Result: mss = max { sum_{i=u}^{v} X_i }
      for 0 ≤ u ≤ v < n

EXECUTE a := 0
for i := 0 to n − 1 do
  EXECUTE S_i := a + X_i
  EXECUTE a := S_i
EXECUTE a := NEGINF
for i := n − 1 downto 0 do
  EXECUTE M_i := max (S_i, a)
  EXECUTE a := M_i
for i := 0 to n − 1 do
  EXECUTE B_i := M_i − S_i + X_i
EXECUTE mss := NEGINF
for i := 0 to n − 1 do
  EXECUTE mss = max (mss, B_i)
```

```
procedure seqCopy (n, Y, Z)
  for i := 0 to n − 1 do
    EXECUTE Y_i := Z_i
procedure seqScanl (f, n, A, B)
  seqCopy (n, B, A)
  for j := 0 to log n − 1 do
    for i := 2^j to n − 1 do
      EXECUTE B_i := f (B_{i−2^j}, B_i)
procedure seqScanr (f, n, A, B)
  seqCopy (n, B, A)
  for j := 0 to log n − 1 do
    for i := n − 1 − 2^j downto 0 do
      EXECUTE B_i := f (B_i, B_{i+2^j})
function seqFoldl1 (f, a, n, A)
  EXECUTE q := a
  for i := 1 to n − 1 do
    EXECUTE q := f (q, A_i)
  return q
begin
  seqScanl (+, n, X, S)
  seqScanr (max, n, S, M)
  for i := 0 to n − 1 do
    EXECUTE B_i := M_i − S_i + X_i
  mss := seqFoldl1 (max, NEGINF, n, B)
end
```

Fig. 6. Algorithm 1 on the left: the sequential MSS expressed within the RAM-APM with time complexity $O(n)$ (NEGINF is the absolute value of the largest negative number.) and Algorithm 2 on the right: the sequential MSS using a sequential multiprefix operation expressed within the same . RAM-APM with $O(n \log n)$ time.

The aim is to speed up the algorithm using a parallel scan on the PRAM model. To prepare for this step, we first perform a horizontal transformation that restructures the computation, making it compatible with the prefix operations.

This results in Algorithm 2 (Figure 6 right). It is interesting to observe that the transformation has actually resulted in a slower algorithm; its benefit is to put us into a position to perform further transformations that will more than make up for this slow-down.

We now perform a vertical transformation from the RAM to the PRAM models, producing Algorithm 3 (Figure 7 (left)). As usual with vertical transformations, there is very little change to the structure of the algorithm; the main effect of the transformation is to use the PRAM to perform the iterations without data dependencies in $O(1)$ time. Algorithm 3 is still using the basic PRAM, so the parallel prefix operations require $O(\log n)$ time. The final step, Algorithm 4 (Figure 7 (right)), is produced by performing a vertical transformation onto the PRAM* model, which supports parallel prefix operations as built-in operations with a cost of $O(1)$ time.

procedure *parScanl* (f, n, A, B)
  Defined in Figure 4
procedure *parScanr* (f, n, A, B)
  Similar to *parScanl*
function *parFold* (f, n, A)
  Similar to *parScanl*
begin
  parScanl $(+, n, X, S)$
  parScanr (max, n, S, M)
  READ $m_i := M_i, s_i := S_i, x_i := X_i$
    for $0 \leq i < n$
  EXECUTE $b_i := m_i - s_i + x_i$
    for $0 \leq i < n$
  WRITE $B_i := b_i$ for $0 \leq i < n$
  $mss :=$ *parFold* (max, n, B)
end

MPADDL (n, X, S)
MPMAXR (n, S, M)
READ $m_i := M_i, s_i := S_i, x_i := X_i$
  for $0 \leq i < n$
EXECUTE $b_i := m_i - s_i + x_i$
  for $0 \leq i < n$
WRITE $B_i := b_i$ for $0 \leq i < n$
$mss :=$ FOLDMAX (n, B)

Fig. 7. Algorithm 3 on the left: MSS Parallel Scan. PRAM APM, $O(\log n)$ time. Algorithm 4 on the right: MSS Parallel Scan. PRAM* APM, $O(1)$ time.

6 Conclusion

The three components of the APM methodology—an APM with its ParOps, the ParOp cost models and the algorithm—reflect the abstract programming model, the architecture of a parallel system, and the parallel program. The architecture is represented here very abstractly in the form of costs.

In general, all three components of the methodology are essential. The costs are an important guide to the design of an efficient parallel algorithm. Separating the costs from the APMs allows several different ones to be associated

with the same operation, representing the same functionality implemented on different machines. However, it is not enough just to keep the semantics of the APM parallel operations and their costs: the APM definitions are still needed, as they make explicit the organization of data and computation into sites. Efficient algorithm design for parallel machines must consider not only *when* an computation on data is performed, but also *where*. An example is programming on a distributed memory machine where a poorly chosen distribution of data to sites may cause time-consuming communication.

We therefore conclude that all three components of the methodology are essential, but they should be separated from each other and made distinct. For solving a particular problem, some parts of the structure may not be needed, and can be omitted. For example, a programmer may find the intuition about cost provided by the ParOp definitions is sufficient, in which case the separate cost model structure is unnecessary.

References

[1] F. Abolhassan, J. Keller, and W.J. Paul. On the Cost–Effectiveness of PRAMs. In *Proc. 3rd IEEE Symp. on Parallel and Distributed Processing*, pages 2–9, 1991.

[2] S. G. Akl. *Parallel Computation—Models and Methods*. Prentice Hall, 1997.

[3] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P3L: A structured high level programming language and its structured support. *Concurrency: Practice and Experience*, 7(3):225–255, 1995.

[4] R. Bird and O. de Moor. *Algebra of Programming*. Prentice Hall, September 1996.

[5] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.

[6] D.E. Culler, R. Karp, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. *4th Symp. on Principles and Practice of Parallel Programming*, 28(4):1–12, 1993.

[7] J. Darlington, A.J. Field, P.G. Harrison, P.H.J. Kelly, D.W.N. Sharp, Q. Wu, and R.L. While. Parallel programming using skeleton functions. In *Proceedings of the PARLE'93*, volume 694 of *LNCS*, pages 146–160, Munich, Germany, June 1993.

[8] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In *Proceedings of the 10th ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[9] R. Foschia, T. Rauber, and G. Rünger. Modeling the Communication Behavior of the Intel Paragon. In *Proc. 5th Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'97)*, IEEE, pages 117–124, 1997.

[10] S. Gorlatch and C. Lengauer. Parallelization of divide-and-conquer in the bird-meertens formalism. *Formal Aspects of Computing*, 7(6):663–682, 1995.

[11] M. Hill, W. McColl, and D. Skillicorn. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.

[12] C.B. Jay, M. Cole, M. Sekanina, and P.A. Steckler. A Monadic Calculus for Parallel Costing of a Functional Language of Arrays. In *Proc. Euro-Par'97*, volume 1300 of *LNCS*, pages 650–661, 1997.

[13] J. O'Donnell. *Research Directions in Parallel Functional Programming*, chapter Data Parallelism. Springer Verlag, 1999.

[14] J. O'Donnell and G. Rünger. A methodology for deriving parallel programs with a family of abstract parallel machines. In *Euro-Par'97: Parallel Processing*, volume 1300 of *LNCS*, pages 662–669. Springer, August 1997. Passau, Germany.
[15] J. O'Donnell and G. Rünger. Abstract parallel machines. *Computers and Artificial Intelligence*, To Appear.
[16] P. Pepper. Deductive derivation of parallel programs. In *Parallel Algorithm Derivation and Program Transformation*, pages 1–53. Kluwer, 1993.
[17] J.E. Savage. *Models of Computation*. Addison Wesley, 1998.
[18] D. Skillicorn. Cost modeling. In K. Hammond and G. Michaelson, editors, *Research Direction in Parallel Functional Programming*, pages 207–218. Springer, 1999.
[19] D. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, 1998.
[20] D.B. Skillicorn. Towards a Framework for Cost-Based Transformations. *Journal of Systems and Architecture*, 42:331–340, 1995.
[21] D.B. Skillicorn and W. Cai. A Cost Calculus for Parallel Functional Programming. *Journal of Parallel and Distributed Computing*, 28(1):65–83, 1995.
[22] L.G. Valiant. A bridging model for parallel computation. *Comm. of the ACM*, 33(8):103–111, 1990.
[23] Z. Xu and K. Hwang. Early Prediction of MPP Performance: SP2, T3D and Paragon Experiences. *Parallel Computing*, 22:917–942, 1996.

Recursion Unrolling for Divide and Conquer Programs

Radu Rugina and Martin Rinard

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
{rugina, rinard}@lcs.mit.edu

Abstract. This paper presents *recursion unrolling*, a technique for improving the performance of recursive computations. Conceptually, recursion unrolling inlines recursive calls to reduce control flow overhead and increase the size of the basic blocks in the computation, which in turn increases the effectiveness of standard compiler optimizations such as register allocation and instruction scheduling. We have identified two transformations that significantly improve the effectiveness of the basic recursion unrolling technique. *Conditional fusion* merges conditionals with identical expressions, considerably simplifying the control flow in unrolled procedures. *Recursion re-rolling* rolls back the recursive part of the procedure to ensure that a large unrolled base case is always executed, regardless of the input problem size.
We have implemented our techniques and applied them to an important class of recursive programs, divide and conquer programs. Our experimental results show that recursion unrolling can improve the performance of our programs by a factor of between 3.6 to 10.8 depending on the combination of the program and the architecture.

1 Introduction

Iteration and recursion are two fundamental control flow constructs. Iteration repeatedly executes the loop body, while recursion repeatedly executes the body of the procedure. Loop unrolling is a classical compiler optimization. It reduces the control flow overhead by producing code that tests the loop termination condition less frequently. By textually concatenating copies of the loop body, it also typically increases the sizes of the basic blocks, improving the effectiveness of other optimizations such as register allocation and instruction scheduling.

This paper presents *recursion unrolling*, an analogous optimization for recursive procedures. Recursion unrolling uses a form of procedure inlining to transform the recursive procedures. Like loop unrolling, recursion unrolling reduces control flow overhead and increases the size of the basic blocks in the computation. But recursion unrolling is somewhat more complicated than loop unrolling. The basic form of recursion unrolling reduces procedure call overheads such as saving registers and stack manipulation. We have developed two transformations that optimize the code further. *Conditional fusion* merges conditionals

S.P. Midkiff et al. (Eds.): LCPC 2000, LNCS 2017, pp. 34–48, 2001.

with identical expressions, considerably simplifying the control flow in unrolled procedures and increasing the sizes of the basic blocks. *Recursion re-rolling* rolls back the recursive part of the procedure to ensure that a large unrolled base case is always executed, regardless of the problem size.

1.1 Divide and Conquer Programs

We have applied recursion unrolling to divide and conquer programs [10, 8, 5]. Divide and conquer algorithms solve problems by breaking them into smaller subproblems and recursively solving the subproblems. They use a base case computation to solve problems of small sizes and terminate the recursion.

Divide and conquer algorithms have several appealing properties that make them a good match for modern parallel machines. First, they tend to have a lot of inherent parallelism. The recursive structure of the algorithm naturally leads to recursively generated concurrency, which typically generates more than enough concurrency to keep the machine busy. Second, divide and conquer programs also tend to have good cache performance and to automatically adapt to different cache sizes and hierarchies. As soon as a subproblem fits into one level of the memory hierarchy, the program runs out of that level or below until the problem has been solved.

To fully exploit these properties, divide and conquer programs have to be efficient and execute useful computation most of the time, rather than spending substantial time on dividing problems into subproblems, or on combining subproblems. The size of the base case controls the balance between the computation time and the divide and combine time. If the base case is too small, the program spends most of its time in the divide and combine phases instead of performing useful computation. Unfortunately, the simplest and least error-prone coding styles reduce the problem to its minimum size (typically a problem size of one) before applying a very simple base case. Programmers therefore typically start with a simple program with a small base case, then unroll the recursion by hand to obtain a larger base case with better performance. This manual recursion unrolling is a tedious, error-prone process that obscures the structure of the code and makes the program much more difficult to maintain and modify.

The recursion unrolling algorithm presented in this paper automates the process of generating efficient base cases. It gives the programmer the best of both worlds: clean, simple divide and conquer programs with efficient execution.

1.2 Conditional Fusion

Since divide and conquer programs split the given problem into several subproblems, their recursive part typically contains multiple recursive calls. The size of the unrolled code therefore increases exponentially with the number of times the recursion is unrolled. Moreover, the control flow of the unrolled recursive procedure also increases exponentially in complexity. The typical structure of a divide and conquer program is a conditional with the base case on one branch and the

recursive calls on the other branch. Recursion unrolling generates an exponential number of nested if statements. To substantially simplify the control flow in the unrolled code, we apply a transformation called *conditional fusion* which merges conditional statements with equivalent test conditions. This transformation simplifies the generated code and improves the performance by reducing the number of conditional instructions and coalescing groups of small basic blocks into larger basic blocks.

1.3 Recursion Re-rolling

Recursion unrolling increases the code size both for the base case and for the recursive part. Compared to the recursive part of the original recursive program, the recursive part of the unrolled procedure divides the given problem into a larger number of smaller subproblems. This has the advantage that several recursive levels are removed from the recursion call tree. But this accelerated division into subproblems may generate base case subproblems of small size, even when the recursion unrolling produces unrolled base cases for larger problem sizes. To ensure that the computation always executes the more efficient larger base case, we apply another transformation, *recursion re-rolling*, which replaces the recursive part of the unrolled procedure with the recursive part of the original program.

1.4 Contributions

This paper makes the following contributions:

- **Recursion Unrolling:** It presents a new technique, recursion unrolling, for inlining recursive procedures. This technique iteratively constructs a set of unrolled recursive procedures. At each iteration, it conceptually inlines calls to recursive procedures.
- **Target programs:** It shows how to use recursion unrolling for an important class of recursive programs, divide and conquer programs. Recursion unrolling is used for these programs to automatically generate more efficient unrolled base cases of larger size.
- **Code Transformations:** It presents two new code transformations, conditional fusion and recursion re-rolling, that substantially improve the performance of recursive code resulting from recursion unrolling of divide and conquer programs. Both of these transformations reduce control flow overhead and increase the sizes of the basic blocks.
- **Experimental Results:** It presents experimental results that characterize the effectiveness of the algorithms on a set of benchmark programs. Our results show that the proposed code transformations can substantially improve the performance of our set of divide and conquer benchmark programs.

The remainder of the paper is organized as follows. Section 2 presents a running example that we use throughout the paper. Section 3 presents the analysis algorithms. Section 4 presents experimental results from our implementation. Section 5 discusses related work. We conclude in Section 6.

```
void dcInc(int *p, int n) {
  if (n == 1) {
    *p += 1;
  } else {
    dcInc(p,     n/2);
    dcInc(p+n/2, n/2);
  }
}
```

Fig. 1. Divide and conquer array increment example

2 Example

Figure 1 presents a simple example that illustrates the kinds of computations that our recursion unrolling is designed to optimize. The `dcInc` procedure implements a recursive, divide-and-conquer algorithm that increments each element of an array. In the divide part of the algorithm, the `dcInc` procedure divides each array into two subarrays. It then calls itself recursively to increment the elements in each subarray. After the execution of several recursive levels the program generates a subarray with only one element, at which point the algorithm uses the simple base case statement `*p += 1` to directly increment the single element of the subarray.

Reducing the problem to a base case of one is the simplest way to write the program. Larger base cases require a more complex algorithm, which in general can be quite difficult to correctly code and debug. But while the small base case is the simplest and easiest way to code the computation, it has a significant negative effect on the performance — the procedure spends most of its time dividing the problem into subproblems. The overhead of control flow, consisting of procedure calls and testing for the base case condition, overwhelms the useful computation. For each instruction that increments an array element, the computation executes at least one conditional instruction and one procedure call. To improve the efficiency of the program, the compiler has to reduce the control flow overhead.

The compiler can achieve control flow elimination in two ways. Procedure inlining can eliminate procedure call overhead. Fusing conditional statements with equivalent conditional expressions can eliminate redundant conditional statements. Our compiler applies both kinds of optimizations.

2.1 Inlining Recursive Procedures

The compiler first inlines the two recursive calls to procedure `dcInc`. Figure 2 shows the result of inlining these recursive calls. For this transformation, the compiler starts with two recursive copies of the original procedure `dcInc`, replaces their recursive calls with mutually recursive calls from one copy to the other, and then inlines one of them into the other. The resulting recursive procedure `dcIncI` has two base cases: the original base case for `n = 1` and a larger

```
void dcIncI(int *p, int n) {
  if (n == 1) {
    *p += 1;
  } else {
    if (n/2 == 1) {
      *p += 1;
    } else {
      dcIncI(p,       n/2/2);
      dcIncI(p+n/2/2, n/2/2);
    }
    if (n/2 == 1) {
      *(p+n/2) += 1;
    } else {
      dcIncI(p+n/2,       n/2/2);
      dcIncI(p+n/2+n/2/2, n/2/2);
    }
  }
}
```

Fig. 2. Program after inlining recursion

```
void dcIncF(int *p, int n) {
  if (n == 1) {
    *p += 1;
  } else {
    if (n/2 == 1) {
      *p += 1;
      *(p+n/2) += 1;
    } else {
      dcIncF(p,           n/2/2);
      dcIncF(p+n/2/2,     n/2/2);
      dcIncF(p+n/2,       n/2/2);
      dcIncF(p+n/2+n/2/2, n/2/2);
    }
  }
}
```

Fig. 3. Program after conditional fusion

base case for `n/2 = 1` (textually, this larger base case is split into two pieces in the generated code).

Compared to the original recursive procedure `dcInc` from Figure 1, the inlined procedure `dcIncI` also divides the given problem into a larger number of smaller subproblems. The inlined procedure generates four subproblems of quarter size, while the original procedure generates only two problems of half size. The transformation therefore eliminates half of the procedure calls in the dynamic call tree of the program.

2.2 Conditional Fusion

The inlined code in `dcIncI` also contains more conditionals and basic blocks than the original recursive code. Since inlining has exposed more code in the body of the procedure, the compiler can now perform intra-procedural transformations to simplify the control flow of the procedure body. In Figure 2 the compiler recognizes that the two if statements have identical test conditions `n/2 == 1`. It therefore applies another transformation, called *conditional fusion*, to replace the two conditional statements with a single conditional statement. Figure 3 presents the resulting recursive procedure `dcIncF` after this transformation. The true branch of the new conditional statement is the concatenation of the true branches of the initial if statements. Similarly, the false branch of the new conditional statement is the concatenation of the false branches of the initial if statements. The test condition of the merged conditional statement is the common test condition of the initial if statements.

```
void dcInc2(int *p, int n) {
  if (n == 1) {
    *p += 1;
  } else {
    if (n/2 == 1) {
      *p += 1;
      *(p+n/2) += 1;
    } else {
      if (n/2/2 == 1) {
        *p += 1;
        *(p+n/2/2) += 1;
        *(p+n/2) += 1;
        *(p+n/2+n/2/2) += 1;
      } else {
        dcInc2(p,                    n/2/2/2);
        dcInc2(p+n/2/2/2,            n/2/2/2);
        dcInc2(p+n/2/2,              n/2/2/2);
        dcInc2(p+n/2/2+n/2/2/2,      n/2/2/2);
        dcInc2(p+n/2,                n/2/2/2);
        dcInc2(p+n/2+n/2/2/2,        n/2/2/2);
        dcInc2(p+n/2+n/2/2,          n/2/2/2);
        dcInc2(p+n/2+n/2/2+n/2/2/2, n/2/2/2);
      }
    }
  }
}
```

Fig. 4. Program after second unrolling iteration

2.3 Unrolling Iterations

Because of the recursive structure of the program, the above transformations can be repeatedly applied. The recursive program `dcIncF` in Figure 3 represents the program after the first unrolling iteration. It performs the same overall computation as the original program `dcInc`, but it has a different internal structure.

The compiler can now use `dcIncF` and `dcInc` to unroll the recursion further. Since the two procedures perform the same computation, the compiler can safely replace their recursive calls with mutually recursive calls between each other and then inline one of them into the other. The compiler can further apply conditional fusion on the resulting recursive procedure. It thus produces the result of the second unrolling iteration, the recursive procedure `dcInc2` shown in Figure 4. It has a bigger base case for `n/2/2 == 1` and its recursive part divides the problem into an even larger number of smaller subproblems than `dcIncF`.

The recursion unrolling process can continue now by transforming recursive procedures `dcInc` and `dcInc2` into mutually recursive procedures, and then applying the above transformations. The unrolling process stops when the number of iterations reaches the desired unrolling factor.

```
void dcIncR(int *p, int n) {
  if (n == 1) {
    *p += 1;
  } else {
    if (n/2 == 1) {
      *p += 1;
      *(p+n/2) += 1;
    } else {
      if (n/2/2 == 1) {
        *p += 1;
        *(p+n/2/2) += 1;
        *(p+n/2) += 1;
        *(p+n/2+n/2/2) += 1;
      } else {
        dcIncR(p,     n/2);
        dcIncR(p+n/2, n/2);
      }
    }
  }
}
```

Fig. 5. Program after re-rolling

2.4 Re-rolling Recursion

Inlining recursive procedures automatically unrolls both the base case and the recursive part. Depending on the input problem size, the unrolled recursive part may lead to small base case subproblems that do not exercise the bigger, unrolled base cases. For instance, for procedure `dcInc2`, if the initial problem size is `n = 8`, the recursive calls will divide the problem into subproblems of size `n = 1`. Therefore, the bigger base case for `n == 4` does not get executed.

Since most of the time is spent at the bottom of the recursion tree, the goal of the compiler is to ensure that the bigger base cases are always executed. To obtain this goal, the compiler applies a final transformation, called *recursion re-rolling*, which rolls back the recursive part of the unrolled procedure. The result of re-rolling procedure `dcInc2` is shown in Figure 5, in figure `dcIncR`. The compiler detects that the recursive part of the initial procedure `dcInc` is executed on a condition which is always implied by the condition on which the recursive part of the unrolled procedure `dcInc2` is executed. The compiler can therefore safely replace the recursive part of `dcInc2` with the recursive part of `dcInc`, thus rolling back only the recursive part of the unrolled procedure. Thus, the recursive part of the procedure is unrolled only temporarily, to generate the base cases. After the large base cases are generated, the recursive part is rolled back.

Algorithm *RecursionUnrolling* (Proc f, Int m)

$f^{(0)}_{unroll}$ = clone(f);

for (i=1; i $\leq m$; i++)
  $f^{(i)}_{unroll}$ = RecursionInline($f^{(i-1)}_{unroll}$, f);
  $f^{(i)}_{unroll}$ = ConditionalFusion($f^{(i)}_{unroll}$);

$f^{(m)}_{reroll}$ = RecursionReRoll($f^{(m)}_{unroll}$, f);

return $f^{(m)}_{reroll}$

Fig. 6. Top Level of the recursion unrolling algorithm

3 Algorithms

This section presents in detail the algorithms that enable the compiler to perform the transformations presented in the previous section.

3.1 Top Level Algorithm

Figure 6 presents the top level algorithm for recursion unrolling. The algorithm takes two parameters: a recursive procedure f to unroll, and an unrolling factor m. The algorithm will unroll f m times. The algorithm iteratively builds a set $S = \{f^{(i)}_{unroll} \mid 0 \leq i \leq m\}$ of unrolled versions of the given recursive procedure. Different versions have base cases of different sizes. The internal structure of different versions is therefore different, but all versions perform the same computation.

The algorithm starts with a copy of the procedure f. This is the version of f unrolled zero times, $f^{(0)}_{unroll}$. Then, at each iteration i, the algorithm uses the version $f^{(i-1)}_{unroll}$ created in the previous iteration to build a new unrolled version $f^{(i)}_{unroll}$ of f with a bigger base case. To create the new version, the compiler inlines the original procedure f into the version from the previous iteration. The recursion inlining algorithm performs the inlining of recursive procedures. It takes two recursive versions from the set S and inlines one into another. After inlining, the compiler applies conditional fusion to simplify the control flow and coalesce conditional statements in the new recursive version $f^{(i)}_{unroll}$.

After it executes m iterations, the compiler stops the unrolling process. The last unrolled version $f^{(m)}_{unroll}$ has the biggest base case and the biggest recursive part. The compiler finally applies recursion re-rolling to roll back the recursive part of $f^{(m)}_{unroll}$.

```
Algorithm RecursionInline ( Proc f1, Proc f2 )

  Proc f3 = clone(f1);
  Proc f4 = clone(f2);

  foreach cstat ∈ CallStatements(f3, f3) do
    replace callee f3 in cstat with f4

  foreach cstat ∈ CallStatements(f4, f4) do
    replace callee f4 in cstat with f3

  foreach cstat ∈ CallStatements(f3, f4) do
    replace cstat with inlined procedure f4

  return f3
```

Fig. 7. Recursion inlining algorithm

3.2 Recursion Inlining

The recursion inline algorithm takes two recursive procedures, $f1$ and $f2$, which perform the same overall computation, and inlines one of them into the other. The result of this transformation is a recursive procedure with a base case bigger than any of the base cases of procedures $f1$ and $f2$.

Figure 7 presents the recursion inlining algorithm. Here, CallStatements(f, g) represents the set of procedure call statements with caller f and callee g. The compiler first creates two copies $f3$ and $f4$ of the parameter procedures $f1$ and $f2$, respectively. It then replaces each recursive call in $f3$ and $f4$ with calls to the other procedure. Because $f1$ and $f2$ perform the same computation, each of the new mutually recursive procedures $f3$ and $f4$ will perform the same computation as the original procedures $f1$ and $f2$. With direct recursion translated into mutual recursion, each call statement has a different caller and callee. This enables procedure inlining at the mutually recursive call sites. The compiler therefore inlines the procedure $f4$ into $f3$. The resulting inlined version of $f3$ is a recursive procedure which performs the same computation as the given procedures $f1$ and $f2$, but has a bigger base case and splits a given problem into a larger number of smaller subproblems.

3.3 Conditional Fusion

Conditional fusion is an intra-procedural transformation that merges conditional `if` statements with equivalent condition expressions. The conditional fusion algorithm searches the control flow graph of the unrolled procedure for consecutive conditional statements with this property.

```
Algorithm ConditionalFusion ( Proc f )

  foreach meta-basic-block B
      in bottom-up traversal of f do

    Boolean failed = false
    Statement newcond

    foreach meta-statement stat in B do
      if ( not IsConditional(stat) ) then
        failed = true
        break
      else if ( IsEmpty(newcond) )
        newcond = clone(stat)
      else if ( not SameCondition(newcond, stat) )
        failed = true
        break
      else
        Append(newcond.True, stat.True);
        Append(newcond.False, stat.False);

    if (not failed) then
      replace B with newcond

  return f
```

Fig. 8. Conditional fusion algorithm

For detecting such patterns, a *hierarchically* structured control flow graph is more appropriate. A hierarchical control flow graph is a graph of *meta-basic-blocks*. A meta-basic-block is a sequence of *meta-statements*. A meta-statement is either a program instruction, a conditional statement, or a loop statement. There is no program instruction that jumps in or out of a meta-basic-block. Bodies of loop statements and branches of conditional statements are, in turn, hierarchical control flow graphs.

Using a hierarchical control flow graph representation, the conditional fusion algorithm is formulated as shown in Figure 8. The compiler traverses the hierarchical control flow structure in a bottom-up fashion. At each level, it inspects the meta-statements in the current basic block B. It checks if all the meta-statements in B are conditional statements and if they all have equivalent condition expressions. If not, the *failed* flag is set to true and no transformation is performed. When checking the equivalence of condition expressions, the compiler also verifies that the conditional statements do not write any of the variables of the condition expressions. This ensures that condition expressions of

```
Algorithm RecursionReRoll ( Proc f1, Proc f2 )

  MetaBasicBlock B1 = RecursivePart(f1)
  MetaBasicBlock B2 = RecursivePart(f2)

  Boolean cond1 = RecursionCondition(f1)
  Boolean cond2 = RecursionCondition(f2)

  if ( cond1 implies cond2 ) then
    replace calls in B2 to f1 with calls to f2
    replace B1 with B2 in procedure f1

  return f1
```

Fig. 9. Recursion re-rolling algorithm

different if statements refer to variables with the same values. As it checks the statements, the compiler starts building the merged `if` statement. If *stat* is the current conditional statement, the compiler appends its true branch to the true branch of the new conditional, and its false branch to the false branch of the new conditional. After scanning the whole basic block, if the flag *failed* is not set, the compiler replaces B with the newly constructed conditional statement.

3.4 Recursion Re-rolling

The recursion re-rolling transformation rolls back the recursive part of the unrolled procedure, leaving the unrolled base case unchanged. It ensures that the largest unrolled base case is always executed, regardless of the input problem size.

Figure 9 presents the algorithm for recursion re-rolling. The algorithm is given two procedures which are versions of the same recursive computation. Procedure $f1$ has an unrolled recursive part and procedure $f2$ has a rolled recursive part. To re-roll the recursion of $f1$, the compiler first identifies the recursive parts two procedures, $B1$ and $B2$ respectively. The recursive part of a procedure is the smallest meta-basic-block in the procedure that contains all the recursive calls and which represents the whole body of the procedure when executed. The compiler then detects the conditions on which the recursive parts are executed. If the condition *cond*1 on which the recursive part of the unrolled procedure is executed always implies the condition *cond*2 on which the rolled recursion of $f2$ is executed, then the compiler performs the re-rolling transformation. Knowing that both $f1$ and $f2$ perform the same computation, the compiler first replaces calls to $f2$ in $B2$ with calls to $f1$. It then replaces block $B1$ with block $B2$ to complete rolling back the recursive part of $f1$.

Machine	Input Size	Unrolling Types 0u	1u	1u+f	1u+fr	2u	2u+f	2u+fr	3u+fr	Hand Coded
Pentium III	512	9.22	3.41	2.83	2.97	11.49	9.34	2.69	2.55	1.16
Pentium III	1024	73.80	69.43	64.75	23.61	32.51	24.63	20.70	20.47	9.19
PowerPC	512	14.35	4.19	3.28	2.89	17.32	25.19	1.63	1.33	0.59
PowerPC	1024	114.60	136.84	137.20	23.17	33.87	35.91	12.91	10.74	4.69
Origin 2000	512	30.57	9.92	6.77	6.84	30.54	29.81	3.95	3.62	1.24
Origin 2000	1024	244.44	239.64	230.43	54.59	81.13	58.55	31.46	28.73	9.90

Table 1. Running times of unrolled versions of Mul (seconds)

Machine	Input Size	Unrolling Types 0u	1u	1u+f	1u+fr	2u	2u+f	2u+fr	3u+fr	Hand Coded
Pentium III	512	3.14	2.15	2.00	0.99	1.86	1.32	0.84	0.83	0.71
Pentium III	1024	24.77	14.62	13.14	8.53	21.25	15.86	6.99	6.58	5.48
PowerPC	512	4.88	4.15	4.01	1.16	2.41	2.25	0.73	0.66	0.70
PowerPC	1024	39.16	26.58	24.91	9.46	29.20	32.33	5.91	5.33	5.64
Origin 2000	512	10.77	7.34	6.56	2.42	5.06	3.31	1.39	1.26	1.20
Origin 2000	1024	85.86	48.47	40.98	19.20	54.56	44.53	10.96	9.95	9.57

Table 2. Running times of unrolled versions of LU (seconds)

4 Experimental Results

We used the SUIF compiler infrastructure [1] to implement the recursion unrolling algorithms presented in this paper. We present experimental results for two divide and conquer programs:

- **Mul:** Divide and conquer blocked matrix multiply. The program has one recursive procedure with 8 recursive calls and a base problem size of a matrix with one element.
- **LU:** Divide and conquer LU decomposition. The program has four mutually recursive procedures. Each of them has a base problem size of a matrix with one element. The main recursive procedure has 8 recursive calls.

We implemented our compiler as a source-to-source translator. It takes a C program as input, locates the recursive procedures, then unrolls the recursion to generate a new C program. We then compiled and ran the generated C programs on three machines: a Pentium III machine running Linux, a PowerPC running Linux, and an SGI Origin 2000 running IRIX.

Table 1 presents the running times for various versions of the Mul program. Each column is labeled with the number of times that the compiler unrolled the recursion; we report results for the computation unrolled 0, 1, 2, and 3 times. If the column is labeled with an f, it indicates that compiler applied the conditional fusion transformation. If the column is labeled with an r, it indicates that the compiler applied the recursion re-rolling transformation. So, for example, the column labeled 1u+fr contains experimental results for the version with the recursion unrolled once and with both conditional fusion and recursion re-rolling. Depending on the architecture, the best automatically unrolled version of program Mul performs between 3.6 to 10.8 times better than the unoptimized version Table 2 presents the running times for various versions of the LU decomposition program. Depending on the architecture, the best automatically unrolled version of this program performs between 3.8 to 8.6 times better than the unoptimized version.

We also evaluate our transformations by comparing the performance of our automatically generated code with that of several versions of the programs with optimized, hand coded base cases. We obtained these versions from the Cilk benchmark set [9]. The last column in Tables 1 and 2 presents the running times of the hand coded versions. The best automatically unrolled version of Mul performs between 2.2 and 2.9 worse than the hand optimized version. The performance of the best automatically unrolled version of LU is basically comparable to that of the hand coded version. These results show that our transformations can generate programs whose performance is close to, and in some cases identical to, the performance of programs with hand coded base cases.

4.1 Impact of Re-rolling

The running times in Tables 1 and 2 emphasize the impact of recursion re-rolling on the performance of the program. Whenever the unrolled recursion makes the program skip its largest unrolled base case, recursion re-rolling can deliver substantial performance improvements. For instance, in the case of program Mul with recursion unrolled twice, running on the Origin 2000, on a matrix of 512x512 elements, recursion re-rolling dramatically improves the performance — with recursion re-rolling, the running time drops from 29.81 seconds to 3.95 seconds. For this example, recursion inlining and conditional fusion produce additional base cases of sizes 2 and 4. But because the unrolled recursive part divides each problem in several subproblems of size 1/8 at each step, these base cases never get executed. The program always executes the inefficient base case of size 1. Re-rolling the recursion ensures that the efficient base case of size 4 gets always executed.

The structure of the inlined recursion also explains why programs whose recursive part is not re-rolled may perform worse after several inlining steps. For instance, the Mul program on a Pentium III has a surprisingly high running time of 11.49 seconds after two unrolling iterations, compared to its fast execution of 3.41 seconds after a single unrolling iteration. The reason for this performance difference is that, for the given problem size of 512, the problem is always reduced

to a base case of size 2 when recursion is unrolled once, while the program always ends up executing the smaller and inefficient base case of size 1 when recursion is unrolled twice.

Finally, our results for different problem sizes show that the impact of re-rolling depends on the problem size. For LU running on the PowerPC, with recursion unrolled twice, the version with re-rolling runs 3.08 times faster than the original version for a matrix of 512x512 elements, and 5.47 times faster than the original version for a matrix of 1024x1024 elements. The fact that the size of the base case that gets executed before re-rolling depends on the problem size explains this discrepancy.

4.2 Impact of Conditional Fusion

Our results show that conditional fusion can achieve speedups of up to 1.5 over the versions without conditional fusion, as in the case of the Mul program running on the SGI Origin 2000, on a matrix of 512x512 elements, with recursion unrolled once. In the majority of cases, fusion of conditionals improves program performance. In a few cases, though, the modified cache behavior after conditional restructuring causes a slight degradation in the performance.

The major advantage of conditional fusion transformation is that it enables recursion re-rolling, which has significant positive impact on the running times. To apply recursion re-rolling, the compiler has to identify and separate the recursive parts and the base cases. Conditional fusion is the key transformation that enables the compiler to identify these parts in the unrolled code.

5 Related Work

Procedure inlining is a classical compiler optimization [3, 2, 4, 6, 12, 7, 11]. The usual goal is to eliminate procedure call and return overhead and to enable further optimizations by exposing the combined code of the caller and callee to the intraprocedural optimizer. Some researchers have reported a variety of performance improvements from procedure inlining; others have reported that procedure inlining has relatively little impact on the performance.

Our initial recursion unrolling transformation is essentially procedure inlining. We augment this transformation with two additional transformations, conditional fusion and recursion re-rolling, that significantly improve the performance of our target class of divide and conquer programs. We therefore obtain the benefit of a reduction in procedure call and return overhead. We also obtain more efficient code that eliminates redundant conditionals and sets up the recursion so as to execute the efficient large base case most of the time.

In general, we report much larger performance increases than other researchers. We attribute these results to several factors. First, we applied our techniques to programs that heavily use recursion and therefore suffer from significant overheads that recursion unrolling can eliminate. Second, conditional fusion and recursion re-rolling go beyond the standard procedure inlining transformation to further optimize the code.

6 Conclusion

This paper presents recursion unrolling, a technique for improving the performance of recursive computations. Like loop unrolling, recursion unrolling reduces control flow overhead and increases optimization opportunities by generating larger basic blocks. But recursion unrolling is somewhat more complicated than loop unrolling. The basic form of recursion unrolling reduces procedure call overheads such as saving registers and stack manipulation. We have developed two transformations that optimize the code further. *Conditional fusion* merges conditionals with identical expressions, considerably simplifying the control flow in unrolled procedures. *Recursion re-rolling* rolls back the recursive part of the procedure to ensure that the biggest unrolled base case is always executed, regardless of the problem size.

References

[1] S. Amarasinghe, J. Anderson, M. Lam, and A. Lim. An overview of a compiler for scalable parallel machines. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[2] Andrew W. Appel. Unrolling recursion saves space. Technical Report CS-TR-363-92, Princeton University, March 1992.

[3] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of the SIGPLAN '89 Conference on Program Language Design and Implementation*, Portland, OR, June 1989. ACM, New York.

[4] P. Chang, S. Mahlke, W. Chen, and W. Hwu. Profile-guided automatic inline expansion for C programs. *Software—Practice and Experience*, 22(5):349–369, May 1992.

[5] S. Chatterjee, A. Lebeck, P. Patnala, and M. Thottethodi. Recursive array layouts and fast matrix multiplication. In *Proceedings of the 11th Annual ACM Symposium on Parallel Algorithms and Architectures*, Saint Malo, France, June 1999.

[6] K. Cooper, M. W. Hall, and L. Torczon. An experiment with inline substitution. *Software—Practice and Experience*, 21(6):581–601, June 1991.

[7] J. W. Davidson and A. M. Holler. A study of a C function inliner. *Software—Practice and Experience*, 18(8):775–790, August 1988.

[8] J. Frens and D. Wise. Auto-blocking matrix-multiplication or tracking BLAS3 performance from source code. In *Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Las Vegas, NV, June 1997.

[9] M. Frigo, C. Leiserson, and K. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the SIGPLAN '98 Conference on Program Language Design and Implementation*, Montreal, Canada, June 1998.

[10] F. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–755, November 1997.

[11] S. Richardson and M. Ganapathi. Interprocedural analysis versus procedure integration. *Information Processing Letters*, 32(3):137–142, August 1989.

[12] R. Scheifler. An analysis of inline substitution for a structured programming language. *Commun. ACM*, 20(9), September 1977.

An Empirical Study of Selective Optimization*

Matthew Arnold[1,2], Michael Hind[2], and Barbara G. Ryder[1]

[1] Rutgers University
110 Frelinghuysen Road, Piscataway, NJ 08854, USA
{marnold,ryder}@cs.rutgers.edu
[2] IBM Thomas J. Watson Research Center,
P.O. Box 704, Yorktown Heights, NY 10598, USA
hind@watson.ibm.com

Abstract. This paper describes an empirical study of selective optimization using the Jalapeño Java virtual machine. The goal of the study is to provide insight into the design and implementation of an adaptive system by investigating the performance potential of selective optimization and identifying the classes of applications for which this performance can be expected. Two types of offline profiling information are used to guide selective optimization, and several strategies for selecting the methods to optimize are compared.
The results show that selective optimization can offer substantial improvement over an optimize-all-methods strategy for short-running applications, and for longer-running applications there is a significant range of methods that can be selectively optimized to achieve close to optimal performance. The results also show that a coarse-grained sampling system can provide enough accuracy to successfully guide selective optimization.

1 Introduction

One technique for increasing the efficiency of Java applications is to compile the application to native code, thereby gaining efficiency over an interpreted environment. To satisfy Java's dynamic semantics, such as dynamic class loading, most compilation-based systems perform compilation at runtime. However, any gains in application execution performance achieved by such JIT ("Just In Time") systems must overcome the cost of application compilation.

An alternative strategy is to *selectively optimize* the methods of an application that dominate the execution time with the goal of incurring the cost of optimization for only those methods in which the performance of compiled code will be most beneficial. One example of this approach is to use two compilers: a quick nonoptimizing compiler and an optimizing compiler. With this compile-only strategy, all methods are initially compiled by the nonoptimizing compiler and only selective methods are optimized.

* Funded, in part, by IBM Research and NSF grant CCR-9808607.

S.P. Midkiff et al. (Eds.): LCPC 2000, LNCS 2017, pp. 49–67, 2001.

The goal of this work is to empirically evaluate the limitations of selective optimization. Although systems that selectively optimize methods during execution do exist [22, 24, 32, 8, 15, 6] the focus of such work is typically overall system performance, which is affected by many characteristics of the particular system being studied. These characteristics include 1) when a recompilation decision is made, 2) the overhead and online nature of the profiling information, 3) the sophistication and overhead of the recompilation decision-making component, and 4) the efficiency and performance increase obtained by using the optimizing compiler. This study holds these factors constant to provide a comprehensive evaluation of the effectiveness of selective optimization. Namely, 1) all optimization occurs before execution; 2) offline profiling is used; 3) optimization decisions are made before execution is begun; and 4) the Jalapeño optimizing compiler [12] is used.

In this work we

- establish the performance gains possible for selective optimization and identify classes of applications for which these gains can be expected;
- define metrics that characterize the performance of selective optimization and report their values;
- compare the performance of selective optimization when driven by two different profiling techniques, time-based and sample-based; and
- investigate different strategies for choosing methods to optimize.

The results provide useful insight for the design and implementation of an adaptive system. We have observed that selective optimization can offer substantial improvement over an all-opt (optimize all executing methods) strategy for our short-running benchmark suite, and for the longer-running suite there is a significant range of methods that can be selected to achieve close to optimal performance. The results also show that a coarse-grained sampling system can provide the accuracy necessary to successfully guide selective optimization, suggesting that the performance results may be obtainable with an adaptive system. Indeed, the results from this work were used during the design of the Jalapeño adaptive system [6].

The rest of this paper elaborates on these points and is organized as follows. Sections 2 and 3 describe the background and methodology used in this study. Section 4 presents the main results and direct conclusions. Section 5 investigates the performance of several automatic method selection strategies. Section 6 describes related work. Section 7 draws conclusions and discusses these findings with respect to adaptive optimization.

2 Background

The results presented in this study were obtained using the Jalapeño JVM [3, 4] being developed at IBM T.J. Watson Research Center. Except for a small amount of code that manages operating system services, Jalapeño is written completely in Java [4]. In addition to eliminating language barriers between the application

and the JVM, writing a JVM in Java also enables optimization techniques to be applied to the JVM itself, such as inlining JVM code directly into the application and adaptively optimizing the JVM to the application.

An advangage of a compile-only approach is that optimized and unoptimized code can be used together, without the overhead of entering and exiting interpretation mode. Jalapeño currently contains two fully operational compilers, a nonoptimizing *baseline* compiler and an *optimizing* compiler [12]. The version of the optimizing compiler used for this study performs the following optimizations:

- on-the-fly optimizations during bytecode to IR translation, such as copy propagation, constant propagation, dead code elimination, and register renaming for local variables [34];
- flow-insensitive optimizations, such as scalar replacement of aggregates and the elimination of local common subexpressions, redundant bounds checks (within an extended basic block [14]) and redundant local exception checks;
- semantic expansion transformations [35] of standard Java library classes and static inlining, including the inlining lock/unlock and allocation methods.

These optimizations are collectively grouped together into the default optimization level (level 1). More sophisticated optimizations are under development.

Both the nonoptimizing (baseline) and optimizing compilers compile one method at a time, producing native RS/6000 AIX machine code. On average the nonoptimizing compiler is 76 times faster than the optimizing compiler used in this study and the resulting performance benefit can vary greatly. For our benchmark suite, the execution time speedup ranges from 1.22–8.17 when the application is completely compiled by the optimizing compiler.

Jalapeño provides a family of interchangeable memory managers [3] with a concurrent object allocator and a stop-the-world, parallel, type-accurate garbage collector. This study uses the nongenerational copying memory manager.

3 Methodology

The experimental results were gathered on an IBM RS/6000 PowerPC 604e with two 333MHz processors and 1048MB RAM, running AIX 4.3. Both processors were free to be used during the execution of the benchmarks, however, no background or parallel compilation is performed; all compilation occurs pre-execution, and Jalapeño currently does not perform parallel compilation. The study uses the SPECjvm98[1] benchmark suite [17], Jalapeño's optimizing compiler [12], the pBOB multithreaded business transaction benchmark [9], and the Volano benchmark [33], which is a multithreaded server application that simulates chat rooms. The benchmarks range in cumulative class file sizes from 10,156 (209_db) to 1,516,932 (opt-cmp) bytes.

[1] These results do *not* follow the official SPEC reporting rules, and therefore should not be treated as official SPEC results.

The performance runs are grouped into two categories: *short running* and *longer running*. The short-running category includes the SPECjvm98 benchmarks using the size 10 (medium) inputs and the Jalapeño optimizing compiler compiling a small subset of itself to give a comparable runtime. The `pBOB` and `Volano` benchmarks are designed to be longer-running server benchmarks and thus they are not included in the shorter running benchmarks. The longer-running category includes the SPECjvm98 benchmarks using the size 100 (large) inputs, the Jalapeño optimizing compiler compiling a larger subset of itself, and the `pBOB` and `Volano` benchmarks. The `pBOB` and `Volano` benchmarks run for a fixed amount of time and report their performance as a throughput metric in events per second. To allow comparison, we report an equivalent metric of seconds per 750,000 transactions for `pBOB` and seconds per 140,000 messages for `Volano`.

Each experiment first compiles all application methods using the nonoptimizing compiler (as is the case in a compile-only adaptive system) to ensure that all classes will be loaded when the optimizing compiler is invoked. This avoids (in optimized code) the performance penalties associated with dynamic linking, i.e., calls to methods of classes that have not been loaded, and also allows the called method to be a candidate for inlining. Next, a selected number of application methods are compiled again by the optimizing compiler. Finally, the application is executed without any further compilation being performed. Thus, the application's execution time contains no compilation overhead.

Each benchmark is executed a repeated number of times, where the number of methods optimized, N, is increased. These methods are chosen in order of hotness, based on a past profile of the unoptimized application with the same input. Thus, the number of methods that can be optimized is bound by the number of profiled methods. For each benchmark we chose the following values of N: 0, 1, 2, ..., 19, 20, 30, 40, ..., # profiled methods. We use the term "all methods" to refer to all methods present in the profile.

For each value of N, three quantities are measured: nonoptimizing compilation time (of all methods), optimization time (of N methods), and execution time. We refer to the total of these three times as the *combined time*. All times reported are the minimum of 4 runs of the application for the given value of N. The startup time of the Jalapeño virtual machine is not counted in the timings because it would simply add a constant factor to all times. Optimizing startup time of a JVM is a separate research topic.

3.1 Profiling

Two types of profiling information are used to determine the order in which methods are selected for optimization: *time-based* and *sample-based*. The time-based profile records time spent in each method, and is collected by instrumenting all calls, returns, throws, and catches to check the current time and add it to the appropriate method. Methods are ordered based on the total time spent in the method.

The sampling-based profiling technique samples the running application approximately once every 10 milliseconds when the application reaches predefined sample points. These points are located on entry to a method and on loop back edges. Samples taken on loop back edges are attributed to the method containing the loop. Samples taken on method entries are attributed to both the calling and called method. Methods are ordered based on the total number of samples attributed to them. This sampling technique is efficient — the overhead introduced does not stand out above the noise from one run to the next — and it is being used in the initial Jalapeño adaptive system [6].

The method orderings from both profiles are likely to contain a certain degree of imprecision. Sampling every 10 milliseconds is fairly coarse-grained considering the speed of the processor; thus the sampled profile will contain some imprecision, especially for short-running applications. The time-based profile is also imprecise because the instrumentation introduces overhead, increases code size, and disrupts the instruction cache.

A major difference between two profiles is that the time-based profile is complete in the sense that all executing methods will appear in the profile. The sample-based profile is an incomplete, or partial profile, because it is possible for a method to execute but never be sampled.

4 Results

This section presents the main results of the study. Section 4.1 describes the results using the time-based profile. Section 4.2 describes the results using the sample-based profile and compares these results to those from Section 4.1.

4.1 Time-Based Profile

Each application in the two suites can be plotted on a graph where the x-axis measures the number of (hot) methods selected for optimization and the y-axis measures time in seconds. Fig. 1 shows representative graphs for the short-running and longer-running suites using the time-based profile.[2] Each graph contains two curves, a solid curve connects points of the form (number-of-optimized-methods, combined time). A dashed curve connects points of the form (number-of-optimized-methods, execution time), i.e., the dashed curve does not include any compilation time. The time to compile all methods with the nonoptimizing compiler is included in the combined time (solid curve). This time ranged from 0.06–0.63 seconds, averaging 0.19 seconds.

One observation to make is the execution time difference between the short-running and longer-running benchmark suites. With all methods optimized, the execution time only (the rightmost point on the dotted curve) is in the range of 1.1–4.8 seconds for the short-running benchmarks, and 16.6–70.6 seconds for longer-running benchmarks. Thus it is not surprising that compilation time is a

[2] Due to space constraints we are unable to show all graphs.

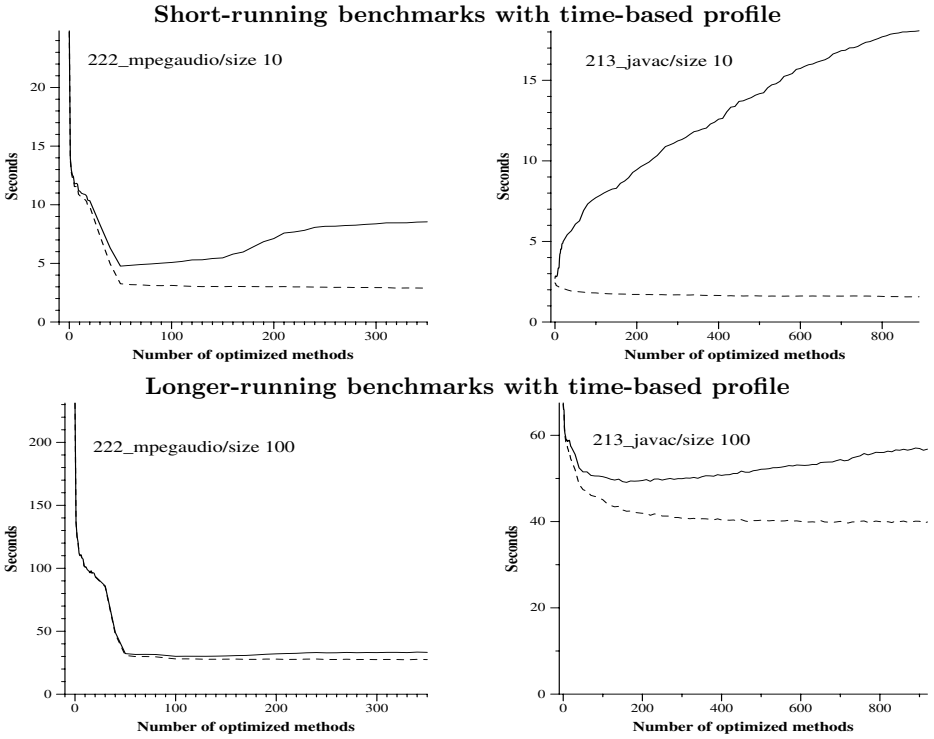


Fig. 1. Graphs for representative short-running and longer-running benchmark suites for the time-based profiles. The x-axis is the number of methods that are optimized. The y-axis is time in seconds. The dashed curve connects points for execution time. The solid curve connects points for the combined time (execution + compilation). x-axis points are recorded for $(0, 1, ..., 20, 30, 40, ..., \#$ profiled methods).

larger percentage of combined time for short-running benchmarks than for the longer-running benchmarks.

Another trend across the benchmarks is a steep initial drop in the curves, reaching a minimum rather quickly. After the combined time (solid curve) reaches a minimum, most of the size 10 graphs rise again, while the size 100 graphs remain relatively flat.

Table 1 describes characteristics of the best combined time ("Best") using selective optimization, i.e., the lowest point on the solid curves, for all benchmarks, using the time-based profile. The first column of Table 1 lists the benchmark name. The remaining columns present the results for the two categories of benchmarks, four columns for the short-running programs and four columns for the longer-running programs. Within each category the first three columns report the characteristics used to achieve the best combined time (Best) using selective optimization. This could be no methods optimized, all methods op-

Table 1. Effectiveness of selective optimization using time-based profiling for both the short and longer-running benchmarks. Recall that `pBOB` and `Volano` are not included in the short-running suite because they are designed to be server benchmarks.

	Short-running suite				Longer-running suite			
	Best Characteristics				Best Characteristics			
	Methods for Best		Pct of Profile	Best Combined	Methods for Best		Pct of Profile	Best Combined
Benchmark	Number	Pct	Covered	Time [s]	Number	Pct	Covered	Time [s]
201_compress	16	9%	99.9%	3.8	15	8%	99.9%	39.9
202_jess	7	1%	88.6%	1.7	40	8%	98.9%	31.3
209_db	4	2%	84.6%	1.5	9	5%	99.9%	70.9
213_javac	0	0%	0%	2.7	160	17%	92.9%	49.1
222_mpegaudio	50	14%	99.8%	4.8	100	29%	99.9%	30.0
227_mtrt	20	6%	86.3%	3.8	70	21%	99.2%	19.7
228_jack	3	1%	40.5%	5.8	13	3%	82.4%	44.3
opt-cmp	1	0%	11%	11.2	290	18%	95.6%	75.8
pBOB	—	—	—	—	200	33%	98.9%	54.8
Volano	—	—	—	—	13	2%	93.9%	38.6
Average	12.6	3.8%	63.8%	4.4	91.1	14.4%	96.1%	45.4

timized, or some methods optimized in order of hotness. The column marked "Number" reports the number of (hot) methods optimized. The column marked "Pct" gives the percentage of all methods in the application that are optimized. The column marked "Pct of Profile Covered" gives the percentage of the time-based profile that is covered by the selected methods. The following column gives the Best time in seconds.

The number of methods optimized for Best varies greatly across the benchmarks, ranging from 0 to 290 methods (0–33% of all methods), demonstrating that the benchmarks have rather different execution patterns regarding method hotness. For example, `201_compress` spends 99.9% of its execution time in just 9% of its methods. This is in contrast to the more object-oriented benchmarks, such as `213_javac`, `opt-cmp` and `pBOB`, which distribute their execution time more evenly among a much larger set of methods, and thereby presenting a bigger challenge for selective optimization. For the short-running benchmarks, Best includes 0 and 1 methods for `213_javac` and `opt-cmp`, respectively, covering only 0% and 11% of the time-based profile. For the longer-running benchmarks, Best selects the largest number of methods to optimize for the three object-oriented benchmarks. Yet despite the large number of methods optimized, these benchmarks are among the four smallest regarding percentage of profile covered.

The bar charts in Figs. 2 and 3 compare the combined time (compilation + execution) performance of three interesting method selection strategies: no methods optimized (none-opt), all methods optimized (all-opt), and Best. Recall

that Best, by definition, gives the best combined time, so it will always be better than or equal to the other two configurations. Below each benchmark name is the *execution time only* when all methods are optimized (corresponding to the right-most point on the dashed lines in Fig. 1), which we refer to as *all-opt-exe*. All bars are normalized to this value; thus, all three method selection strategies are shown relative to a system in which compile time is free and all methods run at peak (for this environment) performance.[3] Given Java's dynamic semantics, it does not appear that such a system, without any runtime compilation or interpretation, is possible.

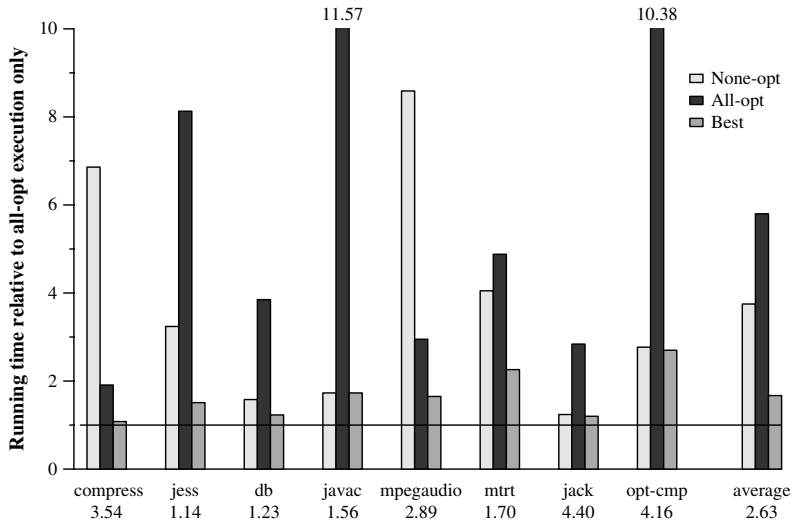


Fig. 2. Short-running suite performance comparison. The height of each bar is the combined time normalized to all-opt-exe time. The numbers under the benchmark names are the all-opt-exe time in seconds.

For the short-running benchmarks (Fig. 2), Best averages an additional 67% slower than all-opt-exe. The all-opt scheme is the worst of the three selections for 6 of the 8 benchmarks. This in not surprising because the benchmarks do not run long enough to recover the cost of optimizing all methods. For 5 of 8 benchmarks, the improvement of Best over the other two extremes is substantial. For all benchmarks it averages 56% better than none-opt and 71% better than all-opt.

For the longer-running benchmarks (Fig. 3), Best averages only an additional 13% in combined time compared to an all-opt-exe configuration. All-opt is much closer to Best for all cases, averaging 28% worse than all-opt-exe. Because of the

[3] An adaptive system can utilize feedback-directed optimizations, i.e., optimizations based on the application's current execution profile, to potentially obtain performance beyond that of all-opt-exe.

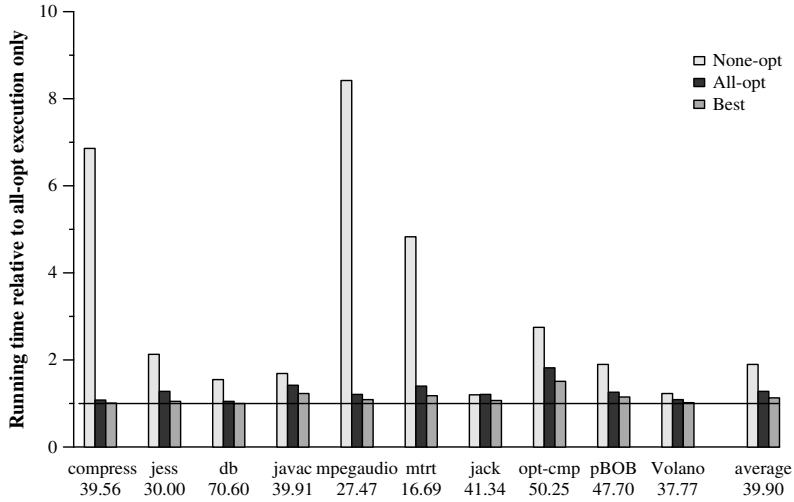


Fig. 3. Longer-running suite performance comparison. The height of each bar is the combined time normalized to all-opt-exe time. The numbers under the benchmark names are the all-opt-exe time in seconds.

benchmarks longer execution time, none-opt is the worst strategy in all cases. These results suggest that although an all-opt strategy is effective for longer-running applications the optimization overhead is not insignificant.

Sweet Spot The previous performance results for Best are based on a particular ordering of the methods using an offline time-based profile. However, one may wonder how closely a technique must estimate the set of actual hot methods that achieve Best to obtain reasonable performance. To address this question, we record the number of collected timings that are within N percent of the Best combined time. We refer to this number as being in the *sweet spot* for this benchmark. Table 2 provides this information for the following values of N: 1, 5, 10, 25, and 50. The value is expressed as the percentage of all collections within N% of the Best combined time. Due to space considerations we present for each sweet spot range the average and range of percentages over the benchmarks in each suite. For example, the value of 2.6% in the "1%" column states that on average, 2.6% of the collected timings of the short-running benchmarks fell within 1% of the Best time. The next row, "1–5%" states that 2.6% represents an average of values that ranged from 1–5%.

This table quantifies properties that are visually present in the graphs of Fig. 1 as well as those graphs not shown. For example, the graph of the short-running `213_javac` in Fig. 1 shows few points as low as the Best value at point (0, 2.39), i.e., the no methods optimized resulted in the best running time. In fact, for this benchmark only 1% of the x-axis data points are within 50% of the Best combined time. This is in contrast to the graph of the longer-running

Table 2. Percentage of x-axis data points that are within N% of Best combined time using a time-based profile for $N = 1, 5, 10, 25, 50$.

Benchmark	Metric	Sweet Spot Percentage				
		1%	5%	10%	25%	50%
Short running	Avg	2.6%	3.9%	6.2%	13.0%	23.0%
Short running	Range	1–5%	1–11%	1–20%	1–34%	1–50%
Longer running	Avg	7.1%	33.1%	62.2%	89.4%	92.5%
Longer runnnig	Range	1–16%	5–95%	10–95%	25–100%	50–100%

`222_mpegaudio`, where 90% of the selections are within 5% of Best performance. Although the longer-running benchmarks, in general, are more forgiving than the short-running benchmarks, their 5% sweet spots do vary considerable, 5%–95%.

4.2 Sample-Based Profile

As mentioned in Section 3.1, a sample-based profile was also used to examine the feasibility of selective optimization. The same experiments were repeated, except the sample-based profile was used to establish method hotness. Fig. 4 shows two example performance graphs using the sample-based profile. The curves start out similar to those from the time-based profile; however, the curves end much earlier because only a subset of the executing methods are sampled.

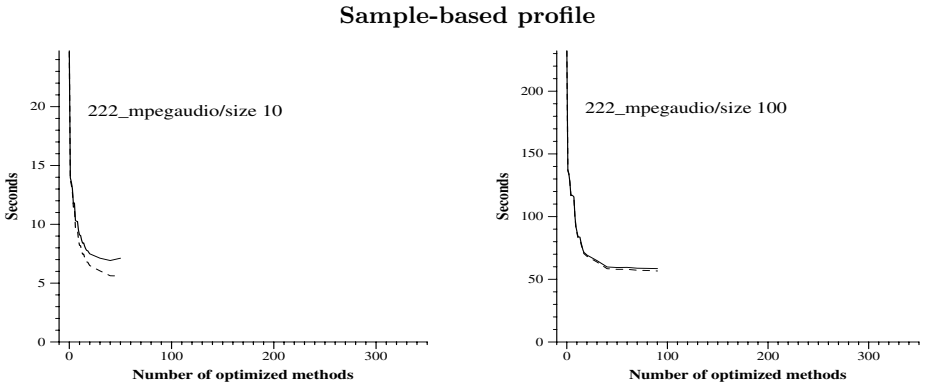


Fig. 4. Graphs for representative short-running and longer-running benchmark suites for the sample-based profiles

Table 3 shows characteristics of the sample-based profile for all benchmarks. The second column shows the percentage of all executed methods that are sampled, ranging from 6.7%–57.5% and averaging 24.6%. The remaining columns of

Table 3. Comparison of "Best" using the sample-based and time-based profile

	Benchmark	% Methods Sampled	Num Methods for Best			Sampled Best Time	Sampled Best / Time Best
			Time	Sample	Both		
Short-running suite	201_compress	8.5%	16	10	10	3.8	1.00
	202_jess	6.7%	7	4	4	1.7	1.00
	209_db	9.9%	4	8	3	1.6	1.02
	213_javac	13.6%	0	0	0	2.7	1.00
	222_mpegaudio	15.5%	50	40	28	4.8	1.01
	227_mtrt	26.0%	20	20	14	3.9	1.01
	228_jack	17.7%	3	5	4	5.7	0.99
	opt-cmp	29.7%	1	17	1	9.4	0.84
	Average	15.9%	12.6	15.5	8.0	4.2	0.98
Longer-running suite	201_compress	11.0%	15	11	10	39.9	1.00
	202_jess	15.3%	40	40	36	31.4	1.00
	209_db	8.8%	9	7	5	70.6	1.00
	213_javac	41.9%	160	160	122	49.3	1.01
	222_mpegaudio	27.2%	100	90	64	30.2	1.01
	227_mtrt	34.8%	70	60	50	19.2	0.97
	228_jack	33.4%	13	30	12	43.8	0.99
	opt-cmp	57.5%	290	220	203	75.2	0.98
	pBOB	46.2%	200	210	157	53.4	0.97
	Volano	13.6%	13	12	2	39.0	1.01
	Average	28.9%	91.0	88.0	67.9	45.2	0.99

Table 3 show characteristics about the Best from the sampled-profile (sampled-best), and compare it to Best from the time-based profile (time-best). The three columns marked "Num Methods for Best" give the number of methods that are optimized to achieve time-best and sample-best, respectively, and the number of methods optimized in both cases, i.e., the intersection of the methods of the two previous columns. For the short-running benchmarks, the methods selected for Best are similar for both profiles; however, for the longer-running benchmarks there is a fairly large variation, especially for `213_javac`, `opt-cmp`, and `pBOB`.

The last two columns of Table 3 show the running time of sample-best and the ratio of sample-best to time-best. Sample-best shows almost no degradation to time-best, and in several cases is better than time-best. As mentioned in Section 3.1, sample-best can be better than time-best because instrumentation perturbs the code. Thus, it is possible for the hotness ordering generated by the sampled profile to be more accurate than the time-based profile. Furthermore, because all calls and returns are instrumented, the error is likely to be greatest for benchmarks that contain a large number of short-lived method calls.[4]

[4] The methodology used for selecting methods to optimize does not consider characteristics of the methods such as compile time or the performance improvement

The most important observation to be made from this data is that the sample-based, incomplete profile achieves the same Best performance as the time-based profile *independent of the running time of the application.* The key is that the sample-based profile is accurate enough for the methods that matter. For the short-running applications there is not much time to optimize, so only the hottest few methods need to be detected by the sample-based profile. As the program runs longer and there is more time to optimize, the sample-based profile will have had enough time to reasonably order the needed number of methods.

5 Using Profiling as a Predictor for Method Selection

Section 4 determined the Best set of methods to optimize by observing the timings of many different executions. In an adaptive optimization system, this decision will be made by observing the profiling information and *predicting* which methods would be the best to optimize. This section investigates the effectiveness of some simple prediction strategies to determine if there exists a single strategy, for all benchmarks, that can approach the results of Best. The following strategies are considered:

Num methods: Optimize the N hottest methods,
Percent of methods: Optimize N% of all profiled application methods (in hotness order),
Time/Samples in method: Optimize any method that is executed for more than N milliseconds or N samples,
Time/Samples in method relative to hottest method: Optimize any method that is executed for more than N% of the time of the hottest method,
Percent profile coverage: Optimize enough hot methods (in hotness order) to cover N% of the profiled execution.

For each strategy a wide range of values for N were explored, with the goal of finding a value of N which performs well for all of the benchmarks of similar execution time. The value of N that resulted in the lowest performance degradation from Best, averaged over all benchmarks of similar size, is chosen.

Table 4 summarizes the results for the short-running and longer-running suites, respectively, for both types of profiles. The first column lists the profile used and benchmark suite. The last five columns list the strategies as described above. For each pair of profile type and benchmark suite, there are three rows. The first row gives the value of N that resulted in the lowest average performance degradation above Best. Given this value of N, the second and third rows show the average percentage degradation from (worse than) Best, and the range for all benchmarks in that suite, respectively. For example, for the short-running benchmarks with the time-based profile, optimizing the 9 hottest methods resulted in an average degradation of 31% from Best for those benchmarks, with values ranging from 0.1%–71.0%.

of optimization. Therefore, it is possible that either profiling technique could "get lucky" if it happens to bias small methods, or methods that optimize well.

Table 4. Summary of the effectiveness of predictor strategies

Profile Used/ Benchmark Suite	Metric		Num Methods	Pct of Methods	Time/ Samples in Method	Time/Samples in Method Relative to Hottest Method	Pct Profile Coverage
Time/ Short running	Optimal Values		9	5%	38810ms	4.7%	74.3%
	Pct Decrease	Avg	31.7	49.1	27.2	41.5	53.8
		Range	0.1–71.0	1.8–116	0.0–73.5	0–130	0.2–135
Time/ Longer running	Optimal Values		120	19%	253ms	0.05%	99.7%
	Pct Decrease	Avg	3.7	2.5	3.6	3.7	3.7
		Range	0.2–6.6	0.2–11.9	1.5–17.2	0.1–6.7	0.2–7.3
Sample/ Short running	Optimal Values		10	11%	10	3.25%	70.25%
	Pct Decrease	Avg	6.2	35.0	8.5	42.2	50.2
		Range	0.0–31	0.0–148	0.0–51.4	0.9–135	0.9–147
Sample/ Longer running	Optimal Values		180	50%	9	0.10%	99.2%
	Pct Decrease	Avg	2.5	1.2	1.0	2.7	2.3
		Range	0.0–7.7	0.0–2.5	0.0–2.2	0.0–7.4	0.1–9.7

The first observation is that Best is harder to predict for the short-running benchmarks (first and third group of rows) than for the longer-running benchmarks (second and fourth group of rows), as can be seen by comparing the rows labeled "Avg". The average degradations from Best for the short-running suite varies across the different prediction strategies from 27%–53% for the time-based profile and 6.2%–50.2% for the sample-based profile. For the longer-running benchmarks, the worst average degradation from both profiling techniques is 3.7%. This extreme difference between the short- and longer-running suites is most likely not because the location of Best is inherently more difficult to predict for short-running applications, but because the sweet spot of the longer-running applications is much more forgiving.

"Time/Samples in method" is the most effective overall strategy; its average degradation from Best is either the smallest, or close to the smallest, for all four profile/benchmark pairs. One explanation is that with short-running applications this metric limits the number of methods that are candidates for optimization because fewer methods exceed the specified threshold. This implicitly limits the compilation overhead in short-running applications, but allows a larger set of methods to be optimized in longer-running applications.

Interestingly, the sample-based strategies are better than the time-based strategies (having a smaller average degradation from Best) for all but one case. This phenomenon could be partly due to inaccuracies in the time-based profile, however it can also be more fundamental in that method selection is inherently less error prone with a sample-based profile. Because the profile does not contain

all executed methods, it automatically limits the selection to methods to those that are sampled at least once, essentially adding a secondary condition to the prediction strategy.[5]

In particular, "Samples in method" is the best overall strategy, performing very well for both the short- and longer-running benchmarks, and having its optimal values for both categories be very close (9 and 10). This is in contrast to all other short/longer strategy pairs, where the optimal values vary greatly. This result echoes the performance results obtained with an early version of the sampling-based Jalapeño adaptive system, which used an online version of "Samples in method" to guide optimization decisions.

6 Related Work

Radhakrishnan et al. [31] used the Kaffe Virtual Machine to measure the performance improvement possible by interpreting, rather than compiling, cold methods for a subset of the SPECjvm98 benchmarks with input size 1. Although the two studies touch on similar ideas, Radhakrishnan et al. focused on architectural issues for Java, while our study emphasized a comprehensive evaluation of selective optimization designed to aid the design and implementation of an adaptive system.

Kistler [25] presents a continuous program optimization architecture for Oberon that allows "up-to-the-minute" profile information to be used in program reoptimization. He concludes that continuous optimization can produce better code than offline compilation, even when past profiling information is used. Optimizations are evaluated by computing *ideal* performance speedup, which does not include profile or optimization time. This speedup is used to compute the *break-even* point: the length of time the application would have to execute to compensate for the time spent profiling and optimizing.

The HotSpot JVM [24], the IBM Java Just-in-Time compiler (version 3.0) [32], the Intel JUDO system [15], and the Jalapeño JVM [6] are adaptive systems for Java. The first three systems initially interpret an application and later compile performance-critical code. The IBM JIT and Intel JUDO system uses method invocation counters augmented to accommodate loops in methods to trigger compilation. JUDO optionally uses a thread-based technique in which a separate thread periodically inspects the counters and performs compilation. Jalapeño uses a compile-only approach based on the same low-overhead sampling that is used in this paper. It employs a cost/benefit model to trigger recompilation. Details of the HotSpot compilation system are not provided.

Hölzle and Unger [22] describe the SELF-93 system, an adaptive optimization system for SELF. Method invocation counters with an exponential decay mechanism are used to determine candidates for optimization. Unlike the results of our study, they conclude that determining when to optimize a method is more

[5] The sampled profile does not add a secondary condition to "Time/Samples in method" because it already limits the selection in exactly the same manner.

important than determining what to optimize. We attribute this contrasting conclusion to the differing goals of the two systems. The goal of their system is to avoid long pauses in interactive applications, where as we focus on achieving optimal performance for short- and longer-running noninteractive applications.

Detlefs and Agesen present two studies of selective optimization using the SPECjvm98 benchmarks, evaluating both offline [18] and online [2] strategies for selecting methods to optimize. Both studies utilize three possible modes of execution: an interpreter, a fast JIT, and an expensive optimizing compiler. Their studies confirm the performance benefits of selective optimization, and offer evidence that using all three modes is an effective approach.

Plezbert and Cytron [29] propose *continuous compilation*, as well as several other selective optimization strategies, and simulate their effectiveness on C programs.

Bala et al. [8] describe Dynamo, a transparent dynamic optimizer that performs optimizations on a native binary at runtime. Dynamo initially interprets the program, keeping counters to identify sections of code called hot *traces*. These sections are then optimized and written as an executable. The authors report an average speedup of 9%.

Burger and Dybvig [10, 11] explore profile-driven dynamic-recompilation in Scheme, with an implementation of a basic block reordering optimization. They report a 6% reduction in runtime and an average recompile time of 12% of their base compile time.

Hansen [21] describes an adaptive FORTRAN system that makes automatic optimization decisions. When a basic block counter reaches a threshold, the basic block is reoptimized and moved to the next *optimization state*, where more aggressive optimizations are performed.

Another area of research considers performing runtime optimizations that exploit invariant runtime values [7, 20, 28, 27, 16, 30]. The main disadvantage of these techniques is that they rely on programmer directives to identify the regions of code to be optimized. There are also nonadaptive systems that perform compilation/optimization at runtime to avoid the cost of interpretation. This includes early work such as the Smalltalk-80 [19] and Self-91 [13] systems, as well many of today's JIT Java compilers [1, 26, 36].

There are also fully automated profiling systems that use transparent profiling to improve performance of future executions. Such systems include Digital FX!32 [23], *Morph* [37], and *DCPI* [5]. However, these systems do not perform compilation/optimization at runtime.

7 Conclusions

This paper describes an empirical study of selective optimization using the Jalapeño JVM. Two categories of benchmarks, short running (1.1–4.8 seconds) and longer running (16.6–70.6 seconds), are studied using SPECjvm98, pBOB, Volano, and the Jalapeño optimizing compiler as benchmarks. The main observations are

- The Best selective optimization strategy (counting compilation time) is within 67% of all-opt-exe for the short-running benchmarks and 14% for the longer-running benchmarks.
- Best substantially outperformed both none-opt and all-opt for the short-running benchmark suite, averaging 56% and 71% improvement, respectively.
- For the longer-running benchmarks, Best shows a much smaller improvement over all-opt, averaging 12%.
- The sweet spot is fairly small for short-running benchmarks, and large for longer-running benchmarks.
- Best from the incomplete, sample-based profile can match Best from the time-based profile independent of the application's running time and sweet spot size, even though the sample-based profile contains only a fraction of the executed methods.
- Predicting which methods to optimize is much harder for short-running applications than for longer-running applications. Predictions based on the sample-based profile are more effective than predictions based on the time-based profile. "Samples in method" is the most effective overall predictor, averaging 8.5% and 1.03% degradation from Best, respectively, for the short- and longer-running applications.

7.1 Impact on Adaptive Optimization

The goal of this work is to gain a better understanding of the issues surrounding selective optimization, and in doing so provide insight into the more general problem of adaptive optimization. Understanding the performance gains possible from selective optimization, as well as the classes of applications for which these gains can be expected, is useful for guiding efforts in designing an effective adaptive system.

As expected, selective optimization provides the largest potential benefit for the short-running benchmarks. The goal of an adaptive system will be to achieve as much of this win as possible.

The longer-running applications presented in this paper are approaching the point where all-opt is only slightly worse than Best, which, in turn, is only slightly worse than all-opt-exe. In addition, the large sweet spot of the longer-running benchmarks makes it easy to predict a set of methods to optimize that will result in reasonable performance. We feel this is encouraging for adaptive optimization because it suggests that with little effort an adaptive system can achieve close to the performance of all-opt-exe, and by concentrating its efforts on feedback-directed optimizations, such as inlining and specialization, can achieve performance beyond that of a static compiler.

The results of the sample-based profile are also encouraging. We now do not expect the accuracy of our coarse-grained sampler to be a problem for the adaptive system. The sample-based profile was more effective than the time-based profile when used to predict which methods to optimize, even for the short-running benchmarks, which run for as little as 1.5 seconds. This suggests that an adaptive system will be able to use an online, sample-based profile to

make effective optimization decisions, and will be able to trust the profile after a short amount of time.

One concern is how an adaptive system should select methods for optimization. Even with the offline profile providing full knowledge of the future, the Best set of methods to optimize is difficult to predict for the short-running benchmarks. In an adaptive system, the problem becomes even more complex because less profiling information is available. Yet, unlike in this study, an adaptive optimization system is not required to make all of its optimization decisions at one time, and thus the predictors presented in Section 5 are not the only selection options available to an adaptive system. For example, an adaptive system could begin by optimizing only a small part of the application and continue until it is no longer improving performance. We see exploring such possibilities as an interesting area of future work.

Acknowledgments

We thank Vivek Sarkar for encouraging this work and the other Jalapeño team members [3] for their efforts in developing the system used to conduct this study. Stephen Fink, David Grove, Atanas Rountev, Peter Sweeney, Laureen Treacy, and the LCPC program committee provided useful suggestions and feedback on an earlier draft of this work. We also thank Michael Burke for his support.

References

[1] Ali-Reza Adl-Tabatabai, Michał Cierniak, Cuei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, effective code generation in a Just-in-Time Java compiler. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 280–290, Montreal, Canada, 17–19 June 1998. *SIGPLAN Notices* 33(5), May 1998.

[2] Ole Agesen and David Detlefs. Mixed-mode bytecode execution. Technical Report TR-2000-87, Sun Microsystems, June 2000.

[3] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1), 2000.

[4] Bowen Alpern, Dick Attanasio, John J. Barton, Anthony Cocchi, Derek Lieber, Stephen Smith, and Ton Ngo. Implementing Jalapeño in Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 314–324, 1999.

[5] Jennifer M. Andersen, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: Where have all the cycles gone? Technical Note 1997-016a, Digital Systems Research Center, www.research.digital.com/SRC, September 1997.

[6] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 2000.

[7] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. Fast, effective dynamic compilation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 149–159, Philadelphia, Pennsylvania, 21–24 May 1996. *SIGPLAN Notices* 31(5), May 1996.

[8] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.

[9] S. Baylor, M. Devarakonda, S. Fink, E. Gluzberg, M. Kalantar, P. Muttineni, E. Barsness, R. Arora, R. Dimpsey, and S. Munroe. Java server benchmarks. *IBM Systems Journal*, 39(1), 2000.

[10] Robert G. Burger. *Efficient Compilation and Profile-Driven Dynamic Recompilation in Scheme*. PhD thesis, Indiana University, 1997.

[11] Robert G. Burger and R. Kent Dybvig. An infrastructure for profile-driven dynamic recompilation. In *ICCL'98, the IEEE Computer Society International Conference on Computer Languages*, May 1998.

[12] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño dynamic optimizing compiler for Java. In *ACM 1999 Java Grande Conference*, pages 129–141, June 1999.

[13] Craig Chambers and David Ungar. Making pure object-oriented languages practical. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–15, November 1991. *SIGPLAN Notices* 26(11).

[14] Jong-Deok Choi, David Grove, Michael Hind, and Vivek Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 21–31, September 1999.

[15] Michal Cierniak, Guei-Yuan Lueh, and James M. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. In *SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.

[16] Charles Consel and François Noël. A general approach for run-time specialization and its application to C. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 145–156, January 1996.

[17] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98, 1998.

[18] David Detlefs and Ole Agesen. The case for multiple compilers. In *OOPSLA'99 Workshop on Peformance Portability, and Simplicity in Virtual Machine Design*, November 1999. Denver, CO.

[19] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *11th Annual ACM Symposium on the Principles of Programming Languages*, pages 297–302, January 1984.

[20] Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan J. Eggers. An evaluation of staged run-time optimizations in DyC. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 293–304, 1999.

[21] Gilbert J. Hansen. *Adaptive Systems for the Dynamic Run-Time Optimization of Programs*. PhD thesis, Carnegie-Mellon University, 1974.

[22] Urs Hölzle and David Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Transactions on Programming Languages and Systems*, 18(4):355–400, July 1996.

[23] Raymond J. Hookway and Mark A. Herdeg. Digital FX!32: Combining emulation and binary transslation. *Digital Technical Journal*, 9(1):3–12, January 1997.

[24] The Java Hotspot performance engine architecture. White paper available at http://java.sun.com/products/hotspot/whitepaper.html, April 1999.

[25] Thomas P. Kistler. *Continuous Program Optimization*. PhD thesis, University of California, Irvine, 1999.

[26] Andreas Krall. Efficient JavaVM Just-in-Time compilation. In Jean-Luc Gaudiot, editor, *International Conference on Parallel Architectures and Compilation Techniques*, pages 205–212, October 1998.

[27] Mark Leone and Peter Lee. Dynamic specialization in the Fabius system. *ACM Computing Surveys*, 30(3es):1–5, September 1998. Article 23.

[28] Renaud Marlet, Charles Consel, and Philippe Boinot. Efficient incremental runtime specialization for free. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 281–292, 1999.

[29] Michael P. Plezbert and Ron K. Cytron. Does "just in time" = "better late than never"? In *Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 120–131, January 1997.

[30] Massimiliano Poletto, Dawson R. Engler, and M. Frans Kaashoek. tcc: A system for fast, flexible, and high-level dynamic code generation. In *Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation (PLDI)*, pages 109–121, Las Vegas, Nevada, 15–18 June 1997. *SIGPLAN Notices* 32(5), May 1997.

[31] R. Radhakrishnan, N. Vijaykrishnan, L. K. John, and A. Sivasubramaniam. Architectural issues in Java runtime systems. In *Proceedings of the Sixth International Symposium on High Performance Computer Architecture (HPCA-6)*, pages 387–398, Toulouse, France, January 2000.

[32] Toshio Suganama, Takeshi Ogasawara, Mikio Takeuchi, Toshiaki Yasue, Motohiro Kawahito, Kazuaki Ishizaki, Hideaki Komatsu, and Toshio Nakatani. Overview of the IBM Java Just-in-Time compiler. *IBM Systems Journal*, 39(1), 2000.

[33] VolanoMark 2.1. `http://www.volano.com/benchmarks.html`.

[34] John Whaley. Dynamic optimization through the use of automatic runtime specialization. M.eng., Massachussetts Institute of Technology, May 1999.

[35] P. Wu, S. P. Midkiff, J. E. Moreira, and M. Gupta. Efficient support for complex numbers in Java. In *ACM 1999 Java Grande Conference*, San Francisco, CA, June 1999.

[36] Byung-Sun Yang, Soo-Mook Moon, Seongbae Park, Junpyo Lee, SeungIl Lee, Jinpyo Park, Yoo C. Chung, Suhyun Kim, Kemal Ebcioglu, and Erik Altman. LaTTe: A Java VM Just-in-Time compiler with fast and efficient register allocation. In *International Conference on Parallel Architectures and Compilation Techniques*, October 1999.

[37] Xiaolan Zhang, Zheng Wang, Nicholas Gloy, J. Bradley Chen, and Michael D. Smith. System support for automated profiling and optimization. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, Operating Systems Review, 31(5), pages 15–26, October 5–8 1997.

MaJIC: A Matlab Just-In-time Compiler*

George Almasi and David A. Padua

Department of Computer Science
University of Illinois at Urbana-Champaign
{galmasi,padua}@cs.uiuc.edu

Abstract. This paper describes our experience with **MaJIC**, a just-in-time compiler for MATLAB. In the recent past, several compiler projects claimed large performance improvements when processing MATLAB code. Most of these projects are static compilers suited for batch processing; **MaJIC** is a just-in-time compiler. The compilation process is transparent to the user. This impacts the modus operandi of the compiler, resulting in a few interesting analysis techniques. Our experiments with **MaJIC** indicate large speedups when compared to the interpreter, and reasonable performance when compared to static compilers.

1 Introduction

MATLAB [14], a product of Mathworks Inc., is a popular programming language and development environment for numeric applications. MATLAB is similar to FORTRAN 90: it is an imperative language that deals with vectors and matrices. Unlike FORTRAN, however, MATLAB is untyped and polymorphic: the semantic meaning of identifiers is determined by their mode of use, and the operators' meaning depends on the operands.

The main strengths of MATLAB lie both in its interactive nature, which makes it a handy exploration tool, and the richness of its precompiled libraries and toolboxes, which can be combined to solve complex problems. The trouble with MATLAB is its speed of execution. Unless the bulk of the computation is shifted into one of many built-in libraries, MATLAB pays a heavy performance penalty: two or even three orders of magnitude compared to similar Fortran code.

The purpose of our work is to eliminate MATLAB's performance lag while maintaining its interactive nature. Hence the requirement for dynamic, or just-in-time, compilation, which is used for similar purposes in other programming environments, including Smalltalk and Java. **MaJIC** acts like an interpreter: there is no need to deal with Makefiles and a lengthy compilation process, and therefore the user's work flow is not interrupted.

Just-in-time compilation has been around for awhile, but has recently gained new notoriety with the rise of Java. Before Java it was called dynamic compilation. Deutsch described a dynamic compilation system for the Smalltalk language

* This work was supported in part by NSF contract ACI98-70687 and by a graduate fellowship from Intel.

S.P. Midkiff et al. (Eds.): LCPC 2000, LNCS 2017, pp. 68–81, 2001.

as early as 1984 [7]. Java benefits from JIT compilation by reduced machine instruction count per line of bytecode, optimized register usage, interprocedural analysis and, in many cases, replacement of virtual function calls with static calls. Both Java and MATLAB benefit from JIT compilation because it is possible to ascribe more precise semantic meaning to the symbols in the program, thereby reducing the potential polymorphism of the code - although the optimizations turn out to be quite different in practice.

The rest of this paper is organized as follows: we set the stage by describing related work, then we give a high-level overview of our compiler. We describe some of the performance enhancing features we implemented. Finally we present the results we have obtained so far and our conclusions and plans for the future.

2 Other Work with MATLAB

The techniques used in **MaJIC** are based on those developed for DeRose and Padua's FALCON compiler [6, 5], a MATLAB to Fortran 90 translator implemented at the University of Illinois.

MENHIR [3], developed by Francois Bodin at INRIA, is similar to FALCON: it generates code for MATLAB and exploits parallelism by using optimized runtime libraries. MENHIR's code generation is retargetable (it generates C or FORTRAN code). It also contains a type inference engine similar to FALCON's.

MATCOM [13] is a commercial MATLAB-compatible compiler family, developed by Mathtools Inc. This is a full development environment which incorporates a debugger, an editor, many optimizations, and even a JIT compiler. It is only commercially available.

Octave [8] is a freeware MATLAB implementation. Octave may not be very fast, but it is GNU freeware.

MATCH [1] is a MATLAB compiler targeted to heterogeneous architectures, such as DSP chips and FPGAs. It also uses type analysis and generates code for multiple large functional units.

MatMarks [15] and MultiMatlab [17] are projects that are somewhat similar in that they extend MATLAB to deal with a multiprocessing environment. MultiMatlab adds MPI primitives to MATLAB; MatMarks layers MATLAB on top of the Treadmarks software DSM system.

Vijay Menon, at Cornell University, is currently developing a MATLAB vectorizer [16], a tool that transforms loops into Fortran 90-like MATLAB expressions, built on top of the **MaJIC** compiler framework. We are hoping to integrate his work into the compiler soon.

3 MaJIC Software Architecture

In this section we give a short overview of **MaJIC**'s architecture and mode of operation. **MaJIC**'s interaction with the rest of the world is handled by an *interpreter* similar to MATLAB's. Since MATLAB's code is proprietary we

couldn't expand it with the JIT compiler. Therefore we built our own interpreter which is capable of handling a subset of the MATLAB language. The interpreter scans the input file, parses it into a high-level Abstract Syntax Tree, and then executes the program by traversing the tree in program order. It also contains a symbol table, an operand stack, and a function finder.

The *analyzer* performs three operations: symbol disambiguation, type inference, and optional loop dependence analysis. Analysis does not change the program tree, but rather annotates it. It is designed to be invoked at runtime; its input is in effect the current execution state of the interpreter, including the current symbol table and the operand stack.

The *code generator* generates *executable* code directly based on the AST and its annotations, and does so very fast. The code generator allocates registers, emits machine code, resolves addresses and patches forward jumps.

The *code repository* maintains order between the several versions of annotations and compiled code co-existing at a given time. Since a given piece of MATLAB code might be invoked several times, and since analysis depends heavily on the state of the interpreter at the time of invocation, there could be several sets of type annotations for the same code at the same time.

4 MaJIC Compilation Techniques

In this section we describe some of the techniques we used to speed up MATLAB execution.

4.1 Symbol Disambiguation

MATLAB symbols exclusive of keywords represent one of three things: variables, calls to built-in primitives, or calls to user functions. In the interpreter the decision is made at runtime when the symbol's AST node is executed. We call a symbol occurrence *ambiguous* if the way the program is written allows multiple meanings for it. The code box below shows a loop where the first occurrence of the symbol `i` is ambiguous, interpreted as $\sqrt{-1}$ in the first iteration and as a variable afterwards.

```
while(...),
        z = i;
        i = 1;
end
```

In most MATLAB programs, however, symbol ambiguity can be removed at compile time. In particular, symbols that represent variables can be conservatively identified by a variation of reaching definitions analysis: A symbol that has a reaching definition on *all* paths leading to it must be a variable.

The transfer function in this analysis can be set up as usual, but with a slight twist. It is defined on the set of the symbols in the program $\mathcal{S}$ and the nodes of the CFG $\mathcal{N}$ as follows:

$$f : \mathcal{S} \times \mathcal{N} \rightarrow \mathcal{S}$$

$$f(s, \texttt{stmt}) = \begin{cases} s \cup \{a\} & if\ \texttt{stmt}\ has\ the\ form\ \boxed{\texttt{a = ...}} \\ s \cup \{a\} & if\ \texttt{stmt}\ has\ the\ form\ \boxed{\texttt{a(...) = ...}} \\ \phi & if\ \texttt{stmt}\ has\ the\ form\ \boxed{\texttt{... = a...}}\ and\ \{a\} \cap s = \phi \\ s & otherwise \end{cases} \tag{1}$$

Equation 1 says, in effect, that a symbol generates a reaching definition if it appears on the left-hand side. The unusual part of this equation is its third line, which says that any symbol that cannot be proven to be a variable has the side effect of killing all reaching definitions. This is necessary because of the existence of built-in functions like `clear`, which destroys the whole symbol table.

Use-def Chain Generation: the UD chain is obtained as a side-effect of the disambiguation step. It uses the results of the same reaching definitions analysis but determines the set of definitions for each variable by calculating the set union of its reaching definitions.

4.2 Type Inference

Our representation of a MATLAB expression type lies at the base of the inference system. This representation is very close to the one described in [6]. We use the following type descriptors:

- The *intrinsic type* of the expression, such as: `real`, `integer`, `boolean`, `complex` or `string`.
- The *shape* of the expression, i.e. the maximum number of rows and columns it can have.
- The *range* of real values the expression can take [2]. We do not define the range for strings and for complex expressions.

The type analyzer is a monotonic data analysis framework [18], which requires the analyzed property to form a finite semi-lattice and all transition functions to be monotonic.

Our lattice is the Cartesian product $L_i \times L_s \times L_l$ of the three component lattices, which are defined as follows:

$L_i = \{\mathcal{I}, \bot_i, \top_i, \leq_i, \vee_i\}$, where
$\mathcal{I} = \{\bot_i, bool, int, real, cplx, strg, \top_i\}$
$\bot_i \leq_i bool \leq_i int \leq_i real \leq_i cplx \leq_i \top_i$ and $\bot_i \leq_i strg \leq_i \top_i$
$L_s = \{\mathbb{N} \times \mathbb{N}, \bot_s, \top_s, \leq_s, \vee_s\}$, where $\mathbb{N}$ is the set of natural numbers, and
$\bot_s = \{0, 0\}, \top_s = \{\infty, \infty\}$; $\{a, b\} \leq_s \{c, d\}$ iff $a \leq c$ and $b \leq d$
$L_l = \{\mathbb{R} \times \mathbb{R}, \bot_l, \top_l, \leq_l, \vee_l\}$, where $\mathbb{R}$ is the set of real numbers, and
$\bot_l = \{nan, nan\}$; $\top_l = \{-\infty, \infty\}$;
$\{a, b\} \leq_l \{c, d\}$ iff $\{a, b\} = \bot_l$ or $(c \leq a$ and $b \leq d)$

A *type calculator* implements the transition functions for all the known AST types. A fairly large number of rules are required due to the many built-in MATLAB functions.

Convergence Issues. The monotonic data analysis framework is guaranteed to converge if the lattices involved are finite. Our lattices are finite (due to the fact that real numbers are represented in finite precision), but very large and, therefore, analysis can take a very long time. A simple example is presented below. The range of x keeps expanding in steps of 1.0, and it takes an impractically large number of iterations to reach the stationary value of $+\infty$.

```
while(...),
  x = x + 1;
end
```

A similarly simple example can be made up with a shape that grows very slowly.

```
while(...),
  x = [x x];
end
```

"Runaway" types keep changing as the iterations progress, and hence are easily detectable. In these cases **MaJIC** sets the types in question to $\top$ to force convergence.

4.3 Just-in-Time Analysis

In **MaJIC** analysis is performed very late, just before the code is executed. This late timing benefits the analyzer by contributing items of information to which a static analyzer doesn't have access.

- We can assume that the files in the MATLAB search path, and hence the names of the defined user functions, are not going to change between compilation and execution. This allows for greater precision in the symbol disambiguation step. There is an underlying assumption that the MATLAB code is not self-modifying.

– The analyzer has access to the entire interpreter state at the point where the compiler is invoked, including knowledge of the exact types and values of all defined variables. This translates into knowledge of all defined types at the point of entry into the compiled code. The extra information reduces the requirement for heavy analysis because very simple techniques yield good results. For example, there is no symbolic analysis in **MaJIC**.

Constant propagation is achieved as a side-effect of range propagation. A value is constant if its range contains a single number. Because the input information is so precise, many scalar values that would be symbolic parameters in static analysis end up as constants.

Exact shapes are inferred in **MaJIC** by adding another lattice to the type inference engine. Normally, shape inference delivers only upper bounds on the shape of MATLAB arrays; we append a dual lattice L_t to calculate lower bounds too. An exact shape results where the lower and upper bounds are determined to be equal.

Induction variables' ranges can also be used to infer exact shapes. Given an array index expression like `A(x)`, the range of `x` is used to determine the shape of A. Normally, this is an upper bound since there is no guarantee that `x` actually reaches the limits of its range. An array index can only be used to determine a lower bound if it assumes its range limits at least once. In a loop without premature exit instructions, the induction variable is guaranteed to hit its range limits. This "boundary-hitting" property can be propagated through index expressions that combine induction variables and constants to determine lower bounds for the shapes of arrays.

There are a number of ways in which exact shapes can be used to achieve better performance. For example, array bound checks can be removed wherever the array index expression's upper range limit is less than the lower bound of the corresponding array shape. On the right-hand side, this removes the requirement for mandatory error testing; on the left-hand side, it removes the auto-resizing check.

MATLAB's own companion compiler, `mcc`, has command-line switches to disable subscript checks, but using this option does not guarantee that the compiled program will work correctly. **MaJIC** removes subscript checks conservatively. The extra effort for just-in-time analysis is very low, comparing favorably with more conventional techniques such as those described in [10].

4.4 Code Generation

The design of the code generator is based on the trade-off between compilation speed and the quality of generated code, as well as a desire to be able to plug in several native code generators. We chose a two-layered approach: a high-level code selector traverses the AST and chooses the appropriate native code generating routines, which then emit native instructions.

The Code Selector traverses the AST and reads the annotations associated with each node. It performs a top-down pattern matching to select the appropriate implementation for any given AST node. For example, for a multiplication node there are a number of possible implementations, including scalar multiplication, dot product, vector-matrix multiplication and complex vector multiplication, to mention just a few.

Here is a short summary of the non-trivial techniques used by the code selector.

- Preallocation of temporary arrays, when their maximal sizes are known at compile time. Because of the Fortran 90-like expression semantics, complicated MATLAB vector expressions involve temporary matrices at each step, and preallocating these eliminates many calls to `malloc()`. For example, in an expression like $a \times b \times c$, at least one temporary matrix is needed to hold the result of $a \times b$. If upper bounds are known for the shapes of a and b, this temporary can be pre-allocated.
- Elimination of temporaries. Techniques have been developed to deal with this in Fortran 90-like languages [12]. **MaJIC** takes a simple way out and performs code selection to combine several AST nodes into a single library call. For example, expressions like `a*x+b'*c` may be transformed into a single call to `dgemv`.
- When indexed with out-of-bound subscripts, MATLAB arrays attempt to resize themselves to include the offending subscript. Thus, in some loops, arrays are resized in each iteration with the associated performance penalty. We implemented a simple, but effective, strategy of allocating about 10% more memory than needed whenever an array is resized dynamically. Since dynamic resize tends to happen in loops, the surplus of memory will likely be used up later, meanwhile reducing the number of calls to `malloc()`. Speedups from this technique can reach two orders of magnitude or more, depending on the characteristics of the loop.
- Complete unrolling of vector operations when exact array shapes are known. This technique is most effective on small (2×1 to 3×3) matrices and vectors because it eliminates all loop control instructions. In the future, we are planning to expand this technique to completely unroll small MATLAB loops with exact loop bounds.

The Native Code Generator emits code into a string buffer, and then the buffer is simply executed. Native code generation is also a multi-pass process.

Unfortunately there is no really general-purpose dynamic code generator that suits our purposes. The closest contender is Dawson Engler's `vcode` [9], a dynamic assembler with its own RISC-like instruction set. `vcode` has been ported to many popular CPU architectures including sparc and mips, but the x86 port is useless because it doesn't include the floating point instructions.

`tcc` [19] is a C-like language layered on top of `vcode`. It adds a lot of syntactic sugar and low-level optimizations like peephole optimization and register allocation, all performed at runtime. We used `tcc` for our implementation.

The second choice for the **MaJIC** backend is Paolo Bonzini's `GNU Lightning` [4], a dynamic assembler with a platform-independent instruction set. It is ported to the Sparc, PPC and IA32 architectures. `GNU Lightning` is based on Ian Piumarta's `ccg` dynamic assembler. The problem with `GNU Lightning` is that in striving to be architecture independent it restricts the user to 6 integer registers and a stack-based floating point system. These restrictions degrade performance on RISC architectures.

4.5 Repository Management

Since MATLAB is inherently polymorphic, the same source code can have several analyses attached to it, depending on how it is called. The repository keeps these analyses in order and tries to match existing compiled code to any call that is made. The concept of a match is defined very simply, based on the input data used by the analyzer: a given interpreter state matches an analysis stored in the repository if all the affected symbols have a "smaller" ($\leq$) type than the ones used by the analysis. For example a function compiled for complex arguments can be reused when invoked with real actual arguments.

If a function is invoked many times with incompatible arguments, each invocation might require a new analysis, thereby diminishing performance. To increase the change of future matches, **MaJIC**'s analyzer has the option of generalizing the input types before starting processing. This, of course, also results in a performance penalty for the compiled code.

MaJIC uses a gradual strategy for changing types. The first analysis is performed with the types as they are; the second analysis discards range information and exact shapes. If a third call is performed and none of the performed analyses match, all inputs are treated as matrices; finally, if yet another analysis is needed, it is performed with no information whatsoever from the input.

In real life, most MATLAB invocations are monomorphic; none of our benchmark functions need more than two analyses.

5 Experimental Results

To gauge the performance of the **MaJIC** compiler, we measured both the speed of the generated code and the time spent compiling code. Our frame of reference for execution speed is set by the MATLAB interpreter. We measured the performance of three compilers: MATLAB's own compiler `mcc`, the FALCON compiler, and **MaJIC**.

Surprisingly, for FALCON we obtained speedup figures that were better than those reported in [6]. We attribute this to the better Fortran 90 compiler we used as well as to a CPU with a large amount L2 cache that rewards the inherently better locality of compiled code.

5.1 The Benchmarks

We ran our experiments on the benchmarks originally gathered by Luiz DeRose. These are described in [6], so we will limit ourselves to a short description. The 12 benchmarks are all numerically intensive and relatively short: between 25 and 125 lines of code. They are as follows:

- `cgopt`, `icn`, `qmr`, and `sor` are linear equation solvers.
- `crnich`, `dirich`, `finedif`, `galrkn`, `orbec`, and `orbrk` are PDE solvers: finite difference solvers, a FEM solver, and n-body solvers.
- `adapt` is an adaptive quadrature algorithm.
- `mei` is a fractal generator.

The matrices in the benchmarks have sizes between 50×50 and 450×450. The problem sizes were originally determined to yield approximately 1 minute run times, but newer computers tend to finish them faster. We ran all our experiments on a Sun workstation with Solaris 2.7 and a 270MHz Ultrasparc processor.

5.2 Performance Figures

Figure 1 shows the speedups of `mcc`, **MaJIC**, and FALCON on a logarithmic scale. These speedups are based on execution times only (i.e., they don't take into account the compilation times). Missing bars mean speedups are below 1.0.

Execution times tell only half the story, however. Figure 2 shows the time spent from one user prompt to the next in each benchmark (i.e. the time spent by the user waiting for the result). For **MaJIC** and `mcc`, we assume that the compilation time is included in the prompt-to-prompt time; `mcc` tends to be used a lot from the MATLAB prompt, and **MaJIC** incurs the compilation penalty by default. We show two sets of speedups for FALCON, with and without compilation time included.

As the figures show, **MaJIC**'s performance falls between `mcc` and FALCON when judged on execution time alone. However, **MaJIC** takes much less time to compile than `mcc`, making it a better choice when the code is edited - and thus recompiled - frequently.

We can classify the benchmarks based on which compilation techniques influence their performance the most.

- The `adapt` benchmark benefits from the dynamic over-allocation scheme mentioned in section 4.4.
- The `cgopt` and `qmr` benchmarks benefit from a better implementation of the `dgemv` routine, as they spend more than 99% of their time doing matrix-vector multiplication. `qmr` derives additional benefits from an exact match between the MATLAB code and the API of the `dgemv` function. These benchmarks do not benefit much from type analysis; we get similar performance numbers with type analysis turned off.
- The `crnich` and `orbrk` functions suffer because **MaJIC** does not currently implement function inlining.

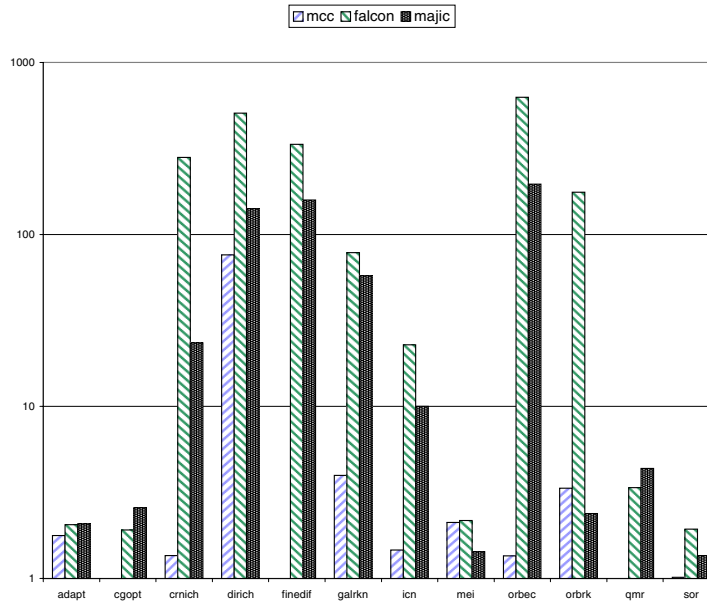


Fig. 1. speedups based on execution time only

– `crnich`, `dirich`, `finedif`, `galrkn`, and `orbec` suffer from the lack of low-level compiler optimizations in the **MaJIC** code generator (e.g., common subexpression elimination and strength reduction). We recompiled the FALCON codes with the Fortran compiler optimizations turned off (`f90 -g`). Figure 3 compares the unoptimized FALCON codes with **MaJIC** speedups. The **MaJIC** numbers include compile time. The performance numbers are pretty close to each other. Just-in-time analysis and high level code generation techniques in **MaJIC** compensate for the time lost while compiling.

We measured the effectiveness of just-in-time analysis by running the analyzer on a subset of our benchmarks, in just-in-time mode and in simulated static mode. In just-in-time, mode the analyzer had access to the interpreter state just before starting compilation; in static mode, the analyzer was cut off from the interpreter and ran with no outside information. Of 500 AST expression nodes, the just-in-time analyzer determined 402 to be scalars; by comparison, static analysis determined only 240 of them to be scalar.

Just-in-time analysis is reasonably fast. Analysis involves 2 to 20 traversals of the abstract syntax tree. The 270MHz Sparc processor that ran the benchmarks took about 0.2 milliseconds per iteration to process an AST node; average compilation time (including disambiguation, type inference, and code generation) is about 170 milliseconds.

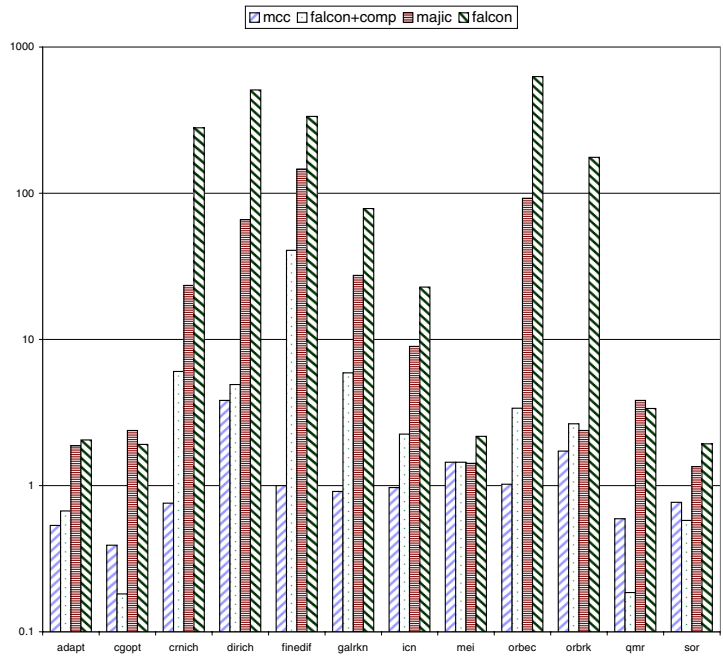


Fig. 2. speedups based on execution + compile time

Figure 4 shows the relative composition of execution time in **MaJIC**. Disambiguation and type inference never take more than 15% of the total time. Actual execution of the code dominates the time spent between user prompts. Code generation can take up to 40% of the time in the case of benchmarks that have large speedups.

6 Conclusion and Future Plans

We have built and tested a just-in-time compiler for MATLAB. It provides adequate performance while reducing the compilation overhead by at least an order of magnitude.

There is still a lot of room for improvement. The quality of the compiled code is mediocre at this point, due to the fact that the dynamic compiler/assembler combinations we use for code generation are not doing any low-level optimizations. Experiments show that by performing common subexpression elimination on the dynamically generated code we could improve performance by at least anoher 50some of the benchmarks; we are planning to implement these next.

Another technique we are planning to implement is to overlap compilation with computation. One of the ways to do this is to perform hybrid compilation,

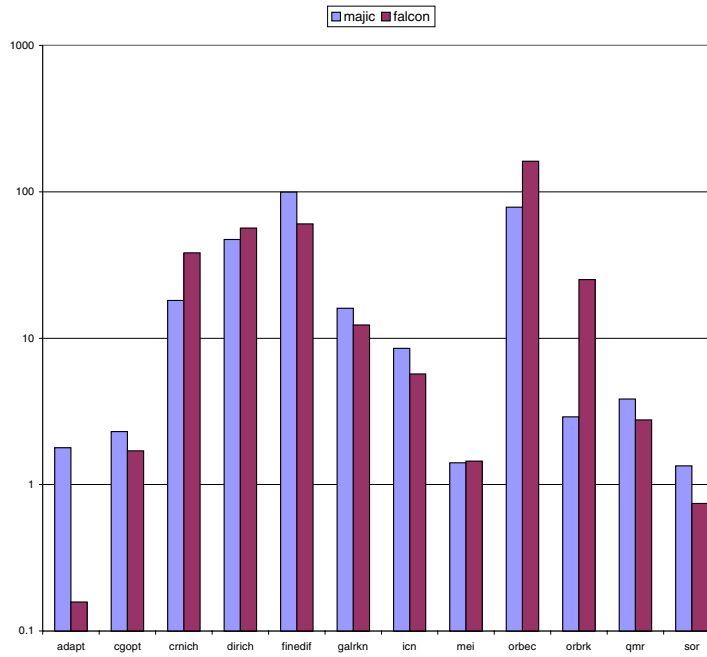


Fig. 3. Speedups without F90 optimizations

i.e. analyze and compile as much of the code in advance as soon as it is available, and to generate semi-compiled templates that can be quickly refined by the JIT compiler.

We also have other techniques under consideration. Some of the MATLAB libraries can be replaced by better ones [20, 11]; some of the loops can be executed in parallel on SMP machines. FALCON performance figures show that function inlining is a very cost-effective technique.

References

[1] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Chang, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, and M. Walkden. Match: A matlab compiler for configurable computing systems. Technical Report CPDC-TR-9908-013, Center for Parallel and Distributed Computing, Northwestern University, Aug. 1999.

[2] W. Blume and R. Eigenmann. Symbolic range propagation. In *Proceedings of the 9th International Parallel Processing Symposium*, April 1995.

[3] F. Bodin. MENHIR: High performance code generation for MATLAB. `http://www.irisa.fr/caps/PEOPLE/Francois/`.

[4] P. Bonzini. The GNU Smalltalk Homepage.

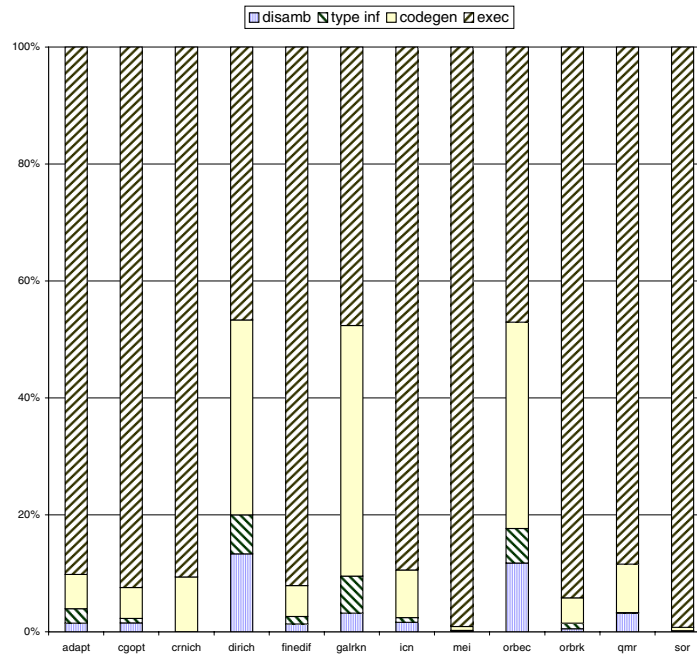


Fig. 4. Composition of execution time in **MaJIC**

[5] L. DeRose and D. Padua. Techniques for the translation of matlab programs into fortran 90. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(2):285–322, March 1999.
[6] L. A. DeRose. Compiler Techniques for MATLAB Programs. Technical Report UIUCDCS-R-96-1996, Department of Computer Science, University of Illinois, 1996.
[7] L. P. Deutsch and A. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th Symposium on the Principles of Programming Languages*, Salt Lake City, UT, 1984.
[8] J. W. Eaton. Octave homepage. `http://bevo.che.wisc.edu/octave/`.
[9] D. R. Engler. VCODE: a portable, very fast dynamic code generation system. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI '96)*, Philadelphia PA, May 1996.
[10] R. Gupta. Optimizing array bounds checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 2(1-4):135–150, 1993.
[11] F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–753, November 1997.
[12] G.-H. Hwang and J. K. Lee. An array operation synthesis scheme to optimize fortran 90 programs. In *PPOPP '95*, pages 112–122, Santa Clara, USA, 1995.
[13] Mathtools inc homepage. `www.mathtools.com`.
[14] Mathworks Inc. homepage. `www.mathworks.com`.

[15] Matmarks homepage. `http://polaris.cs.uiuc.edu/matmarks/matmarks.html`.
[16] V. Menon and K. Pingali. High-level semantic optimization of numerical codes. In *1999 ACM Conference on Supercomputing*. ACM SIGARCH, June 1999.
[17] V. Menon and A. E. Trefethen. MultiMATLAB: Integrating MATLAB with high-performance parallel computing. In *Proceedings of Supercomputing '97*, 1997.
[18] S. S. Muchnick and N. D. Jones. *Program Flow Analysis: Theory and Application*. Prentice Hall, 1981.
[19] M. Poletto, D. R. Engler, and M. F. Kaashoek. tcc: A system for fast, flexible, and high-level dynamic code generation. In *ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)*, pages 109–121, Las Vegas, Nevada, May 1997.
[20] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Supercomputing '98*, 1998.

SmartApps: An Application Centric Approach to High Performance Computing*

Lawrence Rauchwerger[1], Nancy M. Amato[1], and Josep Torrellas[2]

[1] Department of Computer Science, Texas A&M University
{rwerger,amato}@cs.tamu.edu
[2] Department of Computer Science, University of Illinois
torrella@cs.uiuc.edu

Abstract. State-of-the-art run-time systems are a poor match to diverse, dynamic distributed applications because they are designed to provide support to a wide variety of applications, without much customization to individual specific requirements. Little or no guiding information flows directly from the application to the run-time system to allow the latter to fully tailor its services to the application. As a result, the performance is disappointing. To address this problem, we propose application-centric computing, or SMART APPLICATIONS. In the executable of smart applications, the compiler embeds most run-time system services, and a performance-optimizing feedback loop that monitors the application's performance and adaptively reconfigures the application and the OS/hardware platform. At run-time, after incorporating the code's input and the system's resources and state, the SmartApp performs a global optimization. This optimization is *instance* specific and thus much more tractable than a global generic optimization between application, OS and hardware. The resulting code and resource customization should lead to major speedups. In this paper, we first describe the overall architecture of Smartapps and then present the achievements to date: Run-time optimizations, performance modeling, and moderately reconfigurable hardware.

1 Introduction

Many important applications are becoming large consumers of computing power, data storage and communication bandwidth. For example, applications such as ASCI multiphysics simulations, real-time target acquisition systems, multimedia stream processing and geographical information systems (GIS), all put tremendous strains on the computational, storage and communication capabilities of the most modern machines. There are several reasons why the performance of current distributed, heterogeneous systems is often disappointing. First, they are difficult to fully utilize because of the heterogeneity of the processing nodes which are interconnected through a non-homogeneous network with different inter-node latencies and bandwidths. Secondly, the system may change dynamically while the application is running. For example, nodes may fail or appear, network links may be severed, and other links may be established with different

* Research supported in part by NSF CAREER Award CCR-9734471, NSF CAREER Award CCR-9624315, NSF Grant ACI-9872126, NSF-NGS EIA-9975018, DOE ASCI ASAP Level 2 Grant B347886, and Hewlett-Packard Equipment Grants

S.P. Midkiff et al. (Eds.): LCPC 2000, LNCS 2017, pp. 82–96, 2001.

latencies and bandwidths. Finally, in order to obtain decent performance, the work has to be partitioned in a balanced manner.

Current distributed systems have a fairly compartmentalized approach to optimization: applications, compilers, operating systems and even hardware configurations are designed and optimized in isolation and without the knowledge of input data. There is too little information flow across these boundaries and no global optimization is even attempted. For example, many important activities managed by the operating system like paging activity, virtual-to-physical page mapping, I/O activity or data layout in disks are provided with little or no application customization. Since the compiler's analysis can discover much about an application's needs, performance could be boosted significantly if the OS provided hooks for the compiler, and possibly the user, to customize or tailor OS activities to the needs of a particular application. Current hardware is built for general purpose use to lower costs and has almost no tunable parameters that allow the compiler or the OS adjust it to specific application characteristics.

In addition to this lack of compiler/OS/hardware cooperation, a second important problem is that compilers do not necessarily know fully at compile time how an application will behave at run time because the run-time behavior of an application may partly depend on its input data. Consequently, compilers may generate conservative code that does not take advantage of characteristics of the program's input data. This precludes many aggressive optimizations related to code parallelization, and redundancy elimination. Moreover, we can only use expensive, generic methods for load balancing and memory latency hiding. If, instead, the compiler inserted code that, after reading the input data to the program at run-time, adaptively made optimization decisions, performance could be boosted significantly. Furthermore, at a higher level, the compiler may have the possibility of selecting an algorithm from a library of functionally equivalent modules. If this choice is made based on the specific instance of an application then large-scale gains can be obtained. For example, if the code calls for a sorting routine, the compiler can specialize this call to a specific parallel sort that matches both the input data to be sorted as well as the architecture on which it will be executed.

Our ultimate goal is the overall minimization of execution time of dynamic applications in parallel systems. Instead of building individual, *generally optimized* components (compilers, run-time systems, operating systems, hardware) that can work acceptably well with any application, we will subordinate the whole optimization process to the particular needs of a specific application. We will drive the optimization with the requirements of an individual program and for a specific set of input data. Moreover, the optimization will be carried out continuously to adapt to the dynamic, time varying needs of the application. The final form of the executable of an application will take shape only at run-time, after all input data has been analyzed. The resulting smart application (SMARTAPP) will monitor its performance and, when necessary, restructure itself and the underlying OS and hardware to its new characteristics. While this method may cost some additional overhead for every execution the resulting customized performance can more than pay off for long running codes.

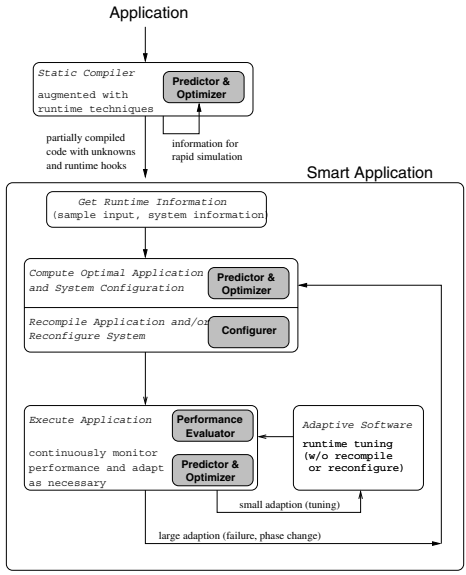


Fig. 1. Smart Application.

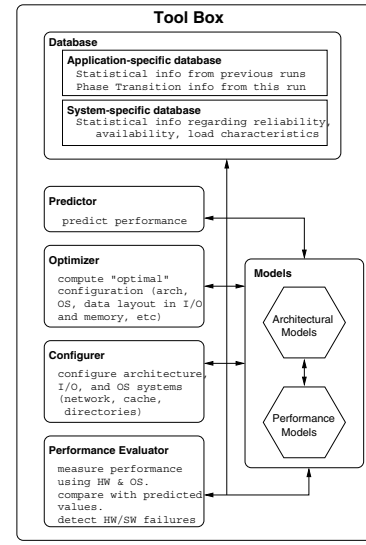


Fig. 2. ToolBox.

2 System Architecture

We now give a general overview of our system which includes components at various levels of development. Some features of SMARTAPPS have been implemented, others have been studied but have not yet been prototyped, while others are still in early stages. We give this high level architectural description that includes both accomplishments as well as work in progress in order to put our work in perspective. In the following sections we discuss in more detail those components that are in a more advanced state.

The adaptive run-time system, shown in Figures 1 and 2, consists of a nested multi-level adaptive feedback loop that monitors the application's performance and, based on the magnitude of deviation from expected performance, compensates with various actions. Such actions may be run-time software adaptation, re-compilation, or operating system and hardware reconfiguration. The system shown in Figure 1 uses techniques from a TOOLBOX shown in Figure 2. The TOOLBOX contains application and system specific databases and algorithms for performance evaluation, prediction and system reconfiguration. The tools are supported by architectural and performance models.

The **first stage** of preparing a dynamic application for execution occurs outside the proposed run-time system. It is a pre-compilation in which all possible static compiler optimizations are applied. However, for many of the more aggressive and effective code transformations, the needed information is not statically available. For example, if the code solves a sparse adaptive mesh-refinement problem, the initial mesh is read from an input file only at the beginning of the execution and is therefore not available for static compilation. In this case, the compiler may use *speculative transformations* which will be validated at run-time. We will generate an intermediate code that will contain all the

necessary compiler-internal information statically available, which will be combined with execution-time information to finish possible optimizations. Calls to generic algorithms or, when possible, parallel algorithm recognition and substitutions will be either left in their most general form or specialized to the extent permitted by static compiler analysis, e.g., type analysis [24].

The **second stage** in an application's life is driven by the run-time system. It starts by reading in and/or sampling the input data which are relevant to the 'unfinished' optimizations. This 'relevant' data is analyzed with fast, approximative methods and essential characteristics are extracted. The result of this analysis will place the instance of this application in a certain 'functioning domain' which represents the possible universe of forms that an application can take at run-time. Calls to routines that perform certain standard functions will be specialized by selecting from a linked library the algorithms and/or their implementations that match the 'functioning domain' (code and data) of this particular instantiation of the program.

Then, a fast RUN-TIME COMPILER, which will be developed from an existing restructurer, will finish the compilation process and generate a highly optimized and adaptable code, the SMART APPLICATION. This executable will include code for adaptive run-time techniques that allow the application to make on-the-fly decisions about various optimizations. To this end, we will use our techniques for detecting and exploiting loop level parallelism in various cases encountered in irregular applications [18, 20, 19]. Load balancing will be achieved through *feedback guided blocked scheduling* [8] which allows highly imbalanced loops to be block scheduled by predicting a good work distribution from previous measured execution times of iteration blocks.

For certain simple algorithms, which can be automatically recognized, e.g., reductions, the compiler will insert code that can substitute the sequential version with a parallel equivalent that best matches the data access pattern of the application. This adaptive parallel algorithm substitution technique can be implemented either through multi-version code (library calls) as is currently done, or through recompilation.

The result of static and dynamic compiler analysis of the application will also enable the program to call upon a tunable, modular OS to change some of its parameters (e.g., page mapping) and to perform some simple modification of the underlying architecture (e.g., type and/or number of system components). During this code generation phase, the compiler will generate (statically or at run-time) a list of specifications for the run-time environment. These application-level specifications are passed to the system configuration optimizer. The PREDICTOR and OPTIMIZER tools will use the application requirements and characteristics to compute an 'optimal' architectural configuration and tune the environment accordingly. In addition to the OS tuning we can perform architectural modifications when feasible. As we show in Section 5 we have simulated the possibility of customizing communication protocols (e.g., specialized cache coherence protocols). In the future we hope to be able specialize processors for computing or communication and distribute the workload between 'classical' processors and processors in memory (IRAM).

In the next sections we elaborate on some of the currently implemented components of the presented SMARTAPPS architecture.

3 Compiler Generated Run-Time Optimizations

The SMART APPLICATION mainly consists of a run-time library embedded by the compiler in the application and which can dynamically select compiler optimizations (e.g., loop parallelization or scheduling for load balance). Some non-intrusive architectural reconfiguration and operating system level tuning may also be employed to obtain fast, low overhead performance improvement. We plan to integrate such adaptive techniques into the application by extending current static and run-time technologies and by developing completely new ones. In the following sections we detail some of these optimization methods and show how they can be incorporated into an integrated adaptive system for dynamic, heterogeneous computing.

3.1 Run-Time Parallelization

We have developed several techniques [18, 19, 20] that can detect and exploit loop level parallelism in various cases encountered in irregular applications: (i) a speculative method to detect fully parallel loops (The LRPD Test), (ii) an inspector/executor technique to compute wavefronts (sequences of mutually independent sets of iterations that can be executed in parallel) and (iii) a technique for parallelizing `while` loops (`do` loops with an unknown number of iterations and/or containing linked list traversals). We now briefly describe the utility of some of these techniques; details of their design can be found in [8, 19, 20] and other related publications.

Partially Parallel Loop Parallelization. We have previously developed a run–time technique for finding an optimal parallel execution schedule for a partially parallel loop [18]. Given the original loop, the compiler generates *inspector* code that performs run–time preprocessing (based on a sorting algorithm) of the loop's access pattern, and *scheduler* code that schedules (and executes) the loop iterations. The inspector is fully parallel, uses no element-wise synchronization, and can implement at run–time array privatization and reduction parallelization. Unfortunately this method is not generally applicable because a proper, side-effect free inspector cannot be extracted from a loop where address and data computation form a dependence cycle.

The Recursive LRPD Test. In previous work we have introduced the LRPD test for `DOALL` parallelization which speculatively executes a loop in parallel and tests subsequently if any data dependences could have occurred [19]. If the test fails, the loop is re–executed sequentially. To qualify more parallel loops, *array privatization* and *reduction parallelization* can be speculatively applied and their validity tested after loop termination. It can be shown that if the LRPD test passes, i.e., the loop is in fact fully parallel, then a significant portion of the ideal speedup of the loop is obtained. The drawback of this method is that if the test fails a slowdown equal to the parallel speculative execution of the loop may be experienced.

We have developed a new technique that can extract the maximum available parallelism from a partially parallel loop and that can be applied to any loop (even if no proper inspector can be extracted) and requires less memory overhead. The main idea of the Recursive LRPD test [8] is that in any block-scheduled loop executed under the processor-wise LRPD test with copy-in, the chunks of iterations that are less than or

equal to the source of the first detected dependence arc are always executed correctly. Only the processors executing iterations larger or equal to the earliest sink of any dependence arc need to re-execute their portion of work. Thus only the remainder of the work (of the loop) needs to be re-executed, which can represent a significant saving over the previous LRPD test method (which would re-execute the whole loop sequentially).

To re-execute the fraction of the iterations assigned to the processors that may have worked off erroneous data we need to repair the unsatisfied dependences. This can be accomplished by initializing their privatized memory with the data produced by the lower ranked processors. Alternatively, we can commit (i.e., copy-out) the correctly computed data from private to shared storage and use on-demand copy-in during re-execution. We then re-apply recursively the fully parallel LRPD test on the remaining iterations until all processors have correctly finished their work. For loops with few cross-processor dependences we can expect to finish in only a few parallel steps. We have used two different strategies when re-executing a part of the loop: We can re-execute only on the processors that have incorrect data and leave the rest of them idle (NRD), or, we can redistribute at every stage the remainder of the work across all processors (RD). There are pros and cons for both approaches. Through redistribution of the work we employ all processors all the time and thus the execution time of every stage decreases (instead of staying constant, as in the NRD case). The disadvantage is that we may uncover new dependences across processors which were satisfied before by executing on the same processor. Moreover, there is a 'hidden' but potentially large cost associated with work redistribution: more remote misses during loop execution due to data redistribution between the stages of the test. The worst case time complexity for no redistribution (NRD) is the cost of a sequential execution. There are at most p steps performing n/p work, where p is the number of processors and n is the number of iterations. In the RD (with redistribution) case we will take progressively less time because we execute in p processors decreasing the amount of work. The number of steps is heavily dependent on the distribution of data dependences of the loop. For example, if we assume that at every step we perform correctly 1/2 the work then the total time is less than twice the fully parallel execution time of the loop. In practice we have obtained better results by adopting a hybrid method which redistributes until the predicted execution time of the remaining work is less than the overhead associated with re-distribution. In other words, we redistribute until the potential benefit of using more processors is outweighed by the cost. From that point on we continue without redistribution. A potential drawback is that the loop needs to be statically block scheduled in increasing order of iteration. The negative impact of this limitation can be reduced through dynamic feedback guided scheduling [7]. By applying this new method exclusively we can remove the uncertainty or unpredictability of execution time associated with the LRPD test – we can guarantee that a speculatively parallelized program will run at least as fast as its sequential version with some additional (minor) testing overhead.

We have implemented the Recursive LRPD test in both RD and NRD flavors and applied it to the three most important loops in TRACK, a Perfect code. The implementation is partially done by our run-time pass in Polaris (to automatically apply the simple LRPD test) and then additional code has been inserted manually. Our experimental test-

bed is a 16 processor ccUMA HP-V2200 system running HPUX11. It has 4Gb of main memory and 4Mb single level caches.

The main loops in TRACK are EXTEND_400, NLFILT_300 and FPTRACK_300. They account for $\approx 90\%$ of sequential execution time. We have increased the input set to increase the execution time as well as all associated data structures. The degree of parallelism in the loop from NLFILT is very input sensitive and ranges from fully parallel to a significant number of cross-processor dependences. All considered loops are very load imbalanced and thus, until our feedback guided load balancing is fully implemented, causes low speedups. Figures 3 (a-c) show the speedups for individual loops and Figure 3(d) shows the speedup for the entire program. Previous to this technique this code was considered sequential.

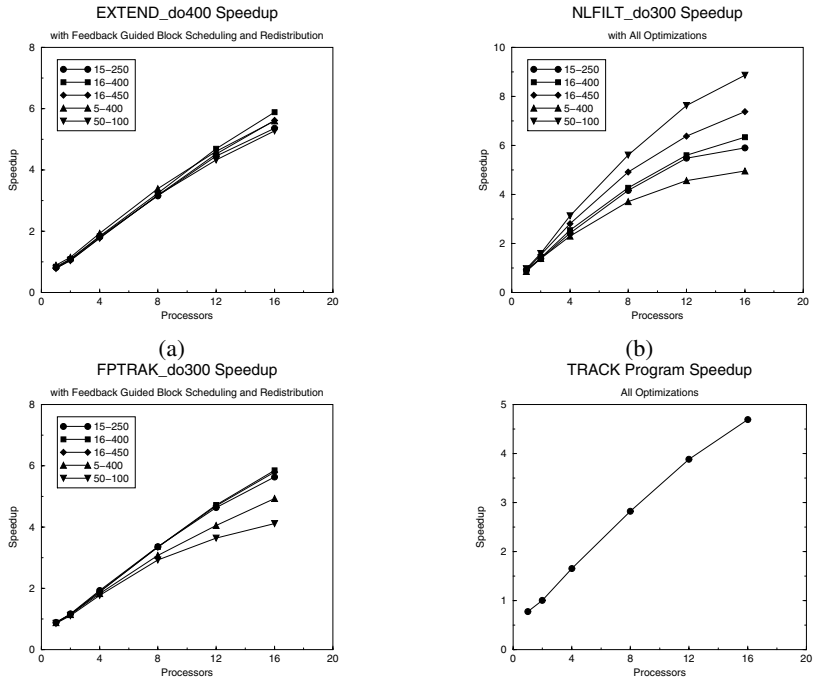


Fig. 3. .

3.2 Adaptive Algorithm Selection: Choose the Right Method for Each Case

Memory accesses in irregular programs take a variety of patterns and are dependent on the code itself as well as on their input data. Moreover, some codes are of a dynamic nature, i.e., they modify their behavior during their execution. For example, they might simulate position dependent interactions between physical entities.

A special and very frequent case of loop dependence pattern occurs in loops which implement reduction operations. In particular, reductions (also known as updates) are

at the core of a very large number of algorithms and applications – both scientific and otherwise – and there is a large body of literature dealing with their parallelization.[1] It is difficult to a find a reduction parallelization algorithm (or for that matter, other

APP	MO	DIM	SP	CON	CHR	Recom. Scheme	Experimental Result
Irreg	2	100,000	25	100	0.92	rep	rep > ll > sel > lw
- DO 100		500,000	5	20	0.71	lw	lw > rep > ll > sel
		1,000,000	1.25	5	0.40	lw	lw > rep > ll > sel
		2,000,000	0.25	1	0.26	sel	sel > lw > ll > rep
Nbf	1	25,600	25	200	0.25	ll	sel > ll > rep > lw
- DO 50		128,000	6.25	50	0.25	sel	sel > ll > rep > lw
		256,000	0.625	5	0.25	sel	sel > ll > rep > lw
		1,280,000	0.25	2	0.25	sel	sel > ll > rep > lw
Moldyn	2	16,384	23.94	95.75	0.41	rep	rep > ll > sel > lw
- ComputeForces loop		42,592	7.75	31	0.36	rep	rep > ll > sel > lw
		70,304	1.69	6.75	0.33	ll	ll > rep > sel > lw
		87,808	0.375	1.5	0.29	ll	ll > rep > sel > lw
Spark98	1	30,169	0.625	5	0.18	sel	sel > ll > rep > lw
- smvpthread() loop		7,294	0.6	4.8	0.2	sel	ll > sel > rep > lw
Charmm	2	332,288	35.88	17.9	0.14	sel	ll > sel > rep > lw
- DO 78			17.94	8.97	0.15	sel	ll > sel > rep > lw
		664,576	1.12	4.48	0.13	sel	ll > sel > rep > lw
Spice	28	186,943	0.14	0.04	0.125	hash	hash > ll > rep
- bjt100		99,190	0.20	0.06	0.125	hash	hash > ll > rep
		89,925	0.16	0.05	0.125	hash	hash > ll > rep
		33,725	0.16	0.05	0.126	hash	hash > ll > rep

Fig. 4. The data has been obtained from the execution of the applications 8 processors. DIM: number of reduction elements; SP: sparsity; CON: connectivity; CHR: ratio of total number of references to space needed for per processor replicated arrays; MO: mobility.

optimizations) that will work well in all cases. We have designed an adaptive scheme that will detect the type of reference pattern through static (compiler) and dynamic (run-time) methods and choose the most appropriate scheme from a library of already implemented choices [24]. To find the best choice we establish a taxonomy of different access patterns, devise simple, fast ways to recognize them, and model the various old and newly developed reduction methods in order to find the best match. The characterization of the access pattern is performed at compile time whenever possible, and otherwise, at run-time, during an inspector phase or during speculative execution.

From the point of view of optimizing the parallelization of reductions (i.e., selecting the best parallel reduction algorithm) we recognize several characteristics of memory

[1] A reduction variable is a variable whose value is used in one associative and commutative operation of the form $x = x \otimes exp$, where $\otimes$ is the operator and x does not occur in exp or anywhere else in the loop.

references to reduction variables. **CH** is a histogram which shows the number of elements referenced by a certain number of iterations. **CHR** is the ratio of the total number of references (or the sum of the **CH** histogram) and the space needed for allocating replicated arrays across processors. **CON**, the Connectivity of a loop, is a ratio between the number of iterations of the loop and the number of distinct memory elements referenced by the loop [12]. The **Mobility (MO)** per iteration of a loop is directly proportional to the number of distinct elements that an iteration references. The **Sparsity (SP)** is the ratio of referenced elements to the dimension of the array. The **DIM** measure gives the ratio between the reduction array dimension and cache size. If the program is dynamic then changes in the access pattern will be collected, as much as possible, in an incremental manner. When the changes are significant enough (a threshold that is tested at run-time) then a re-characterization of the reference pattern is needed.

Our strategy is to identify the regular components of each irregular pattern (including uniform distribution), isolate and group them together in space and time, and then apply the best reduction parallelization method to each component. We have used the following novel and previously known parallel reduction algorithms: local write (**lw**) [12] (an 'owner compute' method), private accumulation and global update in replicated private arrays (**rep**), replicated buffer with links (**ll**), selective privatization (**sel**), sparse reductions with privatization in hash tables (**hash**). Our main goal, once the type of pattern is established, is to choose the appropriate reduction parallelization algorithm, that is, the one which best matches these characteristics. To make this choice we use a decision algorithm that takes as input measured, real, code characteristics, and a library of available techniques, and selects an algorithm for the given instance.

The table shown in Fig.4 illustrates the experimental validation of our method. All memory reference parameters were computed at run-time. The result of the decision process is shown in the "Recommended scheme" column. The final column shows the actual experimental speedup obtained with the various reduction schemes which are presented in decreasing order of their speedup. For example, for Irreg, the model recommended the use of Local Write. The experiments confirm this choice: *lw* is listed as having the best measured speedup of all schemes.

In the experiment for the SPICE loop, the hash table reduces the allocated and processed space to such an extent that, although the setup of a hash table is large, the performance improves dramatically. It is the only example where hash table reductions represent the best solution because of the very sparse nature of the references. We believe that codes in C would be of the same nature and thus benefit from hash tables. There are no experiments with the Local Write method because iteration replication is very difficult due to the modification of shared arrays inside the loop body.

4 The Toolbox: Modeling, Prediction, and Optimization

In this section, we describe our current results in developing a performance PREDICTOR whose predictions will be used to select among various algorithms, and to help diagnose inefficiencies and identify potential optimizations.

Significant work has been done in low-level analytical models of computer architectures and applications [23, 1, 22, 17]. While such analytical models had fallen out of

favor, being replaced by comprehensive simulations, they have recently been enjoying a resurgence due the need to model large-scale NUMA machines and the availability of hardware performance counters [13]. However, these models have mainly been used to analyze the performance of various architectures or system-level behaviors.

In [2], we propose a cost model that we call *F*, which is based on values commonly provided by hardware performance monitors, that displays superior accuracy to the BSP-like models (our results on the SGI PowerChallenge use the MIPS R10000 hardware counters [21]). Function *F* is defined under the assumption that the running time is determined by one of the following factors: (1) the accesses issued by some processor at the various levels of the hierarchy, (2) the traffic on the interconnect caused by accesses to main memory, or (3) bank contention caused by accesses targeting the same bank. For each of the above factors, we define a corresponding function (*F1*, *F2*, and *F3*, resp.) which should dominate when that behavior is the limiting factor on performance. That is, we set *F*= $\max\{$ *F1*, *F2*, *F3*$\}$. The functions are linear relations of values measurable by hardware performance counters, such as loads/stores issued, L1 and L2 misses and L1 and L2 write-backs, and the coefficients are determined from micro-benchmarking experiments designed to exercise the system in that particular mode.

A complete description of *F*, including detailed validation results, can be found in [2]. We present here a synopsis of the results. The function *F* was compared with three *BSP*-like cost functions based, respectively, on the *Queuing Shared Memory* (*QSM*) [11] and the (d, x)-*BSP* [6], which both embody some aspects of memory contention, and the *Extended BSP* (*EBSP*) model [14], which extends the *BSP* to account for unbalanced communication. Since the *BSP*-like functions do not account for the memory hierarchy, we determined an optimistic (min) version and a pessimistic (max) version for each function. The accuracy of the *BSP*-like functions and *F* were compared on an extensive suite of synthetic access patterns, three bulk-synchronous implementations of parallel sorting, and the NAS Parallel Benchmarks [10]. Specifically, we determined measured and predicted times (indicated by T_M and T_P, respectively) and calculated the prediction error as $\mathrm{ERR} = \frac{\max\{T_M, T_P\}}{\min\{T_M, T_P\}}$, which indicates how much smaller or larger the predicted time is with respect to the measured time.

A summary of our findings regarding the accuracy of the *BSP*-like functions and *F* is shown in Tables 1-3, where we report the maximum value of ERR over all runs (when omitted, the average values of ERR are similar). Overall, the *F* function is clearly superior to the *BSP*-like functions. The validations on synthetic access patterns (Table 1) underscore that disregarding hierarchy effects has a significant negative impact on accuracy. Moreover, *F*'s overall high accuracy suggests that phenomena that were disregarded when designing it (such as some types of coherency overhead) have only a minor impact on performance. Since the sorting algorithms (Table 3) exhibit a high degree of locality, we would expect the optimistic versions of the *BSP*-like functions to perform much better than their pessimistic counterparts, and indeed this is the case (errors are not shown for $EBSP_{\min}$ and $DXBSP_{\min}$ because they are almost identical to the errors for $QSM_{\min}$). A similar situation occurs for the MPI-based NAS benchmarks (Table 2).

Performance predictions from a HW counter-based model. One of the advantages of the *BSP*-like functions over the counter-based function *F*, is that, to a large extent, the compiler or programmer can determine the input values for the function. While the

Synthetic Access Patterns – ERRs			
Function		AVG ERR	MAX ERR
QSM	MIN	24.15	88.02
	MAX	53.85	636.79
EBSP	MIN	24.15	88.02
	MAX	27.29	648.35
DXBSP	MIN	6.36	31.84
	MAX	34.8	411.46
F		1.19	1.91

Table 1. Synthetic Access Patterns.

NAS Parallel Benchmarks – MAX ERR					
	QSM		*EBSP*	*DXBSP*	
NPB	MIN	MAX	MAX	MAX	*F*
CG	2.46	258.31	210.10	166.91	1.46
EP	2.42	262.53	252.32	169.64	1.02
FT	2.05	309.40	245.64	199.92	1.63
IS	1.57	404.81	354.47	261.57	1.39
LU	2.15	295.01	236.80	190.62	1.32
MG	1.57	403.48	289.11	260.71	1.73
BT	2.77	229.68	189.08	148.41	1.05
SP	2.13	298.69	194.21	193.00	1.05

Table 2. NAS Parallel Benchmarks.

Sorting Programs – MAX ERR						
		QSM		*EBSP*	*DXBSP*	
Sort	**SStep**	MIN	MAX	MAX	MAX	*F*
Radix	SS1	2.12	400.14	320.48	258.55	1.41
	SS4	2.72	321.56	302.11	207.78	1.39
Sample	SS2	2.17	320.95	252.25	207.38	1.15
	SS4	2.89	287.72	247.31	185.91	1.11
	SS5	2.58	361.36	327.08	233.49	1.26
Column	SS1	3.44	268.13	205.23	173.25	1.06
	SS2	2.46	268.13	264.49	173.25	2.05
	SS3	2.88	230.37	228.11	148.85	1.88
	SS4	2.61	245.56	247.10	158.67	2.09
	SS5	1.36	484.93	280.03	313.34	1.16

Table 3. Sorting algorithms for selected supersteps.

		Errors for *F* (Measured)		**Errors for *F* (Estimated)**	
Sort	**SStep**	AVG	MAX	AVG	MAX
Radix	SS1	1.22	1.32	1.01	1.09
	SS4	1.11	1.16	1.13	1.16
Sample	SS2	1.05	1.09	1.20	1.21
	SS4	1.06	1.11	1.03	1.04
	SS5	1.13	1.17	1.24	1.26
Column	SS1	1.12	2.49	1.17	1.72
	SS2	1.80	1.89	2.02	2.12
	SS3	1.66	1.69	1.84	1.87
	SS4	1.78	1.83	2.05	2.06
	SS5	1.16	1.17	1.88	1.90

Table 4. Sorting algorithms: comparison of *F*'s accuracy with measured vs. estimated counters.

counter-based function exhibits excellent accuracy, it seems that one should actually run the program to obtain the required counts, which would annihilate its potential as a performance predictor. However, if one could guess the counter values in advance with reasonable accuracy, they could then be plugged into *F* to obtain accurate predictions. For example, in some cases meaningful estimates for the counters might be derived by extrapolating values for large problem sizes from pilot runs of the program on small input sets (which could be performed at run-time by the adaptive system). To investigate this issue, we developed least-squares fits for each of the counters used in *F* for those supersteps in our three sorting algorithms that had significant communication. The input size n of the sorting instance was used as the independent variable. For each counter, we obtained the fits on small input sizes ($n/p = 10^5 \cdot i$, for $1 \leq i \leq 5$), and then used the fits to forecast the counter values for large input sizes ($n/p = 10^5 \cdot i$, for $5 < i \leq 10$). These estimated counter values were then plugged in *F* to predict the execution times for the larger runs. The results of this study are summarized in Table 4. It can be seen

that in *all* cases, the level of accuracy of F using the extrapolated counter values was not significantly worse than what was obtained with the actual counter values. These preliminary results indicate that at least in some situations a hardware counter-based function does indeed have potential as an *a priori* predictor of performance. Currently, we are working on applying this strategy to other architectures, including the HP V-Class and the SGI Origin 2000.

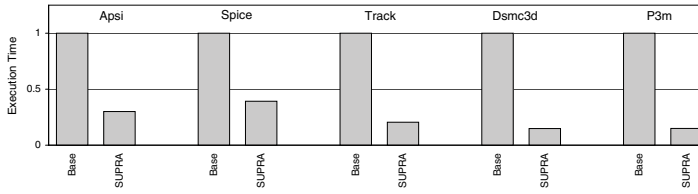
5 Hardware

Smart applications exploit their maximum potential when they execute on reconfigurable hardware. Reconfigurable hardware provides some hooks that enable it to work in different modes. In this case, smart applications, once they have determined their true behavior statically or dynamically, can actuate these hooks and conform the hardware for the application. The result is large performance improvements.

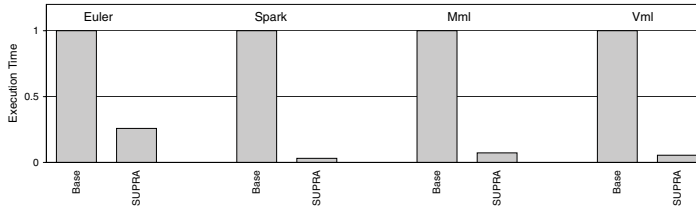
A promising area for reconfigurable hardware is the hardware cache coherence protocol of a CC-NUMA multiprocessor. In this case, we can have a base cache coherence protocol that is generally high-performing for all types of access patterns or behaviors of the application. However, the system can also support other specialized cache coherence protocols that are specifically tuned to certain application behaviors. Applications should be able to select the type of cache coherence protocol used on a code section basis. We provide two examples of specialized cache coherence protocols here. Each of these specialized protocols is composed of the base cache coherence transactions plus some additional transactions that are suited to certain functions. These two examples are the *speculative parallelization protocol* and the *advanced reduction protocol*.

The speculative parallelization protocol is used profitably in sections of a program where the dependence structure of the code is not analyzable by the compiler. In this case, instead of running the code serially, we run it in parallel on several processors. The speculative parallelization protocol contains extensions that, for each protocol transaction, check if a dependence violation occurred. Specifically, a dependence violation will occur if a logically later thread reads a variable before a logically earlier thread writes to it. The speculative parallelization protocol can detect such violations because it tags every memory location that is read and written, with the ID of the thread that is performing the access. In addition, it compares the tag ID before the access against the ID of the accessing thread. If a dependence violation is detected, an interrupt runs, repairs the state, and restarts execution. If such interrupts do not happen too often, the code executes faster in parallel with the speculative parallelization protocol than serially with the base cache coherence protocol. More details can be found in [25, 26, 27].

The advanced reduction protocol is used profitably in sections of a program that contain reduction operations. In this case, instead of transforming the code to optimize these reduction operations in software, we simply mark the reduction variables and run the unmodified code under the new protocol. The protocol has extensions such that, when a processor accesses the reduction variable, it makes a privatized copy in its cache. Any subsequent accumulation on the variable will not send invalidations to other privatized copies in other caches. In addition, when a privatized version is displaced from a cache, it is sent to its original memory location and accumulated onto the existing



(a)



(b)

Fig. 5. Execution time improvements by reconfiguring the cache coherence protocol hardware to support (a) speculative parallelization, and (b) advanced reduction operations.

value. With these extensions, the protocol reduces to a minimum the amount of data transfer and messaging required to perform a reduction in a CC-NUMA. The result is that the program runs much faster. More details can be found in [27].

We now see the impact of cache coherence protocol recongurability on execution time. Figure 5 compares the execution time of code sections running on a 16-processor simulated multiprocessor like the SGI Origin 2000 [21]. We compare the execution time under the base cache coherence protocol and under a reconfigurable protocol called SUPRA. In Figures 5(a) and 5(b) SUPRA is reconfigured to be the speculative parallelization and advanced reduction protocols, respectively. In both charts, for each application, the bars are normalized to the execution time under Base.

From the figures, we see that the ability to reconfigure the cache coherence protocol to conform to the individual application's characteristics is very beneficial. The code sections that can benefit from the speculative parallelization protocol (Figure 5(a)), run on average 75% faster under the new protocol, while those that. can benefit from the advanced reduction protocol (Figure 5(b)) run on average 85% faster under the new protocol.

6 Conclusions and Future Work

So far we have made good progress on the development of many the components of SmartApps. We will further develop these and combine them into an integrated system.

The performance of parallel applications is very sensitive to the type and quality of operating system services. We therefore propose to further optimize SmartApps by

interfacing them with an existing customizable OS. While there have been several proposals of modular, customizable OSs, we plan to use the K42 [3] experimental OS from IBM, which represents a commercial-grade development of the TORNADO system [16, 4]. Instead of allowing users to actually alter or rewrite parts of the OS and thus raise security issues, the K42 system allows the selective and parametrized use of OS modules (objects). Additional modules can be written if necessary but no direct user access is allowed to them. This approach will allow our system to configure the type of services that will contribute to the full optimization of the program.

So far we have presented various run-time adaptive techniques that a compiler can safely insert into an executable under the form of multiversion code, and that can adapt the behavior of the code to the various dynamic conditions of the data as well as that of the system on which it is running. Most of these optimization techniques have to perform a test at run-time and decide between multi-version sections of code that have been pre-optimized by the static compilation process. The multi-version code solution may, however require an impractical number of versions. Applications exhibiting partial parallelism could be greatly sped up through the use of selective, point-to-point synchronizations and whose placement information is available only at run-time. Motivated by such ideas we plan on writing a two stage compiler. The first will identify which performance components are input dependent and the second stage will compile at run time the best solution. We will target what we believe are the most promising sources of performance improvement for an application executing on a large system: increase of parallelism, memory latency and I/O management. In contrast to the run-time compilers currently under development [5, 9, 15] which mainly rely on the benefits of partial evaluation, we are targeting very high level transformations, e.g., parallelization, removal of (several levels) of indirection and algorithm selection.

References

[1] S. Adve, V. Adve, M. Hill, and M. Vernon. Comparison of Hardware and Software Cache Coherence Schemes. In *Proc. of the 18th ISCA*, pp. 298–308, June 1991.

[2] N. M. Amato, J. Perdue, A. Pietracaprina, G. Pucci, and M. Mathis. Predicting performance on smps. a case study: The SGI Power Challenge. In *Proc. IPDPS*, pp. 729–737, May 2000.

[3] M. Auslander, H. Franke, B. Gamsa, O. Krieger, and M. Stumm. Customization lite. In *Proc. of 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, May 1997.

[4] J. Appavo B. Gamsa, O. Krieger and M. Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proc. of OSDI*, 1999.

[5] B. Grant, et. al. An evaluation of staged run-time optimizations in Dyce. In *Proc. of the SIGPLAN 1999 PLDI, Atlanta, GA*, May 1999.

[6] G.E. Blelloch, P.B. Gibbons, Y. Mattias, and M. Zagha. Accounting for memory bank contention and delay in high-bandwidth multiprocessors. *IEEE Trans. Par. Dist. Sys.*, 8(9):943–958, 1997.

[7] J. Mark Bull. Feedback guided dynamic loop scheduling: Algorithms and experiments. In *EUROPAR98*, Sept., 1998.

[8] F. Dang and L. Rauchwerger. Speculative parallelization of partially parallel loops. In *Proc. of the 5th Int. Workshop LCR 2000, Lecture Notes in Computer Science*, May 2000.

[9] D. Engler. Vcode: a portable, very fast dynamic code generation system. In *Proc. of the SIGPLAN 1996 PLDI Philadelphia, PA*, May 1996.

[10] D. Bailey *et al.* The NAS parallel benchmarks. *Int. J. Supercomputer Appl.*, 5(3):63–73, 1991.

[11] P. B. Gibbons, Y. Matias, and V. Ramachandran. Can a shared-memory model serve as a bridging-model for parallel computation? In *Proc. ACM SPAA*, pp. 72–83, 1997.

[12] H. Han and C.-W. Tseng. Improving compiler and run-time support for adaptive irregular codes. In *PACT'98*, Oct. 1998.

[13] R. Iyer, N. Amato, L. Rauchwerger, and L. Bhuyan. Comparing the memory system performance of the HP V=AD-Class and SGI Origin 2000 multiprocessors using microbenchmarks and scientific applications. In *Proc. of ACM ICS*, pp. 339–347, June 1999.

[14] B. H. H. Juurlink and H. A. G. Wijshoff. A quantitative comparison of parallel computation models. In *Proc. of ACM SPAA*, pp. 13–24, 1996.

[15] D. Keppel, S. J. Eggers, and R. R. Henry. A case for runtime code generation. TR UWCSE 91-11-04, Dept. of Computer Science and Engineering, Univ. of Washington, Nov. 1991. .

[16] O. Krieger and M. Stumm. Hfs: A performance-oriented flexible file system based on building-block compositions. *IEEE Trans. Comput.*, 15(3):286–321, 1997.

[17] S. Owicki and A. Agarwal. Evaluating the performance of software cache coherency. In *Proc. of ASPLOS III*, April 1989.

[18] L. Rauchwerger, N. Amato, and D. Padua. A Scalable Method for Run-time Loop Parallelization. *Int. J. Paral. Prog.*, 26(6):537–576, July 1995.

[19] L. Rauchwerger and D. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. *IEEE Trans. on Par. and Dist. Systems*, 10(2), 1999.

[20] L. Rauchwerger and D. Padua. Parallelizing WHILE Loops for Multiprocessor Systems. In *Proc. of 9th IPPS*, April 1995.

[21] Silicon Graphics Corporation 1995. *SGI Power Challenge: User's Guide*, 1995.

[22] R. Simoni and M. Horowitz. Modeling the Performance of Limited Pointer Directories for Cache Coherence. In *Proc. of the 18th ISCA*, pp. 309–318, June 1991.

[23] J. Torrellas, J. Hennessy, and T. Weil. Analysis of Critical Architectural and Programming Parameters in a Hierarchical Shared Memory Multiprocessor. In *ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems*, pp. 163–172, May 1990.

[24] H. Yu and L. Rauchwerger. Adaptive reduction parallelization. In *Proc. of the 14th ACM ICS, Santa Fe, NM*, May 2000.

[25] Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for Speculative Run-Time Parallelization in Distributed Shared-Memory Multiprocessors. In *Proc. of HPCA-4*, pp. 162–173, 1998.

[26] Y. Zhang, L. Rauchwerger, and J. Torrellas. Speculative Parallel Execution of Loops with Cross-Iteration Dependences in DSM Multiprocessors. In *Proc. of HPCA-5*, Jan. 1999.

[27] Ye Zhang. *DSM Hardware for Speculative Parallelization*. Ph.D. Thesis, Department of ECE, Univ. of Illinois, Urbana, IL, Jan. 1999.

Extending Scalar Optimizations for Arrays

David Wonnacott

Haverford College, Haverford, PA 19041, USA
davew@cs.haverford.edu
http://www.haverford.edu/cmsc/davew/index.html

Abstract. Traditional techniques for array analysis do not provide dataflow information, and thus traditional dataflow-based scalar optimizations have not been applied to array elements. A number of techniques have recently been developed for producing information about array dataflow, raising the possibility that dataflow-based optimizations could be applied to array elements. In this paper, we show that the value-based dependence relations produced by the Omega Test can be used as a basis for generalizations of several scalar optimizations.

Keywords: array dataflow, constant folding, constant propagation, copy propagation, dead code elimination, forward substitution, unreachable code elimination, value-based array dependences

1 Introduction

Traditional scalar analysis techniques produce information about the flow of values, whereas traditional array data dependence analysis gives information about memory aliasing. Thus, many analysis and optimization techniques that require dataflow information have been restricted to scalar variables, or applied to entire arrays (rather than individual array elements). A number of techniques have recently been developed for accurately describing the flow of values in array variables [1, 2, 3, 4, 5, 6, 7, 8]. These are more expensive than scalar dataflow analysis, and the developers (except for Sarkar and Knobe) have focused on optimizations with very high payoff, such as automatic parallelization. However, once the analysis has been performed, we can apply the result to other optimizations as well.

In this paper, we give algorithms for dead-code elimination, constant propagation and constant folding, detection of constant conditions and elimination of unreachable code, copy propagation, and forward substitution for arrays. Our algorithms have two important properties: First, they work with *elements* of arrays and *executions* of statements, rather than treating each variable or statement as a monolithic entity. Second, they are not based on descriptions of finite sets of values, and thus (unlike the techniques of [8]) can detect the constant array in Figure 1(b) as well as that in Figure 1(a).

S.P. Midkiff et al. (Eds.): LCPC 2000, LNCS 2017, pp. 97–111, 2001.

```
REAL*8 A(0:3)

    . . .

A(0) = -8.0D0/3.0D0
A(1) =  0.0D0
A(2) =  1.0D0/6.0D0
A(3) =  1.0D0/12.0D0
```

(a) `A` in `107.MGRID` [8]

```
      DO 10 J = 1, JMAX
         JPLUS(J) = J+1
10    CONTINUE

      IF ( .NOT. PERIDC ) THEN
         JPLUS(JMAX) = JMAX
      ELSE
         JPLUS(JMAX) = 1
      ENDIF
```

(b) `JPLUS` in `initia` in `ARC2D` [9]

Fig. 1. Definitions of Constant Arrays

2 Representing Programs with Integer Tuple Sets and Relations

Information about scalar dataflow is generally defined at the statement level, without distinguishing different executions of a statement in a loop. In contrast, information about array dataflow, and optimizations based on this information, should be based on *executions* of each statement, as the set of elements used may vary with the loop iteration. We use the Omega Library [10] to represent and manipulate this information.

The remainder of this section reviews our representations of iteration spaces, data dependences, and mappings from iterations to values, as well as the operations that we apply to these representations. Readers who are familiar with earlier work of the Omega Project may wish to skip to the new material, which begins with Section 3.

2.1 Representing Sets of Iterations

We identify each execution of a given statement with a tuple of integers giving the indices of the surrounding loops. For example, consider the definition of `JPLUS` the `ARC2D` benchmark of the perfect club benchmark set [9]. We will refer to three defining statements as S (the definition in the loop), T (the definition in the `then` branch of the `if`), and U (the definition in the `else` branch). Statement S is executed repeatedly – we label the first execution [1], the second [2], etc., up to $[JMAX]$. We can represent the set $iterations(S)$, the of iterations of S, using constraints on a variable J (representing the loop index):

$$iterations(S) = \{\ [J] \mid 1 \leq J \leq JMAX\ \}.$$

These constraints can refer to values not known at compile time (such as `JMAX`), allowing us to represent a set without having an upper bound on its size. The Omega Library lets us use arbitrary affine constraints (i.e., the expressions being compared can be represented by sums of constants and variables multiplied

by constant factors), though the library is not efficient over this entire domain. However, our previous experimental work indicates that, for the constraints that arise in practice, it can perform most of the operations discussed in this paper efficiently (see Section 2.4 and [7] for details).

We can use constraints to describe a set of loop iterations (as above) or the set of values of symbolic constants for which a statement executes. For example,

$$iterations(T) = \{\ [\ ] \mid \neg PERIDC\ \}.$$

Executions of statements inside no loops correspond to zero-element tuples (as above); executions in nested loops correspond to multi-element tuples. The set of executions of the first assignment to `JP1` in Figure 2 (Statement V) is

$$iterations(V) = \{\ [N, J] \mid 2 \leq N \leq 4 \wedge JLOW \leq J \leq JUP\ \}.$$

```
      DO 300 N = 2,4
         DO 210 J = JLOW,JUP
            JP1 = JPLUS(J)
            JP2 = JPLUS(JP1)
            DO 212 K = KLOW,KUP
C              ...
               WORK(J,K,4) = -(C2 + 3.*C4 + C4M)*XYJ(JP1,K)
               WORK(J,K,5) = XYJ(JP2,K)*C4
212         CONTINUE
210      CONTINUE
C
      IF(.NOT.PERIDC)THEN
         J = JLOW
         JP1 = JPLUS(J)
         JP2 = JPLUS(JP1)
         DO 220 K = KLOW,KUP
C           ...
            WORK(J,K,4) = -(C2 + 3.*C4 + C4M)*XYJ(JP1,K)
            WORK(J,K,5) = XYJ(JP2,K)*C4
220      CONTINUE
C      ...
C      (uses of WORK array follow)
```

Fig. 2. Uses of `JPLUS` in `300.stepfx` in `ARC2D` (Simplified)

2.2 Representing Mappings

We can use relations between integer tuples to describe mappings, such as a mapping from loop iterations to values. For example, the fact that iteration J of Statement S produces the value $J + 1$ can be described with the relation

$$value(S) = \{\ [J] \rightarrow [J + 1] \mid 1 \leq J \leq JMAX\ \}.$$

2.3 Representing Dependences as Relations

Array data dependence information, whether based on memory aliasing or value flow, can also be described with integer tuple relations. The domain of the tuple represents executions of the source of the dependence, and the range executions of the sink. For example, if Figure 1(b) were followed immediately by Figure 2, then the value-based dependence from the definition of `JPLUS` in Statement S to the use in Statement V, $\delta_{S\rightarrow V}$, would be:

$$\delta_{S\rightarrow V} = \{\ [J] \rightarrow [N, J] \mid 1, JLOW \leq J \leq JUP, JMAX - 1 \wedge 2 \leq N \leq 4\ \}$$

Note that the constraint $J \leq JMAX - 1$ indicates that there is no dependence from iteration $[JMAX]$ of the definition, since that definition is killed by the conditional assignment.

When all constraints are affine, the analysis we describe in [4, 7] can produce exact value-based dependence relations. This occurs for any procedure in which control flow consists of `for` loops and `if`s, and all loop bounds, conditions, and subscripts are affine functions of the surrounding loop indices and a set of symbolic constants. In other cases, our analysis may produce an approximate result.

2.4 Operations on Sets and Relations

Our algorithms use relational operations such as union, intersection, subtraction, join, and finding or restricting the range or domain of a relation. (The relational join operation, $\bullet$, is simply a different notation for composition: $A \bullet B \equiv B \circ A$.) In some cases, we manipulate sets describing iteration spaces, or relations giving the value-based dependences between two statements. Our experience with dataflow analysis indicates that we can generally create these sets and relations, and perform the above operations on them, efficiently [7].

In other cases, we discuss the manipulation of transitive dependences. The transitive value-based flow dependence from statement A to C describes all the ways in which information from an execution of A can reach an execution of C. For example, there are transitive dependences from the definition of `JPLUS` in Figure 1(b) to the definition of `WORK` in Figure 2, since `JPLUS` is used in the definition of `JP1`, which is used in the definition of `WORK`.

When there are no cycles of dependences [1], we can compute transitive dependences using composition. For example, if dependences run only from A to B and from B to C, we can compute the transitive dependence from A to C as $\delta_{A\rightarrow B} \bullet \delta_{B\rightarrow C}$.

If there is a dependence cycle, computing transitive dependences requires the transitive closure operation, defined as the infinite union

$$R^* \ = \ I \ \cup \ R \ \cup \ R \bullet R \ \cup \ R \bullet R \bullet R \ \ldots$$

[1] Note that each dependence must point forward in time, so there can be no cycle of dependences among iterations. Thus, we use the term "cycle of dependences" to refer to a cycle of dependences among statements – an iteration of statement B depends on some iteration of A, and an iteration of A on some iteration of B.

where I is the identity relation. For example, there is a dependence from the second statement of Figure 3 to itself, and the transitive closure of this self-dependence shows that every iteration is dependent on every earlier iteration. Transitive closure differs from the other operations we perform in that it cannot be computed exactly for arbitrary affine relations, and in that we do not have significant experience with its efficiency in practice. Furthermore, finding all transitive dependences for a group of N statements may take $O(N^3)$ transitive closure operations [11]. Thus, we provide alternative (less precise) algorithms in cases where our general algorithm would require transitive closure operations.

```
      JPLUS(1) = 2
      DO 10 J = 2, JMAX
         JPLUS(J) = JPLUS(J-1)+1
10    CONTINUE

      IF ( .NOT. PERIDC ) THEN
         JPLUS(JMAX) = JMAX
      ELSE
         JPLUS(JMAX) = 1
      ENDIF
```

Fig. 3. Alternative Definition of `JPLUS`, with Dependence Cycle

2.5 Generating Code from Sets of Iterations

Our optimizations make changes to a statement's set of iterations. For example, since the value defined in iteration `JMAX` of Statement S is never used (it must be killed by the conditional definition that follows it), we can eliminate iteration `JMAX`. That is, we can change the iteration space so that it ends at $JMAX - 1$ rather than $JMAX$:

$$iterations(S) := \{\ [J] \mid 1 \leq J \leq JMAX - 1\ \}.$$

In principle, we could remove loop iterations by adding `if` statements to the affected loop bodies, but this would introduce excessive overhead. Instead, we use the code generation facilities of the Omega Library to produce efficient code for such loop nests [12].

3 Dead Code Elimination

We generalize the technique of dead code elimination [13] by searching not for statements that produce a value that is never used, but for *iterations* that do so. We define as live any execution of a statement that is "obviously live" (i.e.,

produces a result of the procedure or may have a side effect) or may have dataflow to some live statement execution.

$$iterations(X) := obviously\_live(X) \bigcup_{\delta_{X \to Y}} domain(\delta_{X \to Y}) \quad (1)$$

For example, we can remove the last iteration of Statement S, as we saw in Section 2.5.

Our union operation is currently somewhat less effective at detecting redundancy than our subtraction operation. In practice, our code generation algorithms may work better if we reformulate Equation 1 to subtract, from the non-obviously-live iterations, any iterations that are not in the domain of any dataflow relation.

As we remove iterations, we must update the dataflow information involving that statement, possibly causing other iterations of other statements to become dead. However, we can't just to apply Equation 1 and adjust the dataflow information until we reached a fixed point. Our analysis eliminates executions of statements, not entire statements, so (in the absence of a bound on the number of loop iterations) an iterative application may not terminate.

When there is no cycle of dependences, we simply topologically sort the set of statements (ordering them such that all iterations of each statement depend only on prior statements), and remove dead iterations in a single traversal of this list (working from the results back to the start of the procedure). The above algorithm is equivalent to first finding all transitive value-based flow dependences to obviously live iterations, and then removing iterations that are not in the domain of these dependences. However, we expect that most code will not have many dead iterations, so finding all transitive dependences to perform dead code elimination would usually be a waste of time.

For code that does contain a dependence cycle (such as Figure 3), we produce a safe approximation of the sets of dead iterations by handling each dependence cycle as follows: If there are any dataflow edges from the cycle to any live iteration, or any obviously live iterations in the cycle, we mark all iterations of the cycle as live; otherwise, they must be dead. In principle, we could also use transitive dependence calculations within each cyclic component, but we fear the overhead of doing so would be prohibitive.

4 Constant Propagation

We generalize the notion of constant by defining as constant those statement executions for which the value produced is a function only of the surrounding loop indices and the set of symbolic constants. This includes traditional scalar constant expressions (which are constant functions of the loop indices), and all executions of all assignments in Figure 1, but not statements that read input.

We represent these constants with mappings from the symbolic constants and indices of the surrounding loops to the value computed. When these mappings are

affine, and the values produced are integral, these mappings can be manipulated by the Omega Library. For example, as we saw in Section 2.2,

$$value(S) = \{\ [J] \rightarrow [J+1] \mid 1 \leq J \leq JMAX\ \}.$$

Note that these relations may include conditions under which the values are defined, as in

$$value(T) = \{\ [] \rightarrow [JMAX] \mid \neg P\ \}.$$

(We have abbreviated $PERIDC$ as P to make the presentation that follows more concise).

We can extend this approach to handle a finite set of values of other types by keeping a table of constant values and storing a mapping from loop indices to indices into this table. For example, we indicate that the first definition of Figure 1(b) produces the value $\frac{-8.0}{3.0}$ by storing this real number constant at the next available index in our constant table (for example, 0), and generating a relation mapping to this index (such as $\{\ [] \rightarrow [0]\ \}$). We also flag such relations to ensure that we never treat the table index itself as the value of the expression.

4.1 Intra-procedural Constant Propagation

We initially identify all iterations of some statements as constant via a syntactic analysis: This step identifies as constant all iterations of any statement with a constant expression or an expression made up of constants and the loop indices. Note that we can safely treat as non-constant any constant function that is outside the scope of our expression evaluator. This step identifies all the assignments in both parts of Figure 1 as constant, and produces value relations such as the ones shown above. Note that, for code with no constants, our analysis stops at this step.

Next, we can identify as constant any executions of reads with incoming dataflow arcs from constant statement executions, such as all uses of `JPLUS` in the `stepfx` routine (shown in Figure 2). If Figure 2 immediately followed Figure 1(b), the read of `JPLUS(J)` in Statement V would receive dataflow from all three definitions of `JPLUS`:

$$\delta_{S \rightarrow V} = \{\ [J] \rightarrow [N, J] \mid 1, JLOW \leq J \leq JUP, JMAX - 1 \wedge 2 \leq N \leq 4\ \}$$
$$\delta_{T \rightarrow V} = \{\ [] \rightarrow [N, JMAX] \mid JLOW \leq JMAX \leq JUP \wedge 2 \leq N \leq 4 \wedge \neg P\ \}$$
$$\delta_{U \rightarrow V} = \{\ [] \rightarrow [N, JMAX] \mid JLOW \leq JMAX \leq JUP \wedge 2 \leq N \leq 4 \wedge P\ \}$$

For a use of a variable in a statement X, we generate a mapping from the iteration in which a read occurs to the constant value read, which we call $value\_in(X)$. This relation is the union over all incoming dataflow of the inverse of each dataflow relation with the value relation for each constant source statement:

$$value\_in(X) = \bigcup_{\delta_{Y \rightarrow X}} (\delta_{Y \rightarrow X})^{-1} \bullet value(Y) \tag{2}$$

In general, $value\_in(X)$ may be defined over a subset of $iterations(X)$, or be restricted by constraints on the symbolic constants. In our running example, the constant value read in V depends on `PERIDC`:

$$\begin{aligned} &value\_in(V) = \\ &\quad \{\ [N, J] \rightarrow [J+1] \mid 1, JLOW \leq J \leq JUP, JMAX - 1 \wedge 2 \leq N \leq 4\ \} \\ &\cup \{\ [N, JMAX] \rightarrow [JMAX] \mid JLOW \leq JMAX \leq JUP \wedge 2 \leq N \leq 4 \wedge \neg P\ \} \\ &\cup \{\ [N, JMAX] \rightarrow [1] \mid JLOW \leq JMAX \leq JUP \wedge 2 \leq N \leq 4 \wedge P\ \} \end{aligned}$$

Since Statement V simply assigns this value to $JP1$, $value(V) = value\_in(V)$.

```
      DO 10 J=1,JMAX-1
        JPLUS(J) = J+1
10    CONTINUE
      JPLUS(JMAX)=1

      DO 20 J=2,JMAX-1
        B(J) = 5*JPLUS(J+1) - 2*JPLUS(J-1) + J
20    CONTINUE
```

Fig. 4. Hypothetical Constant Definition of `B`

In general, a constant expression may combine several reads, constants, and loop index variables. In this case, we must produce a separate $value\_in(X_i)$ for each read i in statement X, and then combine these to produce $value(X)$ according to the expression computed by the statement. Consider what would happen in the second loop of Figure 4 (Statement R): The read `JPLUS(J+1)` (which we'll call R_1) has dataflow from the two assignments (Statements P and Q):

$$\begin{aligned} \delta_{P \rightarrow R_1} &= \{\ [J] \rightarrow [J-1] \mid 3 \leq J \leq JMAX - 1\ \} \\ \delta_{Q \rightarrow R_1} &= \{\ [] \rightarrow [JMAX - 1] \mid 3 \leq JMAX\ \} \end{aligned}$$

The read `JPLUS(J-1)` (R_2) has dataflow only from the definition in the loop:

$$\delta_{P \rightarrow R_2} = \{\ [J] \rightarrow [J+1] \mid 1 \leq J \leq JMAX - 2\ \}$$

The value relations are thus

$$\begin{aligned} value\_in(R_1) &= \{\ [J] \rightarrow [J+2] \mid 2 \leq J \leq JMAX - 2\ \} \\ &\quad \cup \{\ [JMAX - 1] \rightarrow [1] \mid 3 \leq JMAX\ \} \\ value\_in(R_2) &= \{\ [J] \rightarrow [J] \mid 3 \leq J \leq JMAX - 1\ \} \end{aligned}$$

We then combine the value relations for each read X_i to produce $value\_in(X)$. This combined relation is defined in terms of the value relations $value\_in(X_1), value\_in(X_2), ...$ as

$$value\_in(X) = \{[I_1, I_2, ...] \rightarrow [v_1, v_2, ...] \mid [I_1, I_2, ...] \rightarrow [v_i] \in value\_in(X_i)\} \quad (3)$$

In our example, $value\_in(R)$ maps from J to the values read in the expression, i.e. `JPLUS(J+1)`, `JPLUS(J-1)`, and `J`, as follows:

$$value\_in(R) = \{\ [J] \rightarrow [J+2, J, J] \mid 2 \leq J \leq JMAX - 2\ \} \\ \cup \{\ [JMAX-1] \rightarrow [1, JMAX-1, JMAX-1] \mid 3 \leq JMAX\ \}.$$

We then build, from the expression in statement X, a relation $expression(X)$ that maps from the values read to the value produced by the statement, e.g.

$$expression(R) = \{\ [R1, R2, R3] \rightarrow [5R1 - 2R2 + R3]\ \}$$

We can then define the mapping from iterations to constant values by applying the expression to the incoming values:

$$value(X) = value\_in(X) \bullet expression(X) \tag{4}$$

Note that this join of the incoming constant information $value\_in(X)$ with $expression(X)$ is a form of symbolic constant folding. In our example,

$$value(R) = \{\ [J] \rightarrow [4J + 10] \mid 2 \leq J \leq JMAX - 2\ \} \\ \cup \{\ [JMAX-1] \rightarrow [-JMAX + 6] \mid 3 \leq JMAX\ \}.$$

Note that we can safely use the degenerate case, $value(X) = \{\ [J] \rightarrow [V] \mid false\ \}$ (i.e. the empty set of constant iterations), for any case in which we are unable to detect or represent a constant (e.g. when an $expression(X)$ cannot be represented in our system, or when any $value\_in(X_i)$ is non-constant).

When there are no dependence cycles, we can apply Equations 2, 3, and 4 in a single pass through the topologically sorted statement graph that we used in Section 3. As was the case in dead-code elimination, dependence cycles (such as the one in Figure 3) can keep an iterative analysis from terminating. We could, in principle, produce approximate results via the transitive closure operation. In practice, we simply do not propagate constants within dependence cycles.

4.2 Interprocedural Constant Propagation

In the actual `ARC2D` benchmark, the code for Figures 1(b) and 2 are in separate procedures. However, we may not have interprocedural value-based flow dependence relations.

We therefore perform interprocedural constant propagation on a procedure-by-procedure basis. We start with a pass of inter-procedural data flow analysis in which no attempt is made to distinguish among array elements (and thus standard algorithms for scalars [13] may be employed). We generate a graph in which each procedure is a node, and there is an arc from $P1$ to $P2$ if $P2$ may read a value defined in $P1$. We break this graph into strongly connected components, and topologically sort the components. We process these components in order, starting with those that receive no data from any other component. Within each component, we process procedures in an arbitrary order.

Note that the above ordering may have little effect other than to identify one "initialization" procedure, and place it before all others in our analysis. We believe this will be sufficient for our purposes, however, as we expect that many constants will be defined in such routines.

Our restriction to a single analysis of each procedure could cause us to miss some constants that could, in principle, be identified. For example, assume Procedure $P1$ uses `A` and defines a constant array `B`, and Procedure $P2$ uses `B` and defines the constant array `A`. If one of these arrays cannot depend (even transitively) on the other, we could extract definition information from each procedure and then use it to optimize the other. However, without a study of how such cases arise in practice, it is not clear whether it is best to (a) develop a general algorithm to detect them (this would require interprocedural transitive dependence information, though not on an element-by-element basis), (b) detect simple cases (for example, if either `A` or `B` can be syntactically identified as a constant), or (c) ignore them.

For each procedure, we describe any constant values stored in each array with a mapping from subscripts to values stored in the array. These mappings can be generated by composing our iteration-space based information with relations from array subscripts to iterations (these are defined by the subscript expressions). For example, our intra-procedural analysis of Figure 1(b) produces the following mapping for `JPLUS`:

$$\begin{aligned} &\{\ [J] \rightarrow [J+1] \mid 1 \leq J \leq JMAX - 1\ \} \\ \cup &\{\ [JMAX] \rightarrow [JMAX] \mid P\ \} \\ \cup &\{\ [JMAX] \rightarrow [1] \mid \neg P\ \}. \end{aligned}$$

We then use each mapping in our analysis of each procedure that uses the corresponding array. When cycles in our graph of procedures prevent us from having information about all arrays used in a procedure, we can safely assume all elements of an array are non-constant.

For each use of a constant array, we compose our mapping from subscripts to values with the relation from loop iterations to array subscripts, producing a relation from iterations to values read. This is exactly the information we derived in the second step of Section 4.1, and we continue the analysis from that point.

4.3 Using Propagated Constants

In some cases, it may be profitable to insert the constants detected above into the expressions that use them; in other cases, the value of detecting the constants may lie in their ability to improve other forms of analysis.

For example, a computationally intensive loop in the `resid` subroutine multiplies each of four complicated expressions by a value defined in Figure 1(a). Substituting `0.0` for `A(1)` lets us eliminate one of these expressions, reducing inner loop computation by about 20% [8].

Our constant propagation can improve our analysis of the usage of the `XYJ` array in Figure 2. Without constant propagation, the the presence of `JP1` in

the subscript expression prevents us from describing the pattern of uses with affine terms. This forces our constraint solver to use algorithms that are more expensive and may produce approximate results, which may then interfere with other steps such as cache optimization. We can eliminate the non-affine term, and thus produce an exact description of the usage of `XYJ`, by substituting our constant value relation for `JP1` during the analysis of the `XYJ` array (we do not actually replace `JP1` in the program). Note that `stepfx` is one of the most computationally intensive routines in the `ARC2D` benchmark, taking about one third of the total run time [14], so optimization of these loops is critical for good performance.

Our ability to use information about the values of a conditional expression to eliminate some or all executions of the controlled statement corresponds to traditional unreachable code elimination.

5 Copy Propagation and General Forward Substitution

For the constant functions described in the previous section, it is always legal to insert a (possibly conditional) expression that selects the appropriate constant, even though it may not always be beneficial. Extending our system for more general forward substitution requires a system for determining whether the expression being substituted has the same value at the point of substitution, and a way of estimating the profitability of forward substitution. The latter requires a comparison of costs of evaluating the expression at the point of use vs. computing and storing the value and later fetching it from memory. The cost of the latter is hard to estimate, as subsequent locality optimization can greatly affect it. We have therefore not made a detailed exploration of general forward substitution.

```
for (int t = 0; t<T; t++)
{
  for (int j = 0; j<=N-1; j++)
    old[j] = cur[j];
  for (int i = 1; i<=N-2; i++)
    cur[i] = 0.25 *
      (old[i-1]+2*old[i]+old[i+1]);
}
```

```
for (int t = 0; t<T; t++)
{
  for (int j = 0; j<=N-1; j++)
    old[j] = cur[j];
  for (int i = 1; i<=N-2; i++)
    cur[i] = 0.25 *
      (old[i-1]+2*cur[i]+cur[i+1]);
}
```

Fig. 5. Three Point Stencil before and after Partial Copy Propagation

We have, however, made use of copy propagation as an integrated part of cache optimization [15, 16]. Copy propagation may simplify the task of cache optimization by eliminating a "copy" loop from an imperfect loop nest such as the stencil shown at the left of Figure 5. Such cases, in which the first of two loop nests is a simple array copy, can be detected syntactically. We then find the mapping from the subscript expressions of the destination array (`old`) to

those of the source array (`cur`), assuming these expressions are affine (if not, we simply do not perform copy propagation). We can then apply this mapping to the subscript expressions in each use of the destination array, to find equivalent subscripts for the original array in the new context. For example, we replace `old[i+1]` with `cur[i+1]` in Figure 5 by applying the subscript mapping of the copy statement (simply $[j] \rightarrow [j] \mid 0 \leq j \leq N-1$) to the subscript expression $(i+1)$.

Two conditions must be met for this substitution to be legal: First, the domain of the subscript mapping for the copy must contain the subscripts at the point of use (in our example, $[j] \mid 0 \leq j \leq N-1$ contains $[i+1] \mid 1 \leq i \leq N-2$). However, if this does not hold, we could perform copy propagation for only the iterations where it is true. For example, if the original program set `old[0]` and `old[N-1]` before the `t` loop and ran `i` from 1 to $N-2$, we could perform this copy propagation for all but the last iteration of the `j` loop. We could then propagate the value of `old[N-1]` separately for this last iteration.

Secondly, the substitution must not affect the array dataflow (as this could change the result of the calculation). We can test this by comparing the original dataflow to the dataflow that would occur from the right hand side of the copy to the potentially propagated array use. In our example, if we replace `old[i+1]` with `cur[i+1]`, and compute the flow of value that would occur if the copy loop had written `cur[j]` rather than read it, we get the original dataflow relation for the use of `old[i+1]`. The same is true for the replacement of `old[i]` with `cur[i]`, but this test fails for `old[i-1]`, since `cur[i-1]` would read the value from the previous iteration of the `i` loop, rather than the `j` loop nest. Thus, this form of copy propagation only lets us produce the code on the right of Figure 5.

```
for (int t = 0; t<T; t++)
  for (int i = 1; i<=N-2; i++)
    cur[t%2][i] = 0.25 *
      (cur[(t-1)%2][i-1]+2*cur[(t-1)%2][i]+cur[(t-1)%2][i+1]);
}
if (t%2 == 0)
  ... use cur[0][*]
else
  ... use cur[1][*]
```

Fig. 6. Three Point Stencil after Partial Expansion and Full Copy Propagation

To produce the full copy propagation shown in Figure 6, we must find a way to retain the original dataflow after the propagation. Since the corruption of the dataflow is purely the result of overwriting the value we need, there must be some memory transformation (such as renaming or full array expansion) that will produce this effect. Once all uses of the copy array have been eliminated, the copy loop itself can be removed, allowing the use of techniques that require

perfect loop nests (such as traditional skewing and tiling). Our full technique for producing a new memory mapping that is not prohibitively wasteful of memory (as full array expansion would be) is beyond the scope of this paper. However, even the simple expansion by a factor of two shown in Figure 6, followed by skewing and tiling, can more than double execution speed [15].

This copy propagation algorithm can also be used to extend our constant propagation system to handle cases in which the constant value is not an affine function of the loop indices (in this case, we do not have to worry about intervening writes to the (constant) array).

Full forward substitution of more complex array expressions is a straightforward extension of our copy propagation algorithm. It simply requires the calculation of new subscript expressions, and the elimination of any intervening writes, for each array used in the expression.

6 Related Work

Our algorithms are more general than traditional scalar optimizations in that they handle individual executions of statements and elements of arrays. However, they are less general in that they are "pessimistic" rather than "optimistic". As we have not found any cases in which optimistic analysis is needed for arrays, we suspect such a technique would be of purely academic interest.

There are a number of systems for performing analysis of the flow of values in array variables [1, 2, 3, 4, 5, 6, 7, 8]. In principle, the results of any of these systems could be used as the basis for extensions of traditional scalar dataflow-based optimizations. However, only the work of Sarkar and Knobe [8] addresses this possibility. Their array SSA form lets them represent a finite set of known constant values, and thus is unable to handle our example Figure 1(b) (in which the number of constant values is not known at compile time).

Kelly, Pugh, Rosser, and Shpeisman developed the transitive closure algorithm for the Omega Library [11]. They give a number of applications of this operation, including the calculation of transitive data dependences. They also show that transitive closure can be used to perform scalar induction variable detection or even "completely describe the effect of arbitrary programs" [11, Section3.4]. However, this is used to show "that transitive closure cannot always be computed exactly", and does not seem to be a serious proposal for induction variable detection. It is interesting, though not necessarily useful, to view the pessimistic and optimistic formulations of traditional constant propagation as increasingly accurate approximations of this transitive closure calculation (restricted to cases in which all iterations of each statement must be identified as either constant or potentially non-constant).

Many of our algorithms are simply efficient ways to approximate a result that could be achieved in a straightforward fashion from the set of all transitive dependences. However, Pugh et al. do not discuss the application of transitive closure for these optimizations, and it would almost certainly be prohibitively expensive to perform them by computing all transitive dependences [17].

7 Efficiency

While we do not have extensive experience applying these optimizations, our experience performing dependence analysis with the Omega Library indicates that the operations that we need to perform should run efficiently on the relations that we generate, as long as we can avoid transitive closure. For example, we used the "time" operation of the Omega Calculator (version 1.10) to measure the amount of time required for a Linux workstation with a 400 MHz Celeron processor to produce the value relation for the definition of `JP1`, as discussed at the start of Section 4.1. This combination of three inverse operations, three join operations, two union operations, and final simplification and redundancy checking took under a millisecond. (The exact time depends on how much simplification and redundancy checking is performed; it varied from 0.2ms to 0.6ms.) It may be the case that the transitive closure calculations would also run quickly, but our experience with this operation is much more limited. Sarkar and Knobe [8] do not give timing results for their analysis, so we are unable to compare our execution time to theirs.

8 Conclusions

The availability of information about the flow of values in array variables enables a variety of analysis and optimization techniques that had previously been applied only to scalars. These techniques are most effective when they are updated to work with individual executions of statements rather than treat each statement as an atomic object.

We have presented algorithms for dead-code elimination, constant propagation (which can be used for constant folding, detection of constant conditions, and unreachable code elimination), and forward substitution (including copy propagation), that are based on such definitions. Our algorithms are exact for code that is free of dependence cycles and has affine control flow and subscript expressions. For such code, our dead code elimination algorithm will eliminate any iterations that produce values that are not used, and our constant propagation algorithm will identify as constant any statement executions that produce values that we can represent as constant functions of the surrounding loop indices and symbolic constants. For code with dependence cycles, we provide approximate results, to avoid computing transitive dependences.

This work is supported by NSF grant CCR-9808694.

References

[1] Paul Feautrier. Dataflow analysis of scalar and array references. *International Journal of Parallel Programming*, 20(1):23–53, February 1991.

[2] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Data dependence and data-flow analysis of arrays. In *5th Workshop on Languages and Compilers for*

Parallel Computing (Yale University tech. report YALEU/DCS/RR-915), pages 283–292, August 1992.

[3] Peng Tu and David Padua. Array privatization for shared and distributed memory machines. In *Workshop on Languages, Compilers, and Run-Time Environments for Distributed Memory Multiprocessors*, September 1992.

[4] William Pugh and David Wonnacott. An exact method for analysis of value-based array data dependences. In *Proceedings of the 6th Annual Workshop on Programming Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, August 1993. Also CS-TR-3196, Dept. of Computer Science, University of Maryland, College Park.

[5] Junjie Gu, Zhiyuan Li, and Gyungho Lee. Symbolic array dataflow analysis for array privatization and program parallelization. In *Supercomputing '95*, San Diego, Ca, December 1995.

[6] Denis Barthou, Jean-François Collard, and Paul Feautrier. Fuzzy array dataflow analysis. *Journal of Parallel and Distributed Computing*, 40:210–226, 1997.

[7] William Pugh and David Wonnacott. Constraint-based array dependence analysis. *ACM Trans. on Programming Languages and Systems*, 20(3):635–678, May 1998. `http://www.acm.org/pubs/citations/journals/toplas/1998-20-3/p635-pugh/`.

[8] Vivek Sarkar and Kathleen Knobe. Enabling sparse constant propagation of array element via array SSA form. In *Static Analysis Symposium (SAS'98)*, 1998.

[9] M. Berry et al. The PERFECT Club benchmarks: Effective performance evaluation of supercomputers. *International Journal of Supercomputing Applications*, 3(3):5–40, March 1989.

[10] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. The Omega Library interface guide. Technical Report CS-TR-3445, Dept. of Computer Science, University of Maryland, College Park, March 1995. The Omega library is available from http://www.cs.umd.edu/projects/omega.

[11] Wayne Kelly, William Pugh, Evan Rosser, and Tatiana Shpeisman. Transitive closure of infinite graphs and its applications. *International J. of Parallel Programming*, 24(6):579–598, December 1996.

[12] Wayne Kelly, William Pugh, and Evan Rosser. Code generation for multiple mappings. In *The 5th Symposium on the Frontiers of Massively Parallel Computation*, pages 332–341, McLean, Virginia, February 1995.

[13] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, Inc., 1997.

[14] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the automatic parallelization of 4 Perfect benchmark programs. In *Proceedings of the 4th Workshop on Programming Languages and Compilers for Parallel Computing*, August 1991. Also Technical Report 1193, CSRD, Univ. of Illinois.

[15] David Wonnacott. Achieving scalable locality with time skewing. In preparation. A preprint is available as `http://www.haverford.edu/cmsc/davew/cache-opt/tskew.ps`, and parts of this work are included in Rutgers University CS Tech Reports 379 and 378., 2000.

[16] David Wonnacott. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *Proceedings of the 2000 International Parallel and Distributed Processing Symposium*, May 2000.

[17] Evan Rosser. Personal communication, March 2000.

Searching for the Best FFT Formulas with the SPL Compiler*

Jeremy Johnson[1], Robert W. Johnson[2], David A. Padua[3], and Jianxin Xiong[3]

[1] Drexel University, Philadelphia, PA 19104,
jjohnson@mcs.drexel.edu
[2] MathStar Inc., Minneapolis, MN 55402,
rwj@mathstar.com
[3] University of Illinois at Urbana-Champaign, Urbana, IL 61801,
{padua,jxiong}@cs.uiuc.edu

Abstract. This paper discuss an approach to implementing and optimizing fast signal transforms based on a domain-specific computer language, called SPL. SPL programs, which are essentially mathematical formulas, represent matrix factorizations, which provide fast algorithms for computing many important signal transforms. A special purpose compiler translates SPL programs into efficient FORTRAN programs. Since there are many formulas for a given transform, a fast implementation can be obtained by generating alternative formulas and searching for the one with the fastest execution time. This paper presents an application of this methodology to the implementation of the FFT.

1 Introduction

This paper discusses an approach to implementing and optimizing fast signal transforms, such as the fast Fourier transform (FFT), which can be expressed as matrix factorizations. The approach relies on a domain-specific computer language, called SPL, for expressing matrix factorizations, and a special purpose compiler which translates matrix factorizations expressed in SPL into efficient FORTRAN programs. The factorization of a matrix into a product of sparse structured matrices leads to a fast algorithm to multiply the matrix with an arbitrary input vector, and the SPL compiler translates such a factorization into an efficient program to perform the matrix-vector product. Since SPL programs are mathematical formulas involving matrices, it is possible to transform SPL programs into equivalent programs through the use of theorems from linear algebra. Moreover, given a matrix, it is possible, through the application of various mathematical transformations, to automatically generate a large number of mathematically equivalent SPL programs applying the given matrix to a vector. This allows us, through the use of the SPL compiler, to systematically search for the SPL program with the fastest execution time. When the matrix represents

* This work was partially supported by DARPA through research grant DABT63-98-1-0004 administered by the Army Directorate of Contracting.

S.P. Midkiff et al. (Eds.): LCPC 2000, LNCS 2017, pp. 112–126, 2001.

a signal transform, such as the FFT, this procedure allows us to find a fast implementation of the signal transform. The only difficulty with this approach is that the size of the search space grows exponentially and therefore more refined search techniques than exhaustive search must be utilized. This paper presents an application of this methodology to the implementation and optimization of the FFT. Various techniques are presented that allow us to find nearly optimal SPL programs while only searching through a small subspace of the allowed programs.

The approach presented in this paper was first outlined in [1] and further discussed in [7]. Our work is part of a larger effort, called SPIRAL [11], for automatically implementing and optimizing signal processing algorithms. This paper presents the first efficient implementation of the SPL compiler and hence this first true test of the methodology.

Our approach is similar to that used by FFTW [4, 3], however, FFTW is specific to the FFT, while our use of SPL allows us to implement and optimize a far more general set of programs. FFTW also utilizes a special purpose compiler and searches for efficient implementations. However, their compiler is based on built-in code sequences for a fixed set of FFT algorithms and is only used for generating small modules of straight-line code, called codelets. Larger FFTs are performed using a recursive program which utilizes codelets of different sizes in the base case. A search over a subset of recursive FFT algorithms is performed, using dynamic programming, to find the recursive decomposition and set of codelets with the best performance. Our approach, even when restricted to the FFT, considers a much larger search space and uses search techniques other than dynamic programming. Similar to the work in FFTW, is the package described in [8] for computing the Walsh-Hadamard transform (WHT). This work is closer in spirit to the work in this paper, in that the different algorithms considered are expressed mathematically and the search is carried out over a space of formulas. Similar to FFTW, the WHT package is restricted to a specific transform, and a code generator restricted to the WHT is used rather than a compiler. Finally, in addition to searching over a space of mathematical formulas, this paper also considers a search over a space of compiler options and techniques. In particular, we allow compilation of SPL programs to perform various amounts of loop unrolling, and include in our search different amounts of unrolling.

Related to our approach is the ATLAS project, which uses empirical search techniques to automatically tune high performance implementation of the Basic Linear Algebra Subroutines (BLAS) [15]. The application of search techniques in determining appropriate compiler optimizations has been used in several efforts. Kisuki and Knijnenberg [9] presented similar ideas using the term "iterative compilation". They experimented with loop tiling, loop unrolling and array padding with different parameters, as well as different search algorithms. Their results appeared to be good compared with static selection algorithms. One difference between their work and ours is that our search is carried on both different input programs and different compilation parameters, thus provides more flexibility. Massalin [10] presented a "superoptmizer" which tries to find the shortest ma-

chine code sequence to compute a function. They carried out an exhaustive search within the space of a subset of the machine instruction set to ensure that the final result is optimal. Since the size of their search space increases exponentially, the method only works for very short programs.

In the remainder of the paper we review the SPL language, the SPL compiler, SPL formula generation, and apply search techniques to systematically search for the optimal FFT implementation.

2 Matrix Factorizations and SPL

Many digital signal processing (DSP) transforms are mathematically given as a multiplication of a matrix M (the transform) with a vector x (the sampled signal). Examples are the discrete Fourier transform (DFT), trigonometric transforms, and the Hartley and Haar transforms, [12]. A fast algorithm for these transforms can be given by a factorization of M into a product of sparse matrices, [12, 14]. For example, the well-known Cooley/Tukey algorithm, [2], also known as fast Fourier transform (FFT), for computing an rs-point DFT, F_{rs}, can be written as

$$\mathrm{F}_{rs} = (\mathrm{F}_r \otimes \mathrm{I}_s)T(\mathrm{I}_r \otimes \mathrm{F}_s)L \tag{1}$$

where $\otimes$, denotes the tensor product, I_s is the identity matrix, T a diagonal matrix, and L a permutation matrix both depending on r and s.

SPL [6] is a domain-specific programming language for expressing and implementing matrix factorizations. It was originally developed to investigate and automate the implementation of FFT algorithms [1]. As such, some of the built-in matrices are biased towards the expression of FFT algorithms, however, it is important to note that SPL is not restricted to the FFT. It contains features that allow the user to introduce notation suitable to any class of matrix expressions.

This section reviews some notation for expressing matrix factorizations and summarizes the SPL language.

2.1 Matrix Factorizations

In this section we show how a matrix factorization of the n-point DFT matrix, F_n, leads to a program to compute the matrix-vector product of F_n applied to a vector.

The n-point Fourier Transform is the linear computation

$$y(l) = \sum_{k=0}^{n-1} \omega_n^{lk} x(k) \qquad 0 \le l < n \tag{2}$$

where $\omega_n = e^{-2\pi i/n}$. Computation of the Fourier transform can be represented by the matrix-vector product $y = F_n x$, where F_n is the $n \times n$ matrix $[\omega_n^{lk}]_{0 \le l,k < n}$.

Computing the n-point Fourier transform as a matrix-vector product requires $O(n^2)$ operations. A faster algorithm can be obtained by factoring F_n into a product of simpler matrices.

Theorem 1. *Let $N = 2m$. Then*

$$F_N L_m^N = \begin{bmatrix} F_m & W_m F_m \\ F_m & -W_m F_m \end{bmatrix}, \tag{3}$$

where $W_m = diag(1, \omega_N, \ldots, \omega_N^{m-1})$ and L_m^N is the permutation that rearranges the elements of a vector into m segments containing elements of the input separated by a stride of m and with the i-th segment starting with the i-th element of the input.

This block decomposition can be written compactly using the tensor product (also called Kronecker product). Let A be an $m \times m$ matrix and B a $n \times n$ matrix then the tensor product of A and B, $A \otimes B$, is the $mn \times mn$ matrix defined by the block matrix product

$$A \otimes B = [a_{ij}B]_{1 \le i,j \le m} = \begin{bmatrix} a_{1,1}B & \cdots & a_{1,m}B \\ \vdots & \ddots & \vdots \\ a_{m,1}B & \ldots & a_{m,m}B \end{bmatrix}. \tag{4}$$

Corollary 1.

$$F_{2m} = (F_2 \otimes I_m) T_m^{2m} (I_2 \otimes F_m) L_2^{2m}, \tag{5}$$

where $T_m^{2m} = diag(I_m, W_m)$.

More generally,

Theorem 2 (Cooley-Tukey).

$$F_{rs} = (F_r \otimes I_s) T_s^{rs} (I_r \otimes F_s) L_r^{rs}, \tag{6}$$

where L_r^{rs} is a permutation matrix and T_s^{rs} is a diagonal matrix. Let e_i^n, $0 \le i < n$, be a vector of size n with a one in the i-th location and zeroes elsewhere, and let ω_{rs} be a primitive rs-th root of unity. Then $T_s^{rs}(e_i^r \otimes e_j^s) = \omega_{rs}^{ij}(e_i^r \otimes e_j^s)$ and $L_r^{rs}(e_i^r \otimes e_j^s) = (e_j^s \otimes e_i^r)$.

For example, the 4-point DFT can be factored like the following:

$$F_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & i \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

An algorithm to compute $y = F_N x$, where $N = 2m$, can be obtained using the previous factorization. The output y is obtained using the factorization $F_N = (F_2 \otimes I_m) T_m^N (I_2 \otimes F_m) L_2^N$ by first applying L_2^N to the input x, then applying $(I_2 \otimes F_m)$ to the resulting vector $L_2^N x$, and so on.

2.2 SPL

A SPL program is essentially a sequence of mathematical formulas built up from a parameterized set of special matrices and algebraic operators such as matrix composition, direct sum, and the tensor product. The SPL language uses a prefix notation similar to lisp to represent formulas. For example, the expressions `(compose A B)` and `(tensor A B)` correspond to the matrix product $A \cdot B$ and the tensor product $A \otimes B$ respectively. The language also includes special symbols such as `(F n)` and `(I m)` to represent the discrete Fourier transform matrix F_n and the identity matrix I_m. In addition to the built-in symbols the user can assign a SPL formula to a symbol to be used in other expressions. Moreover, new parameterized symbols can be defined so that SPL programs can refer to other sets of parameterized matrices.

For example, the Cooley-Tukey factorization of F_{rs} for any integer values r and s is represented by the SPL expression

```
(compose (tensor (F r)(I s))(T rs s)(tensor (I r)(F s))(L rs r))
```

Note that all SPL programs represent linear computations of a fixed size. Expressions with parameters such as the one above are used to instantiate families of SPL programs.

The power of SPL for expressing alternative algorithms is illustrated by the following list of formulas corresponding to different variants of the FFT, [5, 13, 14]. Each of the formulas was obtained using the Cooley-Tukey factorization and elementary properties of the tensor product. In the following formulas, the symbol R_{2^k} denotes the k-bit bit-reversal permutation.

Apply Cooley/Tukey inductively

$$F_8 = (F_2 \otimes I_4)T_4^8(I_2 \otimes F_4)L_2^8 \tag{7}$$

Recursive FFT

$$F_8 = (F_2 \otimes I_4)T_4^8(I_2 \otimes ((F_2 \otimes I_2)T_2^4(I_2 \otimes F_2)L_2^4))L_2^8 \tag{8}$$

Iterative FFT (Cooley/Tukey)

$$F_8 = (F_2 \otimes I_4)T_4^8(I_2 \otimes F_2 \otimes I_2)(I_2 \otimes T_2^4)(I_4 \otimes F_2)R_8 \tag{9}$$

Vector FFT (Stockham)

$$F_8 = (F_2 \otimes I_4)T_4^8L_2^8(F_2 \otimes I_4)(T_2^4 \otimes I_2)(L_2^4 \otimes I_2)(F_2 \otimes I_4) \tag{10}$$

Vector FFT (Korn/Lambiotte)

$$F_8 = (F_2 \otimes I_4)T_4^8L_2^8(F_2 \otimes I_4)(T_2^4 \otimes I_2)L_2^8(F_2 \otimes I_4)L_2^8R_8 \tag{11}$$

Parallel FFT (Pease)

$$F_8 = L_2^8(I_4 \otimes F_2)L_4^8T_4^8L_2^8L_2^8(I_4 \otimes F_2)L_4^8(T_2^4 \otimes I_2)L_2^8L_2^8(I_4 \otimes F_2)R_8 \tag{12}$$

These formulas are easily translated into SPL programs. The following SPL program corresponds to the formula for the iterative FFT on 8 points.

```
(define R8 (permutation (0 4 2 6 1 5 3 7)))
(compose (tensor (F 2) (I 4)) (T 8 4)
         (tensor (I 2) (F 2) (I 2)) (tensor (I 2) (T 4 2))
         (tensor (I 4) (F 2)) R8)
```

In general, a SPL program consists of the following constructs. This list refers to SPL 3.01 (see [6] for future enhancements and the most current version).

```
(1) matrix operations
        (tensor formula formula ...)
        (compose formula formula ...)
        (direct_sum formula formula ...)
(2) direct matrix description:
        (matrix (a11 a12 ...) (a21 a22 ...) ... )
        (diagonal (d1 d2 ...))
        (permutation (p1 p2 ...))
(3) parameterized matrices:
        (I n)
        (F n)
        (T mn n) or (T mn n, start:step:end)
        (L mn n)
(4) parameterized scalars:
        W(m n)       ; e^(2*pi*i*n/m)
        WR(m n)      ; real part of W(m n)
        WI(m n)      ; imaginary part of W(m n)
        C(m n)       ; cos(pi*n/m)
        S(m n)       ; sin(pi*n/m)
(5) symbol definition:
        (define name formula)
        (template formula (i-code-list))
```

Templates are used to define new parameterized matrices and by the SPL compiler to determine the code for different formulas in SPL. Templates are defined using an language independent syntax for code called i-code.

3 SPL Compiler

The execution of the SPL compiler consists of six stages: (1) parsing, (2) semantic binding, (3) type control, (4) optimization, (5) scheduling, and (6) code generation. The parser builds an abstract syntax tree (AST), which is then converted to intermediate code (i-code) using templates to define the semantics of different SPL expressions. The i-code is expanded to produce type dependent code (e.g. double precision real or complex) and loops are unrolled depending on compiler parameters. After intermediate code is generated, various optimizations such as constant folding, copy propagation, common sub-expression elimination, and algebraic simplification are performed. Optionally, data dependence analysis

is used to rearrange the code to improve locality. Finally, the intermediate code is converted to FORTRAN, leaving machine dependent compilation stages (in particular register allocation and instruction scheduling) to a standard compiler. (The code generator could easily be modified to produce C code or even native assembly code directly instead for FORTRAN.) Different options to the compiler, command-line flags or compiler directives, control various aspects of the compiler, such as the data types, whether loops are unrolled, and whether the optional scheduler is used. Some compiler optimizations and instruction scheduling can also be obtained at the SPL level, by transforming the input formulas.

The nodes in the AST corresponding to a SPL formula can be interpreted as code segments for computing a specific matrix-vector product. Code is generated by combining and manipulating these code segments, to obtain a program for the matrix-vector product specified by the SPL program. For example, a composition $A \cdot B$ is compiled into the sequence $t = B \cdot x; y = A \cdot t$ mapping input vector x into output vector y using the intermediate vector t. In the same way, the direct sum compiles into operations acting on parts of the input signal in parallel. The tensor product of code sequences for computing A and B can be obtained using the equation $A \otimes B = L^{mn}_m (I_n \otimes A) L^{mn}_n (I_m \otimes B)$.

For example, the code produced for the SPL program corresponding to the 4-point Cooley/Tukey algorithm is

```
subroutine F4(y,x)
implicit complex*16(f)
implicit integer(r)
complex*16 y(4),x(4)
f0   =  x(1) + x(3)
f1   =  x(1) - x(3)
f2   =  x(2) + x(4)
f3   =  x(2) - x(4)
f4   = (0d0,-1d0)*f3
y(1) =  f0 + f2
y(3) =  f0 - f2
y(2) =  f1 + f4
y(4) =  f1 - f4
end
```

In this example, two compiler directives were added: one giving the name `F4` to the subroutine and one causing complex arithmetic to be used. Looking at this example, one already sees several optimizations that the compiler makes (e.g. multiplications by 1 and -1 are removed). More significantly, multiplication by the permutation matrix L^4_2 is performed as re-addressing in the array accesses. Another important point is that scalar variables were used for temporaries rather than array elements. This has significant consequences on the FORTRAN compiler's effectiveness at register allocation and instruction scheduling.

Changing the code type to real, `#codetype real`, breaks up complex numbers into real and imaginary parts which gives the chance for further optimizations. In the case above the (complex) multiplication vanishes.

```
subroutine F4(y,x)
implicit real*8(r)
real*8 y(8),x(8)
r0   =  x(1) + x(5)
r1   =  x(2) + x(6)
r7   =  x(4) - x(8)
y(1) =  r0 + r4
y(2) =  r1 + r5
y(5) =  r0 - r4
y(6) =  r1 - r5
```

```
r2   =  x(1) - x(5)              y(3) =  r2 + r7
r3   =  x(2) - x(6)              y(4) =  r3 - r6
r4   =  x(3) + x(7)              y(7) =  r2 - r7
r5   =  x(4) + x(8)              y(8) =  r3 + r6
r6   =  x(3) - x(7)              end
```

In the previous example, we produced straight-line code. The SPL compiler is also capable of producing code with loops. For example, the formula $I_n \otimes A$ has a straight-forward interpretation as a loop with n iterations, where each iteration applies A to a segment of the input vector. The SPL compiler is instructed to generate code using this interpretation by the following template in which `ANY` matches any integer and `any` matches any SPL expression.

```
; (tensor Im An) parameters:
; p.0: mn, p.1:size(I)=m, p.2:ANY=m, p.3:size(A)=n, p.4:A
(template (tensor (I ANY) any)
  (r.0 = p.3-1
   do p.1
     y(0:1:r.0 p.3) = call p.4(x(0:1:r.0 p.3))
   end))
```

The following example shows how to use the SPL compiler to combine straight-line code with loops using formula manipulation and loop unrolling (loop unrolling is controlled by the compiler directive `#unroll` or by specifying a command line option `-Bn` which indicates that code for matrices of dimension n or less are to be unrolled). Using a simple property of the tensor product, $I_{64} \otimes F_2 = I_{32} \otimes (I_2 \otimes F_2)$. With the option `-R -l-B4`, the SPL compiler produces straight-line code for loop body which computes $(I_2 \otimes F_2)$.

```
subroutine I64F2(y,x)
implicit real*8(f)
implicit integer(r)
real*8 y(128),x(128)
do i0 = 0, 31
    y(4*i0+1) =  x(4*i0+1) + x(4*i0+2)
    y(4*i0+2) =  x(4*i0+1) - x(4*i0+2)
    y(4*i0+3) =  x(4*i0+3) + x(4*i0+4)
    y(4*i0+4) =  x(4*i0+3) - x(4*i0+4)
end do
```

SPL clearly provides a convenient way of expressing and implementing matrix factorizations; however, it can only be considered as a serious programming tool, if the generated code is competitive with the best code available. One strength of the SPL compiler is its ability to produce long sequences of straight-line code. In order to obtain maximal efficiency small signal transforms should be implemented with straight-line code thus avoiding the overhead of loop control or recursion. Table 1 compares code for small FFTs generated by the SPL compiler to the FFT codelets of FFTW on a Sun Ultra 5, 333 MHz.

n	1	2	3	4	5	6
FFTW	0.04	0.07	0.23	0.53	1.50	4.34
SPL FFT	0.03	0.05	0.22	0.50	1.45	4.29

Table 1. Runtime of FFT(2^n) in μs, FFTW vs. SPL

4 SPL Formula Generator

SPL programs can be derived by applying various mathematical theorems such as the Cooley-Tukey theorem. Such mathematical theorems can be thought of as SPL program transformations. For a given matrix, it is possible to generate many different SPL programs for performing the same mathematical computation. While mathematically equivalent, different SPL programs may, when translated by the SPL compiler, lead to FORTRAN programs with vastly different performance. Optimizing a given linear computation can be thought of as a search problem. Simply generate all possible SPL programs, compile them, and select the one which leads to the fastest execution time. By "all" possible algorithms, we mean all of those SPL programs that can be generated by applying a fixed set of mathematical rewrite rules.

In this section we consider a generalization of the Cooley-Tukey theorem which allows us to generate many different FFT algorithms. This space of FFT algorithms allows us to explore various combinations of iterative and recursive FFT algorithms.

The following theorem is an easy corollary of the Cooley-Tukey theorem.

Theorem 3. *Let $N = 2^n$ and let $N = N_1 \cdots N_t$, with $N_i = 2^{n_i}$, be a factorization of N. Using the notation $N(i) = N_1 \cdots N_i$, $\bar{N}(i) = N/N(i)$, $N(0) = 1$, and $N(t) = N$, Then*

$$F_N = \prod_{i=t}^{1} \left\{ (I_{N(i-1)} \otimes F_{N_i} \otimes I_{\bar{N}(i)})(I_{N(i-1)} \otimes T^{\bar{N}(i-1)}_{\bar{N}(i)}) \right\} R \tag{13}$$

where

$$R = \prod_{i=1}^{t} (I_{N(i-1)} \otimes L^{\bar{N}(i-1)}_{N_i}) \tag{14}$$

is a generalization of the bit-reversal permutation.

Applying this factorization recursively to each F_{N_i} until the base case equal to F_2 leads to an algorithm for computing F_N. The special case when $t = 2$ and $N_1 = 2$, and $N_2 = N/2$ leads to the standard recursive FFT. Other recursive breakdown strategies are obtained by choosing different values for N_1 and N_2. The special case when $N_1 = \cdots N_t = 2$ leads to the standard iterative, radix two, FFT. The space of formulas we consider allows us to explore various amounts of recursion and iteration along with different breakdown strategies.

Corresponding to this factorization is a partition of the integer $n = n_1 + \cdots + n_t$. A similar partition is obtained for each n_i, and the resulting algorithm can be represented as a tree, whose nodes are labeled by the integers n_i. The tree, called a partition tree, has the property that the value at any node is equal to the sum of the values of its children. Partition trees were introduced in [8].

The number of trees, T_n, with root equal to k is given by the recurrence

$$T_n = \begin{cases} 1 & n = 1 \\ \prod_{n=n_1+\cdots+n_t} T_{n_1} \cdots T_{n_t} & n > 1 \end{cases} \tag{15}$$

The solution to this recurrence is $\Theta(\alpha^n/n^{3/2})$, where $\alpha = 3+2\sqrt{2} \approx 5.83$. Table 2 lists the initial values for T_n.

n	# of formulas	n	# of formulas	n	# of formulas	n	# of formulas
1	1	5	45	9	20793	13	13649969
2	1	6	197	10	103049	14	71039373
3	3	7	903	11	518859	15	372693519
4	11	8	4279	12	2646723	16	1968801519

Table 2. Size of the formula space

Figure 1 shows the distribution of runtime (in microseconds) for all 20793 formulas for F_{2^9}. All runtimes presented in this paper were obtained on an Ultra 5 workstation, with a 333MHz UltraIIi CPU and 128MB memory. The software environment was Solaris 7, SUN Fortran 77 4.0, and SPL Compiler 3.04. The left side of Figure 1 plots performance in microseconds versus formula and the right side shows a histogram the distribution of runtimes. Each bin shows the number of formulas with runtimes in a given range. The fastest formula is about five times faster than the slowest one and 2.25 times faster than the mean. Also observe that only 1.7% of the total formulas, approximately 500 formulas, fall into the bin with the highest performance.

These plots show that there is a wide range in runtime, and that a significant gain in performance can be obtained by searching for the formula with the least execution time. The only difficulty in this approach to optimization, is that the number of formulas considered grows exponentially and hence an exhaustive search is not feasible for large values of n. In the next section we will present some techniques for reducing the search time.

5 Search

In this section we search for the optimal FFT implementation using the SPL compiler and the FFT formulas generated in the previous section. Several search strategies are applied in an effort to reduce the number of formulas considered,

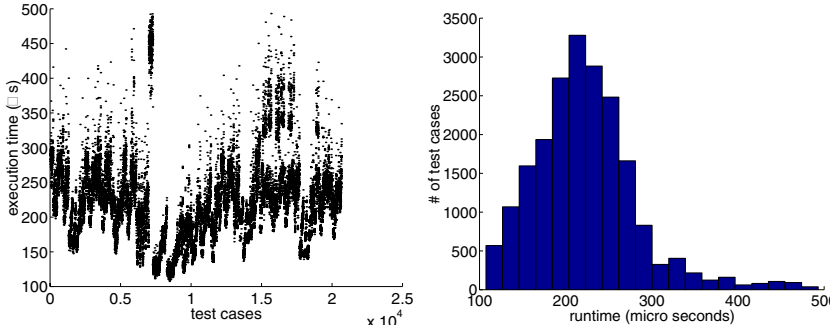


Fig. 1. Execution time(μs) and distribution, n=9, Exhaustive

while still obtaining a nearly optimal implementation. In addition to searching over the space of FFT formulas presented, we also search over several compiler strategies related to loop unrolling.

For each formula, there is a tradeoff between generating straight line code and loop code. Straight line code involves fewer instructions than loop code but its size increases with N. This leads to performance degradation due to cache and memory limitations. Except for small sized problems, partial unrolling is probably the best choice. The SPL compiler can generate several versions of code with different degrees of loop unrolling. The controlling parameter is called the *unrolling threshold*, which is denoted as T_u in this paper. Given a formula, the code generation is carried out top-down over the tree. If the size of a node is less than or equal to T_u, then the entire subtree rooted at this node is implemented by straight-line code; otherwise, the code corresponding to this node will be implemented as a loop.

It is impractical to try all the possible values of T_u. Preliminary experiments show that both compile time and execution time increase dramatically when T_u is greater than 7 on the Ultra 5 workstation. This value could be different on other machines or for other problems, but we believe that the upper bound of the value of interest can always be identified and is much smaller than N when N is large enough. Therefore, in our experiment, only values of $T_u \leq 7$ are considered. The entire search space is the Cartesian product of the formula space and the set of values of T_u.

Since exhaustive search is impractical it is necessary to use a search strategy that enables the optimal formula, or a nearly optimal formula, to be found without considering such a large space of formulas. As a first step one might randomly choose a subset of formulas to consider. However, Fig. 1 suggests that this approach is unlikely to find an optimal formula.

Instead we experimented with several versions of heuristic search based on local transformations of formulas. The first approach, called *single point search* (SPS), starts with a random formula and repeatedly finds a neighboring formula

while checking to see if the neighbor leads to better performance. The second approach, called *multiple point search* (MPS), starts with a smaller set of formulas (in our case 4) and searches locally in a neighborhood of each formula in the initial set.

Both approaches rely on a function *nexttree* to find a "neighbor" to an existing formula. The rules for finding a "neighbor" in SPS are shown in Fig.2. The rules used by MPS are the same, except $R7(Random)$ is not used. The "neighbors" are similar in their tree structure and, as a result, their performance values tend to be close (except when $R7$ is applied).

```
R1 (Merge):      merge two subtrees;
R2 (Split):      split one node into two;
R3 (Resize):     resize two subtrees by one;
R4 (Swap):       swap two subtrees;
R5 (Unrollmore): increase the unrolling threshold by one;
R6 (Unrollless): decrease the unrolling threshold by one;
R7 (Random):     randomly choose another tree;
R8 (Chgchild):   apply these rules to a subtree;
```

Fig. 2. Rules for finding a neighbor

The two approaches were compared to random search and exhaustive search when it was feasible. Data is presented for experiments with $n = 9$ and $n = 16$. The results for other sizes have similar characteristics.

The first observation is that different search strategies behave differently. Figures 3 to 5 show the execution time plotted against formulas considered and the distribution of runtimes (number of formulas with a given runtime) for random search, SPS, and MPS when $n = 9$. For random search, the execution time graph shows a random pattern and the distribution looks like a normal distribution. For MPS, the execution time converges to a small range near the best and the distribution concentrates on the fast side. It shows that the definition of "neighbor" in MPS is effective in controlling the range of the search. For SPS, the run-time also shows some kind of random pattern. However, from the distribution we can see, the run-times are more concentrated in the fast side. The random pattern is due to the use of R7(Random) in finding neighbors.

The second important observation is that a nearly optimal formula can be found while considering substantially fewer formulas than the total size of the search space. Recall that the size of the search space is $O(5.83^n)$. The number of samples used in SPS and MPS was $O(n^2)$. Figure 6 shows at each step the best performance obtained since the beginning of the test. This illustrates how quickly the search methods converge on a formula with good performance.

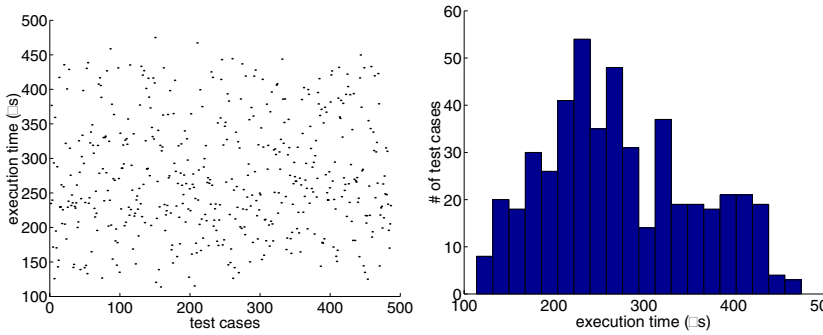


Fig. 3. Execution time(μs) and distribution, n=9, Random

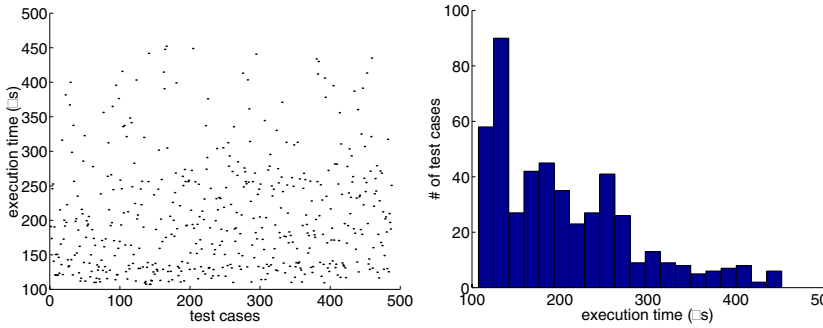


Fig. 4. Execution time(μs) and distribution, n=9, SPS

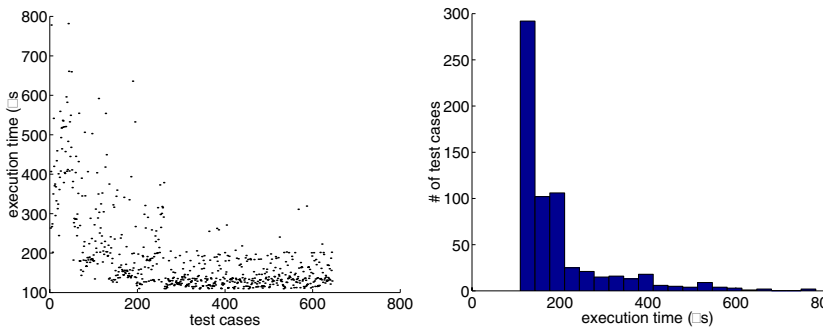


Fig. 5. Execution time(μs) and distribution, n=9, MPS

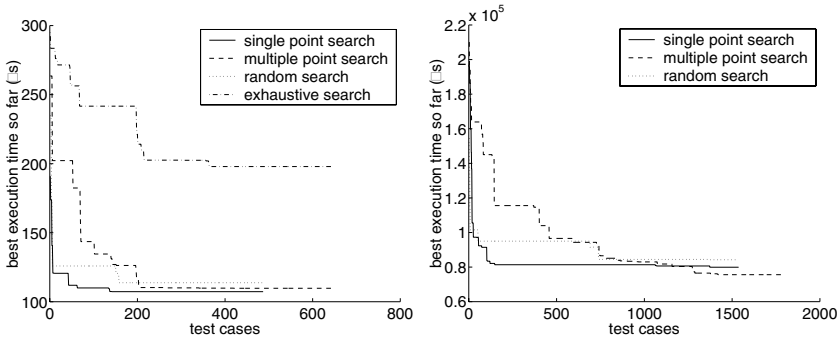


Fig. 6. Best execution time since the beginning (μs), n=9(left), 16(right)

	SPS		MPS		Random		Exhaustive	
n	steps	time(μs)	steps	time(μs)	steps	time(μs)	steps	time(μs)
2	1	0.08	1	0.08	1	0.08	1	0.08
3	1	0.22	7	0.22	4	0.22	1	0.22
4	10	0.51	25	0.51	32	0.51	4	0.51
5	31	1.63	50	1.63	103	1.67	12	1.64
6	34	4.75	112	4.76	6	5.11	68	4.65
7	25	14.63	128	14.62	216	14.80	364	14.57
8	59	40.65	261	40.57	11	66.93	1604	39.43
9	142	107.30	533	109.76	158	113.79	8459	107.46
10	163	264.46	585	265.15	181	299.76		
11	286	604.35	380	586.11	96	633.96		
12	292	1443.50	453	1601.88	269	1602.99		
13	443	3007.30	599	3222.55	795	3418.46		
14	530	8567.36	455	8411.40	1038	9758.44		
15	625	28892.56	272	30054.08	496	31583.56		
16	1536	79886.40	1420	75601.14	1361	84241.70		

Table 3. The best formulas found

Table 3 shows the performance of the best formula found by each algorithm and the number of steps when it was found. The best formulas found by SPS and MPS are very close to the best one found in the exhaustive search (we only verified this for $n \leq 9$). For the random search, although a pretty good formula can be found quickly, the performance of the best formula found is not as good as those found by SPS and MPS. The difference gets larger when n increases.

These experiments show that search techniques can be used to obtain fast implementations of signal processing algorithms like the FFT. The mathematical approach presented in the paper sets the framework for a systematic search, and

while an exhaustive search is infeasible, simple heuristic approaches allow nearly optimal implementations to be found in a reasonable amount of time.

Acknowledgements

The authors would like to thank the referees for their comments and suggestions.

References

[1] L. Auslander, J. R. Johnson, and R. W. Johnson. Automatic implementation of FFT algorithms. Technical Report 96-01, Dept. of Math. and Computer Science, Drexel University, Philadelphia, PA, June 1996. Presented at the DARPA ACMP PI meeting.

[2] J. W. Cooley and J. W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Math. of Computation*, 19:297–301, 1965.

[3] M. Frigo. A fast fourier transform compiler. In *PLDI '99*, pages 169–180, 1999.

[4] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *ICASSP '98*, volume 3, pages 1381–1384, 1998. `http://www.fftw.org`.

[5] Johnson J., Johnson R., D Rodriguez, and R. Tolimieri. A Methodology for Designing, Modifying, and Implementing Fourier Transform Algorithms on Various Architectures . *IEEE Trans. Circuits Sys.*, 9, 1990.

[6] J. Johnson, R. Johnson, D. Padua, and J. Xiong. SPL: Signal Processing Language, 1999. `http://www.ece.cmu.edu/~spiral/SPL.html`.

[7] J. R. Johnson and R. W. Johnson. Automatic generation and implementation of FFT algorithms. In *SIAM Conference on Parallel Processing for Scientific Computing*, March 1999.

[8] J. R. Johnson and M. Püschel. In search of the optimal Walsh-Hadamard transform. In *Proc. ICASSP 2000*, 2000.

[9] T. Kisuki, P.M.W.Knijnenberg, M.F.P.O'Boyle, and H.A.G.Wijshoff. Iterative compilation in program optimization. In *Proc. CPC2000*, pages 35–44, 2000.

[10] H. Massalin. Superoptimizer – a look at the smallest program. In *Proc. ASPLOS II*, pages 122–126, 1987.

[11] J. M. F. Moura, J. Johnson, R. Johnson, D. Padua, V. Prasanna, and M. M. Veloso. SPIRAL: Portable Library of Optimized SP Algorithms, 1998. `http://www.ece.cmu.edu/~spiral/`.

[12] K. R. Rao and P. Yip. *Discrete Cosine Transform*. Academic Press, 1990.

[13] R. Tolimieri, M. An, and C. Lu. *Algorithms for Discrete Fourier Transforms and Convolution*. Springer, 2nd edition, 1997.

[14] C. Van Loan. *Computational Framework of the Fast Fourier Transform*. Siam, 1992.

[15] R. Clint Whaley and Jack Dongarra. Automatically tuned linear algebra software (ATLAS), 1998. `http://www.netlib.org/atlas/`.

On Materializations of Array-Valued Temporaries

Daniel J. Rosenkrantz, Lenore R. Mullin, and Harry B. Hunt III

Computer Science Department
University at Albany – SUNY
Albany, NY 12222
{djr, lenore, hunt}@cs.albany.edu

Abstract. We present results demonstrating the usefulness of monolithic program analysis and optimization prior to scalarization. In particular, models are developed for studying nonmaterialization in basic blocks consisting of a sequence of assignment statements involving array–valued variables. We use these models to analyze the problem of minimizing the number of materializations in a basic block, and to develop an efficient algorithm for minimizing the number of materializations in certain cases.

1 Introduction

Here, we consider the analysis and optimization of code utilizing operations and functions operating on entire arrays or array sections (rather than on the underlying scalar domains of the array elements). We call such operations *monolithic* array operations.

In [16], Veldhuizen and Gannon refer to traditional approaches to optimization as *transformational*, and say that a difficulty with "transformational optimizations is that the optimizer lacks an understanding of the *intent* of the code. ... More generally, the optimizer must infer the *intent* of the code to apply higher–level optimizations." The dominant approach to compiler optimization of array languages, Fortran90, HPF, etc., is transformational optimizations done after scalarization. In contrast, our results suggest that monolithic style programming, and subsequent monolithic analysis prior to scalarization, can be used to perform *radical* transformations based upon *intensional* analysis. Determining *intent* subsequent to scalarization seems difficult, if not impossible, because much global information is obfuscated. Finally, many people doing scientific programming find it easier and natural to program using high level monolithic array operations. This suggests that analysis and optimization at the monolithic level will be of growing importance in the future.

In this paper, we study materializations of array–valued temporaries, with a focus on elimination of array–valued temporaries in basic blocks.

There has been extensive research on nonmaterialization of array-valued intermediate results in evaluating array–valued expressions. The optimization of

S.P. Midkiff et al. (Eds.): LCPC 2000, LNCS 2017, pp. 127–141, 2001.

array expressions in APL entails nonmaterialization of array subexpressions involving composition of indexing operations [1, 2, 3, 4]. Nonmaterialization of array-valued intermediate results in evaluating array–valued expressions is done in Fortran90, HPF, ZPL [7], POOMA [5], C++ templates [14, 15], the Matrix Template Library (MTL) [8], active libraries [16], etc. Mullin's *Psi Calculus* model [10, 9] provides a uniform framework for eliminating materializations involving array addressing, decomposition, and reshaping. Nonmaterialization in basic blocks is more complicated than in expressions because a basic block may contain assignment statements that modify arrays that are involved in potential nonmaterializations. Some initial results on nonmaterialization applied to basic blocks appear in Roth, et.al. [6, 11, 12].

In Section 2, we develop a framework, techniques, algorithms, etc., for the study of nonmaterialization in basic blocks. We give examples of how code can be optimized via nonmaterialization in basic blocks, and then formalize this optimization problem. We then formalize graph–theoretic models to capture fundamental concepts of the relationship between array values in a basic block. These models are used to study nonmaterializations, and to develop techniques for minimizing the number of materializations. A lower bound is given on the number of materializations required. An optimization algorithm for certain cases is given. This algorithm is optimal; the number of materializations produced exactly matches the lower bound.

In Section 3, we give brief conclusions.

2 Nonmaterialization in Basic Blocks

2.1 Examples of Nonmaterialization

Many array operations, represented in HPF and Fortran90 as sectioning and as intrinsics such as `transpose`, `cshift`, `eoshift`, `reshape`, and `spread` involve the rearrangement and replication of array elements. We refer to these operations as *address shuffling operations*. These operations essentially utilize array indexing, and are independent of the domain of values of the scalars involved. Often it is unnecessary to generate code for these operations. Instead of *materializing* the result of such an operation (i.e. constructing the resulting array at run–time), the compiler can keep track of how elements of the resulting array can be obtained by appropriately addressing the operands of the address shuffling operation. Subsequent references to the result of the operation can be replaced by suitably modified references to the operands of the operation.

As an example of nonmaterialization, consider the following statement.

```
X = B + transpose(C)
```

Straightforward code for this example would first materialize the result of the `transpose` operation in a temporary array, say named Y, and then add this temporary array to B, assigning the result to X. Indeed, instead of using a single assignment statement, the programmer might well have expressed the

above statement as such a sequence of two statements, forming part of a basic block, as follows.

```
Y = transpose(C)
X = B + Y
```

A straightforward compiler might produce a separate loop for each of the above statements, but an optimizer could fuse the two loops, producing the following single loop.

```
forall (i = 1:N, j = 1:N)
  Y(i,j) = C(j,i)
  X(i,j) = B(i,j) + Y(i,j)
end forall
```

However, Y need not be materialized at all, yielding the following code.

```
forall (i = 1:N, j = 1:N)
  X(i,j) = B(i,j) + C(j,i)
end forall
```

Nonmaterialization is also an issue in optimizing distributed computation. Kennedy, et. al. [6], Roth [11], and Roth, et. al. [12] developed methods for optimizing stencil computations by nonmaterialization of selected `cshift` operations involving distributed arrays. This nonmaterialization was aimed at minimizing communications, including both the amount of data transmitted, and the number of messages used. The amounts of the shifts in the stencils involved were constants, so the compiler could determine which `cshift` operations in the basic block were potential beneficiaries of nonmaterialization. The compiler could analyze the basic block and choose not to materialize some of these `cshift` operations. The nonmaterialization technique in [6, 11, 12] used a sophisticated form of replication, where a subarray on a given processor was enlarged by adding extra rows and columns that were replicas of data on other processors. The actual computation of the stencil on each processor referred to the enlarged array on that processor.

For instance, consider the following example, taken from Roth, et. al. [12].

```
(1)     RIP = cshift(U,shift=+1,dim=1)
(2)     RIN = cshift(U,shift=-1,dim=1)
(3)     T = U + RIP + RIN
```

The optimized version of this code is the following. The `cshift` operations in statements (1) and (2) are replaced by `overlap_cshift` operations, which transmit enough data from `U` between processors so as to fill in the overlap areas on each processor. In statement (3), the references to `RIP` and `RIN` are replaced by references to `U`, annotated with appropriate shift values, expressed using superscripts.

```
(1)     call overlap_cshift(U,shift=+1,dim=1)
(2)     call overlap_cshift(U,shift=-1,dim=1)
(3)     T = U + U<+1,0> + U<−1,0>
```

2.2 Formulation of Nonmaterialization Optimization Problem for Basic Blocks

We now begin developing a framework for considering the problem of minimizing the number of materializations in a basic block (equivalently, maximizing the number of shuffle operations that are not materialized).

Definition 1. *A* **def–value** *in a basic block is either the initial value (at the beginning of the basic block) of an array variable or an occurrence of an array variable that is the destination of an assignment. (An assignment can be to the complete array, to a section of the array, or to a single element of the array.)*

Definition 2. *A* **complete–def** *is a definition to an array variable in which an assignment is made to the complete array. A* **partial–def** *is a definition to an array variable in which an assignment is made to only some of the array elements, and the other array elements retain their prior values. An* **initial–def** *is an initial value of an array variable.*

Note that each def–value can be classified as either a complete–def, a partial–def, or an initial–def. For instance, consider the following example.

Example 1.

```
(1)     B = cshift(A,shift=+5,dim=1)
(2)     C = cshift(A,shift=+3,dim=1)
(3)     D = cshift(A,shift=-2,dim=1)
(4)     B(i) = 50
(5)     R(2:199) = B(2:199) + C(2:199) + D(2:199)
```

`A`, `B`, `C`, and `D` all dead at end of basic block, `R` live.

For convenience, we identify each def–value by the name of its variable, superscripted with either 0 for an initial–def, or the statement assigning the def–value for a noninitial–def. Example 1 involves initial–defs $\mathtt{A}^0$ and $\mathtt{R}^0$, complete–defs $\mathtt{B}^1$, $\mathtt{C}^2$, and $\mathtt{D}^3$, and partial–defs $\mathtt{B}^4$ and $\mathtt{R}^5$.

An important issue for nonmaterialization is determining whether a given shuffle operation is a candidate for nonmaterialization. The criteria for a given shuffle operation being a candidate is that it be both *safe* and *profitable*. The criteria for being safe is that the source array of the shuffle is not modified while the definition of the destination array is live, and that the destination array is not partially modified while the source array is live [13, 11]. The criteria for profitability can depend on the optimization goal, the shuffle operation involved, the shape of the arrays involved, architectural features of the computing environment on which the computation will be performed, and the distribution/alignment of the arrays involved. For instance, Roth [11] gives criteria for stencil computations on distributed machines. For purposes of this paper, we assume the availability of appropriate criteria for determining whether a given shuffle operation is a candidate for nonmaterialization.

Definition 3. *An* **eligible statement** *is an assignment statement whose right side consists of a shuffle operation meeting the criteria for nonmaterialization. We say that the def–value occurring on the right side of an eligible statement is* **mergeable** *with the def–value occurring on the left side of the statement. An* **ineligible statement** *is a statement that is not an eligible statement.*

In a given basic block, the decisions as to which eligible statements should be nonmaterialized are interdependent. Roth [11] uses a "greedy" algorithm to choose which eligible statements to nonmaterialize. Namely, at each step, his algorithm chooses the first eligible statement, and modifies the basic block so that this statement is not materialized. Each such choice can cause subsequent previously eligible statements to become ineligible. Example 1 illustrates this phenomenom.

Consider the basic block shown in Example 1. Assume that statements (1), (2), and (3) are eligible statements. Roth's method would first consider the shuffle operation in statement (1), and choose to merge `B` with `A`. This merger would be carried out as shown below, changing the partial–def of `B` in statement (4) into a partial–def of `A`. In statements (4) and (5), the reference to `B` is replaced by a reference to `A`, annotated with the appropriate shift value.

```
(1)     call overlap_cshift(A,shift=+5,dim=1)
(2)     C = cshift(A,shift=+3,dim=1)
(3)     D = cshift(A,shift=-2,dim=1)
(4)     A<+5>(i) = 50
(5)     R(2:199) = A<+5>(2:199) + C(2:199) + D(2:199)
```

Note that the new partial–def of `A` in statement (4) makes statements (2) and (3) unsafe, and therefore ineligible, thereby preventing `C` and `D` from being merged with `A`. Thus there would be three copies of the array. **But**, it is better to make a separate copy of `A` in `B`, and let `A`, `C`, and `D` all share the same copy, as follows:

```
(1)     B = cshift(A,shift=+5,dim=1)
(2)     call overlap_cshift(A,shift=+3,dim=1)
(3)     call overlap_cshift(A,shift=-2,dim=1)
(4)     B(i) = 50
(5)     R(2:199) = B(2:199) + A<+3>(2:199) + A<-2>(2:199)
```

The above example generalizes. Suppose that instead of just `C` and `D`, there were n additional variables which could be merged with `A`. Roth's method would only merge `B` with `A`, resulting in a total of $n+1$ materialized arrays. In contrast, by making a separate copy for `B`, there would be a total of only two materialized arrays. Consequently, the "greedy" algorithm can be arbitrarily worse than optimal.

The optimization problem we consider is to minimize the number of materializations in a basic block, under the following seven assumptions:

Assumptions:

1. No dead code. (**No Dead Assumption**)
2. Arrays with different names are not aliased[1]. (**No Aliasing Assumption**)
3. No rearrangement of the ineligible statements in a basic block. (**Fixed Sequence Assumption**)
4. The *mergeability* relation between def–values in a basic block is symmetric and transitive. (**Full Mergeability Assumption**)
5. There is no fine–grained analysis of array indices. Recall that each def–value is classified as being either an initial–def, a complete–def or a partial–def. No analysis is made as to whether two partial–defs overlap or whether a given use overlaps a given partial–def. (**Coarse Analysis Assumption**)
6. The initial value of each variable that is live on entry to a basic block is stored in the variable's official location. (**Live Entry Assumption**)
7. A variable that is live at exit from a basic block, must have its final value at the end of the basic block stored in the variable's official location. (**Live Exit Assumption**)

In considering Assumptions 6 and 7, note that any def–value that is not live at entry or exit from a basic block need not be stored in the official location of its variable.

2.3 The Clone Forest Model

We now consider sets of mutually mergeable def–values, represented as trees in the *clone forest*, defined as follows.

Definition 4. *The* **clone forest** *for a basic block is a graph with a node for each def–value occurring in the basic block, and a directed edge from the source def–value to the destination def–value of each eligible shuffle operation. A* **clone set** *is the set of def–values occurring in a tree of the clone forest. The* **root** *of a given clone set is the def–value that is the root of the clone tree corresponding to the clone set.*

As an example, the clone forest for Example 1 is shown in Figure 1.

Definition 5. *A* **materialization** *is either an initial–def or a complete–def.*

An important observation is that at least one materialization is needed for each clone set whose root is either an initial–def or a complete–def. The overall optimization problem can be viewed as minimizing the total number of materializations for all the clone sets for a given basic block. In Example 1, the only clone set with more than one member is $\{\mathtt{A}^0, \mathtt{B}^1, \mathtt{C}^2, \mathtt{D}^3\}$. The basic block can be optimized by materializing $\mathtt{A}^0$ and $\mathtt{B}^1$ from this clone set, and materializing $\mathtt{R}^0$ from

[1] However, the techniques in this paper can be suitably modified to use aliasing information.

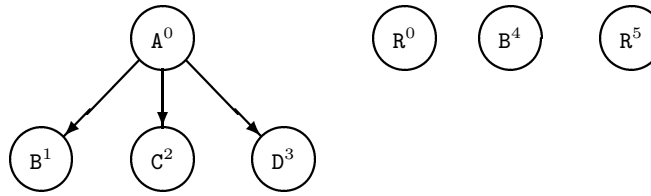


Fig. 1. Clone forest for Example 1

clone set $\{\mathtt{R}^0\}$. Clone sets $\{\mathtt{B}^4\}$ and $\{\mathtt{R}^5\}$ do not require new materializations; they can be obtained by partial–defs to already materialized arrays.

Note that even if there is a materialization for a given clone set, the materialized member of the clone set need not necessarily be the root of the clone set. This freedom may be necessary for optimization, as illustrated in the following example.

Example 2.

```
(1)     A = C + D
(2)     B = cshift(A,shift=+3,dim=1)
(3)     z = A(i) + B(i)
```

`A` dead, `B` live at end of basic block.

Consider the clone set consisting of $\mathtt{A}^1$ and $\mathtt{B}^2$. The root, $\mathtt{A}^1$, is a complete–def, and so at least one materialization is needed for this clone set. Since `A` is dead at the end of the basic block, and `B` is live, it is advantageous to materialize the value of the clone set in variable `B` instead of variable `A`. Thus, an optimized version of the basic block might be as follows. Here, the partial result `C + D` is annotated by the compiler to indicate that the value stored in variable `B` should be appropriately shifted.

```
(1′)    B = (C + D)<+3>
(3)     z = B<−3>(i) + B(i)
```

We now establish a lower bound on the number of separate copies required for a given clone set.

Definition 6. *A given def–value in a basic block is* **transient** *if it is not the last def to its variable in the basic block, and the subsequent def to its variable is a partial–def.*

For instance, in Example 1, def–value $\mathtt{B}^1$ is transient.

Definition 7. *A given def–value in a basic block is* **persistent** *if it is the last def to its variable in the basic block, and its variable is live at the end of the basic block.*

For instance, in Example 1, def–value $\mathtt{R}^5$ is persistent.

Theorem 1. *Consider the clone tree for a given clone set. There needs to be a separate copy for each def–value in the clone tree that is either transient or persistent.*

Proof: Two transient def–values cannot share the same copy because of Assumptions 1 and 5. Two persistent def–values cannot share the same copy because of Assumption 7. A transient def–value and a persistent def–value cannot share the same copy because of Assumptions 1 and 7. □

2.4 Main Techniques for Nonmaterialization Under Constrained Statement Rearrangement

In this section, we outline our main techniques and results on the nonmaterialization problem for basic blocks under constrained statement rearrangement.

First, note that for each transient def–value of a given clone set, there is a subsequent partial–def to its variable, and this partial–def is the root of another clone set. This relationship can be represented in a graph with a node for each clone set, as formalized by the following definition.

Definition 8. *The* **version forest** *for a basic block is a graph with a node for each clone set in the basic block, and a directed edge from clone set α to clone set β if the root of clone set β is a partial–def that modifies a member of α. The* **root def–value** *of a given tree in the version forest is the root def–value of the clone set of the root node of the version tree.*

As an example, the version forest for Example 1 is shown in Figure 2. There are two trees in the version forest, with root def–values $\mathtt{A}^0$ and $\mathtt{R}^0$, respectively.

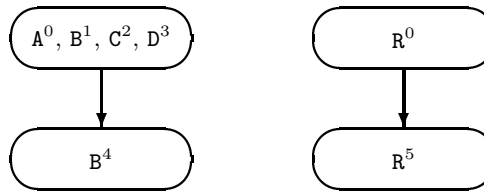


Fig. 2. Version forest for Example 1

Definition 9. *A node of a version forest is* **persistent** *if any of its def–values are persistent.*

For instance, in Figure 2, the only persistent node is the node for clone set $\{\mathtt{R}^5\}$.

Definition 10. *The* **origin point** *of an initial–def is just before the basic block. The* **origin point** *of a complete–def or a partial–def is the statement in which it receives its value. The* **origin point** *of a clone set is the origin point of the root def–value of the clone set.*

For instance, in Figure 2, the origin point of $\{\mathtt{A}^0, \mathtt{B}^1, \mathtt{C}^2, \mathtt{D}^3\}$ is (0), of $\{\mathtt{B}^4\}$ is (4), of $\{\mathtt{R}^0\}$ is (0), and of $\{\mathtt{R}^5\}$ is (5).

Definition 11. *The* **demand point** *of a persistent def–value in a basic block is just after the basic block. The* **demand point** *of a non–persistent def–value is the last ineligible statement that contains a use of the def–value, and is (0) if there is no ineligible statement that uses it. The* **demand point** *of a clone set is the maximum demand point of the def–values in the clone set.*

For instance, in Figure 2, the demand point of $\{\mathtt{A}^0, \mathtt{B}^1, \mathtt{C}^2, \mathtt{D}^3\}$ is (5), of $\{\mathtt{B}^4\}$ is (5), of $\{\mathtt{R}^0\}$ is (0), and of $\{\mathtt{R}^5\}$ is (6).

Next, we formalize the concept of an *essential node* of a version tree. Informally, an essential node is a node whose value is needed *after* the values of all its child nodes (if any) have been produced. Consequently, the materialization used to hold the value of an essential node in order to satisfy this need for the value cannot be the same as any of the materializations that are modified to produce the values of the child nodes.

Definition 12. *A node of a version tree is an* **essential node** *if it is either a leaf node, or a non–leaf node whose demand point exceeds the maximum origin point of its children.*

For instance, in Figure 2, node $\{\mathtt{A}^0, \mathtt{B}^1, \mathtt{C}^2, \mathtt{D}^3\}$ is essential because its demand point (5) exceeds the origin point (4) of its child $\{\mathtt{B}^4\}$. Node$\{\mathtt{R}^0\}$ is not essential because its demand point (0) does not exceed the origin point (5) of its child $\{\mathtt{R}^5\}$. Nodes $\{\mathtt{B}^4\}$ and $\{\mathtt{R}^5\}$ are leaf nodes, and so are essential.

Proposition 1. *Every persistent node of a version tree is essential.*

Proof: A persistent leaf node is essential by definition. A persistent non–leaf node is essential because its demand point is just after the basic block, while the origin point of each of its children is within the basic block. □

Recall that there are three materializations in the optimized code for Example 1. There is the materialization of $\mathtt{A}^0$, which is also used for $\mathtt{C}^2$ and $\mathtt{D}^3$. There is a materialization of transient def–value $\mathtt{B}^1$, which is subsequently modified by the partial–def that creates $\mathtt{B}^4$. Finally, there is a materialization of transient def–value $\mathtt{R}^0$, which is subsequently modified by the partial–def that creates $\mathtt{R}^5$. In this example, the number of materializations in the optimized code equals the number of essential nodes in the version forest. Each of the three essential nodes of the version forest (shown in Figure 2) for this basic block can be associated with one of these materializations. Node $\{\mathtt{A}^0, \mathtt{B}^1, \mathtt{C}^2, \mathtt{D}^3\}$ is associated with materialization $\mathtt{A}^0$. Node $\{\mathtt{B}^4\}$ is associated with materialization $\mathtt{B}^1$ (via the partial–def to $\mathtt{B}^4$). Node $\{\mathtt{R}^5\}$ is associated with materialization $\mathtt{R}^0$ (via the partial–def to $\mathtt{R}^5$). Nonessential node $\{\mathtt{R}^0\}$ is associated with the same materialization ($\mathtt{R}^0$) as is associated with its child node, so that the partial–def $\mathtt{R}^5$ that creates the child node can modify the variable R associated with the parent node.

Now consider the following example, which illustrates changing the destination variable of a complete–def, so that a nonessential parent node utilizes the same variable as its child node.

Example 3.

```
(1)      A = G + H
(2)      B = cshift(A,shift=+5,dim=1)
(3)      C = cshift(A,shift=+3,dim=1)
(4)      x = C(i) + 3
(5)      B(i) = 50
(6)      A = G - H
```

A and B live, C dead at end of basic block.

The version forest for this basic block is shown in Figure 3[2]. The basic block can be optimized by merging A^1, B^2, and C^3 into a single copy stored in B's official location, as shown below. (Thus, the references to A, B, and C in statements (1) through (5) are all replaced by references to B.)

```
(1′)     B = (G + H)<+5>
(4)      x = B<−2>(i) + 3
(5)      B(i) = 50
(6)      A = G - H
```

The complete–def A^1 is transformed into a complete–def to variable B, since this permits the version tree with root def–value A^1 to be evaluated using only one materialization. The materialization is in variable B, rather than variable A, because def–value B^5 is persistent. Def–value A^6 is a persistent def, and so uses the official location of A.

Note that the version forest contains two essential nodes, $\{B^5\}$ and $\{A^6\}$, and the optimized code uses two materializations. In the optimized code, essential node $\{B^5\}$ utilizes variable B, and is associated with new materialization $B^{1'}$ (via the partial–def to B^5). Essential node $\{A^6\}$ is associated with materialization A^6. Nonessential node $\{A^1, B^2, C^3\}$ is associated with the same materialization ($B^{1'}$) as is associated with its child node, so that the partial–def B^5 that creates the child node can modify the variable B utilized by the parent node. The reference in statement (4) to def–value C^3 from node $\{A^1, B^2, C^3\}$ is replaced by a reference to variable B, which holds def–value $B^{1'}$.

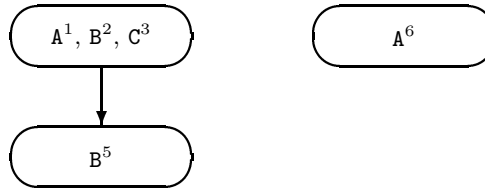


Fig. 3. Version forest for Example 3

[2] Since the only defs to G and H in the basic block are initial–defs, there is no need to include nodes for G^0 and H^0 in the version forest.

The following example illustrates how the freedom to rearrange eligible statements, as permitted by Assumption 3 (Fixed Sequence Assumption), can be exploited to reduce the number of materializations.

Example 4.

```
(1)      B = cshift(A,shift=+5,dim=1)
(2)      A(i) = 120
(3)      x = B(j) + 3
(4)      C = cshift(A,shift=+2,dim=1)
```

`A` and `B` dead, `C` live at end of basic block.

The version forest for this basic block is shown in Figure 4. Note that both nodes of the version forest are essential. The basic block can be optimized by moving the complete–def $\mathtt{C}^4$ forward so that it occurs before statement (2), and letting the partial–def in statement (2) modify `C`. The optimized code is shown below, where old statement (4) is relabeled (4′) and is now the first statement in the basic block.

```
(4′)     C = cshift(A,shift=+2,dim=1)
(2′)     C<−2>(i) = 120
(3′)     x = A<+5>(j) + 3
```

The optimized code contains two materializations, and is made possible by changing the reference to $\mathtt{B}^1$ in the demand point statement (3) to be a reference to its clone $\mathtt{A}^0$. The optimized code utilizes materialization $\mathtt{A}^0$ for node $\{\mathtt{A}^0, \mathtt{B}^1\}$, and utilizes, via partial–def $\mathtt{C}^{2'}$, new materialization $\mathtt{C}^{4'}$ for node $\{\mathtt{A}^2, \mathtt{C}^4\}$.

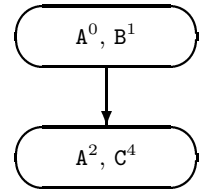


Fig. 4. Version forest for Example 4

Theorem 2. *The number of materializations needed to evaluate a given version tree is at least the number of essential nodes.*

Proof Sketch: Let us say that a given def–value **utilizes** a given materialization if the def–value is obtained from the materialization via a sequence of zero or more partial–defs. Note that all the def–values utilizing a given materialization are def–values for the same variable, and the version forest nodes containing these def–values lie along a directed path in the version forest.

Let us say that a given reference to a def–value **utilizes** the materialization that is utilized by the referenced def–val. Each non–persistent essential node of the given version tree has a nonzero demand point. We say that a **final use** of a non–persistent essential node is a use of a def–value from this node in the demand point statement for the node. We envision arbitrarily selecting a final use of each non–persistent essential node, and we refer to the materialization utilized by this final use as the **key utilization** of the node. We say that the **final use** of a persistent node is one of the persistent def–values in the node, and the **key utilization** of the node is the materialization utilized by this def–value.

Now consider the version forest corresponding to an optimized evaluation of the basic block, where the optimization is constrained by the assumptions given above. Each node of the version forest for the optimized basic block corresponds to a node of the given version forest. In the optimized code, the key utilizations of two distinct essential nodes cannot be the same materialization. □

Next, we note that it is not always possible to independently optimize each version tree, because of possible interactions between the initial and persistent def–values of the same variable. This interaction is captured by the concept of "persistence conflict", as defined below.

Definition 13. *A* **persistence conflict** *in a basic block is a variable that is live on both entry to and exit from the basic block.*

Persistence conflicts may prevent each version tree for a given basic block from being evaluated with a minimum number of materializations, as illustrated by the following example.

Example 5.

```
(1)     B = cshift(A,shift=+5,dim=1)
(2)     A = G + H
(3)     B(i) = 50
(4)     x = B(j) + 3
```

`A` live and `B` dead at end of basic block.

The version forest for this basic block is shown in Figure 5. The only essential nodes are $\{\mathtt{B}^3\}$ and $\{\mathtt{A}^2\}$. Each of the two version trees can by itself be evaluated using only a single materialization, corresponding to def–values $\mathtt{A}^0$ and $\mathtt{A}^2$, respectively. However, there is a persistence conflict involving variable `A`. Assumption 3 prevents statements (2), (3), and (4) from being reordered, so in this example, three materializations are necessary.

We next show that for a version tree with no persistent nodes, the lower bound of Theorem 2 has a matching upper bound. As part of the proof, we provide an algorithm for producing the optimized code.

Theorem 3. *Assuming no persistence conflicts, a version tree with no persistent nodes can be computed using one materialization for each essential node, and no other materializations.*

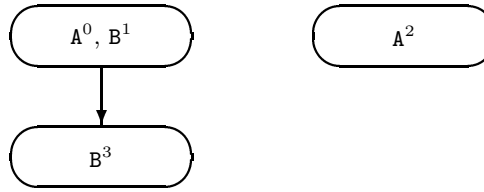


Fig. 5. Version forest for Example 5

Proof Sketch: The following algorithm is given the version tree and basic block as input, and produces the optimized code. If there are multiple version trees, Steps 1 and 2 of the algorithm can be done for each version tree, and then the basic block can be transformed. The algorithm associates a variable name with each version tree node. We call this variable name the **utilization–variable** of the node. In the code produced by the algorithm, all the uses of def–values from a given node of the version tree reference the utilization–variable of that node. In the portion of the algorithm that determines the utilization–variable of each node, we envision the children of each node being ordered by their origin point, so that the child with the last origin point is the rightmost child.

Step 1. A *unique* utilization–variable is associated with each essential node, as follows. With the following exception, the utilization–variable for each essential node is a unique temporary variable[3]. The exception occurs if the root def–value is an initial–def. In this case, the name of the variable of the initial–def is made the utilization–variable for the highest essential node on the path from the root of the version tree to the rightmost leaf.

Step 2. The utilization–variables of nonessential nodes are determined in a bottom–up manner, as follows. The utilization–variable of a nonessential node is made the same as the utilization–variable of its rightmost child (i.e., the child with the maximum origin point).

Step 3. In the evaluation of the basic block, the ineligible statements occur in the same order as given. A materialized shuffle statement is included for each child node whose utilization–variable is different from the utilization–variable of its parent node. This shuffle statement is placed just after the origin point of the parent node. All other eligible statements for the version tree are deleted[4].

Step 4. For each shuffle statement included in Step 3, the utilization–variable of the parent node is made the source of the shuffle statement, and the utilization–variable of the child node is made the destination of the shuffle statement.

Step 5. If the root def–value is a complete–def, then in the evaluation of the basic block, the utilization–variable of the root node is used in the left side of the statement for the complete–def. For each nonroot node, the utilization–variable

[3] Alternately, one of the variable names occurring in the node's clone set can be used, but with each variable name associated with at most one essential node.

[4] However, a `cshift` operation involving distributed arrays is replaced by an `overlap_cshift` operation placed just after the origin point of the parent node.

of the node is used in the left side of the partial–def statement occurring at the origin point of the node.

Step 6. In the ineligible statements, each use of a def–val from the version tree is replaced by a reference to the utilization–variable of the version tree node containing the def–value, with appropriate shift annotation if needed.

Consider the code produced by the above algorithm. In the evaluation of the version tree, the number of variable names used equals the number of essential nodes, and the number of materializations equals the number of variable names used. □

In the optimized code for Example 1, node $\{\mathtt{A}^0, \mathtt{B}^1, \mathtt{C}^2, \mathtt{D}^3\}$ utilizes materialization $\mathtt{A}^0$ and variable A, node $\{\mathtt{B}^4\}$ utilizes materialization $\mathtt{B}^1$ and variable B, and nodes $\{\mathtt{R}^0\}$ and $\{\mathtt{R}^5\}$ utilize materialization $\mathtt{R}^0$ and variable R. In Example 2, the version tree has only one node $(\{\mathtt{A}^1, \mathtt{B}^2\})$; the optimized code for this node utilizes materialization $\mathtt{B}^{1'}$ and variable B. In the optimized code for Example 3, nodes $\{\mathtt{A}^1, \mathtt{B}^2, \mathtt{C}^3\}$ and $\{\mathtt{B}^5\}$ utilize materialization $\mathtt{B}^{1'}$ and variable B, and node $\{\mathtt{A}^6\}$ utilizes materialization $\mathtt{A}^6$ and variable A. In the optimized code for Example 4, node $\{\mathtt{A}^0, \mathtt{B}^1\}$ utilizes materialization $\mathtt{A}^0$ and variable A, and node $\{\mathtt{A}^2, \mathtt{C}^4\}$ utilizes materialization $\mathtt{C}^{4'}$ and variable C.

Theorem 4. *Assuming no persistence conflicts, the number of materializations needed to compute a version tree with no persistent nodes is the number of essential nodes of the version tree.*

Proof: Immediate from Theorems 2 and 3. □

When there are persistance conflicts, the number of materializations needed to compute a version tree is at most the number of essential nodes of the version tree, plus the number of extra persistent def-values in persistent nodes, plus the number of persistance conflicts. This follows, since each persistance conflict can be eliminated by introducing an extra materialization for the variable involved. However, when a persistance conflict involves a variable for which the initial def–value and persistent def–value both occur in the same version tree, we can determine efficiently if this extra materialization is indeed necessary.

3 Conclusions

Minimizing nonmaterializations is a deep problem that can be approached from an intensional perspective, utilizing an analysis of the role of entire arrays. In a single expression, minimizing materializations is mainly an issue of proper array accessing. However, in a basic block, the decisions as to which materializations to eliminate interact, so minimizing materializations is a combinatorial optimization problem. The concepts of clone sets, version forests, and essential nodes seem to model fundamental aspects of the problem. Under the assumptions listed in Section 2.2, each essential node of a version forest requires a distinct materialization, thereby establishing a lower bound on the number of materializations required. Theorem 3 provides an algorithm when persistent variables are not an

issue. The algorithm produces code with one materialization per essential node of the version tree, and no additional materializations. The algorithm is optimal, in the sense of producing the minimum number of materializations.

References

[1] P. S. Abrams. *An APL Machine*. PhD thesis, Stanford University, 1970.

[2] T. A. Budd. An APL compiler for a vector processor. *ACM Transactions on Programming Languages and Systems*, 6(3):297–313, July 1984.

[3] L. J. Guibas and D. K. Wyatt. Compilation and delayed evaluation in APL. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 1–8, Jan. 1978.

[4] A. Hassitt and L. E. Lyon. Efficient evaluation of array subscripts of arrays. *IBM Journal of Research and Development*, 16(1):45–57, Jan. 1972.

[5] W. Humphrey, S. Karmesin, F. Bassetti, and J. Reynders. Optimization of data–parallel field expressions in the POOMA framework. In Y. Ishikawa, R. R. Oldehoeft, J. Reynders, and M. Tholburn, editors, *Proc. First International Conference on Scientific Computing in Object–Oriented Parallel Environments (ISCOPE '97)*, volume 1343 of *Lecture Notes in Computer Science*, pages 185–194, Marina del Rey, CA, Dec. 1997. Springer–Verlag.

[6] K. Kennedy, J. Mellor-Crummey, and G. Roth. Optimizing Fortran 90 shift operations on distributed–memory multicomputers. In *Proceedings Eighth International Workshop on Languages and Compilers for Parallel Computing*, volume 1033 of *Lecture Notes in Computer Science*, Columbus, OH, Aug. 1995. Springer–Verlag.

[7] C. Lin and L. Snyder. ZPL: An array sublanguage. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, pages 96–114, Portland, OR, Aug. 1993. Springer-Verlag.

[8] A. Lumsdaine. The matrix template library: A generic programming approach to high performance numerical linear algebra. In *Proceedings of International Symposium on Computing in Object-Oriented Parallel Environments*, 1998.

[9] L. Mullin. The Psi compiler project. In *Workshop on Compilers for Parallel Computers*. TU Delft, Holland, 1993.

[10] L. M. R. Mullin. *A Mathematics of Arrays*. PhD thesis, Syracuse University, Dec. 1988.

[11] G. Roth. *Optimizing Fortran90D/HPF for Distributed–Memory Computers*. PhD thesis, Dept. of Computer Science, Rice University, Apr. 1997.

[12] G. Roth, J. Mellor-Crummey, K. Kennedy, and R. G. Brickner. Compiling stencils in High Performance Fortran. In *Proceedings of SC'97: High Performance Networking and Computing*, San Jose, CA, Nov. 1997.

[13] J. T. Schwartz. Optimization of very high level languages – I. Value transmission and its corollaries. *Computer Languages*, 1(2):161–194, June 1975.

[14] T. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995. Reprinted in *C++ Gems*, ed. Stanley Lippman.

[15] T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in *C++ Gems*, ed. Stanley Lippman.

[16] T. L. Veldhuizen and D. Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*, Yorktown Heights, NY, 1998. SIAM.

Experimental Evaluation of Energy Behavior of Iteration Space Tiling

Mahmut Kandemir, Narayanan Vijaykrishnan, Mary Jane Irwin, and Hyun Suk Kim

Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802-6106
http://www.cse.psu.edu/~mdl

Abstract. Optimizing compilers have traditionally focused on enhancing the performance of a given piece of code. With the proliferation of embedded software, it is becoming important to identify the energy impact of these traditional performance-oriented optimizations and to develop new energy-aware schemes. Towards this goal, this paper explores the energy consumption behavior of one of the widely-used loop-level compiler optimizations, iteration space tiling, by varying a set of software and hardware parameters.
Our results show that the choice of tile size and input size critically impacts the system energy consumption. Specifically, we find that the best tile size for the least energy consumed is different from that for the best performance. Also, tailoring tile size to the input size generates better energy results than working with a fixed tile size. Our results also reveal that tiling should be applied more or less aggressively based on whether the low power objective is to prolong the battery life or to limit the energy dissipated within a package.

1 Introduction

With the continued growth in the use of mobile personal devices, the design of new energy-optimization techniques has become vital. While many techniques have been proposed at the circuit and architectural level, there is clearly a need for making the software that executes on these hardware energy-efficient as well. This is of importance as the application code that runs on an embedded device is the primary factor that determines the dynamic switching activity, one of the contributors to dynamic power dissipation.

Power is the rate at which energy is delivered or exchanged and is measured in Watts. Power consumption impacts battery energy-density limitations, circuit reliability, and packaging costs. Packaging cost constraints typically require the use of cheaper packages that impose strict limits on power dissipation. For example, plastic packages typically limit the power dissipation to 2W. Thus, limiting the power dissipated within a single package is important to meet cost constraints. Energy, measured in Joules, is the power consumed over time and is

S.P. Midkiff et al. (Eds.): LCPC 2000, LNCS 2017, pp. 142–157, 2001.

an important metric to optimize for prolonging battery life. The proliferation of embedded and mobile devices has made low power designs vital for prolonging the limited battery power. The energy of commercially available re-chargeable batteries has improved only 2% per year over the past fifty years. Since the entire system is operated by the same battery in such devices, the goal of low power optimization must encompass all components in the system. In our work, we investigate the trade-offs when both the entire system energy and the energy consumed within a single package need to be simultaneously optimized. In particular, we study this trade-off when software optimizations are applied.

Optimizing compilers have traditionally focused on enhancing the performance of a give piece of code (e.g., see [26] and the references therein). With the proliferation of embedded software, it is becoming important to identify the energy impact of these traditional performance-oriented optimizations and to develop new energy-aware schemes. We believe that the first step in developing energy-aware optimization techniques is to understand the influence of widely used program transformations on energy consumption. Such an understanding would serve two main purposes. First, it will allow compiler designers to see whether current *performance-oriented* optimization techniques are sufficient for minimizing the energy-consumption, and if not, what additional optimizations are needed. Second, it will give hardware designers an idea about the influence of widely used compiler optimizations on energy-consumption, thereby enabling them to evaluate and compare different energy-efficient design alternatives with these optimizations.

While it is possible and certainly beneficial to evaluate each and every compiler optimization from energy perspective, in this paper, we focus our attention on *iteration space* (*loop*) *tiling,* a popular high-level (loop-oriented) transformation technique used mainly for optimizing data locality [26, 25, 24, 15, 14, 23]. This optimization is important because it is very effective in improving data locality and it is used by many optimizing compilers from industry and academia. While behavior of tiling from performance perspective has been understood to a large extent and important parameters that affect its performance have been thoroughly studied and reported, its influence on system energy is yet to be fully understood. In particular, its influence on energy consumption of different system components (e.g., datapath, caches, main memory system, etc.) is yet to be explored in detail.

Having identified loop tiling as an important optimization, in this paper, we evaluate it, with the help of our cycle-accurate simulator, *SimplePower* [27], from the energy point of view considering a number of factors. The scope of our evaluation includes different tiling styles (strategies), modifying important parameters such as input size and tile size (blocking factor) and hardware features such as cache configuration. In addition, we also investigate how tiling performs in conjunction with two recently-proposed energy-conscious cache architectures, how current trends in memory technology will affect its effectiveness on different system components, and how it interacts with other loop-oriented optimizations

as far as energy consumption is concerned. Specifically, in this paper, we make the following contributions:

- We report the energy consumption caused by different styles of tiling using a matrix-multiply code as a running example.
- We investigate energy-sensitivity of tiling to tile size and input size.
- We investigate its energy performance on several cache configurations including a number of new cache architectures and different technology parameters.
- We evaluate the energy consumption and discuss the results when tiling is accompanied by other code optimizations.

Our results show that while tiling reduces the energy spent in main memory system, it may increase the energy consumed in the datapath and on-chip caches. We also observed a great variation on energy performance of tiling when tile size is modified; this shows that determining optimal tile sizes is an important problem for compiler writers for power-aware embedded systems. Also, tailoring tile size to the input size generates better energy results than working with a fixed tile size for all inputs.

The remainder of this paper is organized as follows. In Section 2, we introduce our experimental platform and experimental methodology. In Section 3, we report energy results for different styles of tiling using a matrix-multiply code as a running example and evaluate the energy sensitivity of tiling with respect to software and hardware parameters and technological trends. In Section 4, we discuss related work and conclude the paper with a summary in Section 5.

2 Our Platform and Methodology

The experiments in this paper were carried out using the *SimplePower* energy estimation framework [27]. This framework includes a transition-sensitive, cycle-accurate datapath energy model that interfaces with analytical and transition-sensitive energy models for the memory and bus sub-systems, respectively. The datapath is based on the ISA of the integer subset of the SimpleScalar architecture [1] and the modeling approach used in this tool has been validated to be accurate (average error rate of 8.98%) using actual current measurements of a commercial DSP architecture [3]. The memory system of *SimplePower* can be configured for different cache sizes, block sizes, associativities, write and replacement policies, number of cache sub-banks, and cache block buffers. *SimplePower* uses the on-chip cache energy model proposed in [9] using 0.8μ technology parameters and the off-chip main memory energy per access cost based on the Cypress SRAM CY7C1326-133 chip. In our design, the datapath and instruction and data caches are assumed to be in a single package and the main memory in a different package. We input our C codes into this framework to obtain the energy results.

All the tiled codes in this paper are obtained using an extended version of the source-to-source optimization framework discussed in [10]. Each tiled version is named by using the indices of the loops that have been tiled. For example, `ij`

denotes a version where only the i and j loops have been tiled. Unless otherwise stated, in both the *tile loops* (i.e., the loops that iterate over tiles) and the *element loops* (i.e., the loops that iterate over elements in a given tile), the original order of loops is preserved except that the untiled loop(s) is (are) placed right after the tile loops and all the element loops are placed into the innermost positions. Note that the matrix-multiply code is fully permutable [14] and all styles of tilings are legal from the data dependences perspective. We also believe that the matrix-multiply code is an interesting case study because (as noted by Lam et al. [14]) locality is carried in three different loops by three different array variables. However, similar data reuse patterns and energy behaviors can be observed in many codes from the signal and video processing domains.

In all the experiments, our default data cache is 4 KB, one-way associative (direct-mapped) with a line size of 32 bytes. The instruction cache that we simulated has the same configuration. All the reported energy values in this paper are in Joules (J).

3 Experimental Evaluation

3.1 Tiling Strategy

In our first set of experiments, we measure the energy consumed by the matrix-multiply code tiled using different strategies. Figure 1 shows the total energy consumption of eight different versions of the matrix-multiply code (one original and seven tiled) for an input size of N=50. We observe that tiling reduces the overall energy consumption of this code.

In order to further understand the energy behavior of these codes, we break down the energy consumption into different system components: datapath, data cache, instruction cache, and main memory. As depicted in Figure 1, tiling a larger number of loops in general increases the datapath energy consumption. The reason for this is that loop tiling converts the input code into a more complex code which involves complicated loop bounds, a larger number of nests, and macro/function calls (for computing loop upper bounds). All these cause more branch instructions in the resulting code and more comparison operations that, in turn, increase the switching activity (and energy consumption) in the datapath. For example, tiling only the i loop increases the datapath energy consumption by approximately 10%.

A similar negative impact of tiling (to a lesser extent) is also observed in the instruction cache energy consumption (not shown in figure). When we tile, we observe a higher energy consumption in the instruction cache. This is due to the increased number of instructions accessed from the cache due to the more complex loop structure and access pattern. However, in this case, the energy consumption is not that sensitive to the tiling strategy, mainly because the small size of the matrix-multiply code does not put much pressure on the instruction cache. We also observe that the number of data references increase as a result of tiling. This causes an increase in the data cache energy. We speculate that

this behavior is due to the influence of back-end compiler optimizations when operating on the tiled code.

When we consider the main memory energy, however, the picture totally changes. For instance, when we tile all three loops, the main memory energy becomes 35.6% of the energy consumed by the untiled code. This is due to reduced number of accesses to the main memory as a result of better data locality. In the overall energy consumption, the main memory energy dominates and the tiled versions result in significant energy savings. To sum up, *we can conclude that* (*for this matrix-multiply code*) *loop tiling increases the energy consumption in datapath, instruction cache, and data cache, but significantly reduces the energy in main memory.* Therefore, if one only intends to prolong the battery life, one can apply tiling aggressively. If, on the other hand, the objective is to limit the energy dissipated within each package (e.g., the datapath+caches package), one should be more careful as tiling tends to increase both datapath and cache energies. It should also be mentioned that, in all the versions experimented with here, tiling improved the performance (by reducing execution cycles). Hence, it is easy to see that since there is an increase in the energy spent (and decrease in execution time) within the package that contains datapath and caches, tiling causes an increase in average power dissipation for that package.

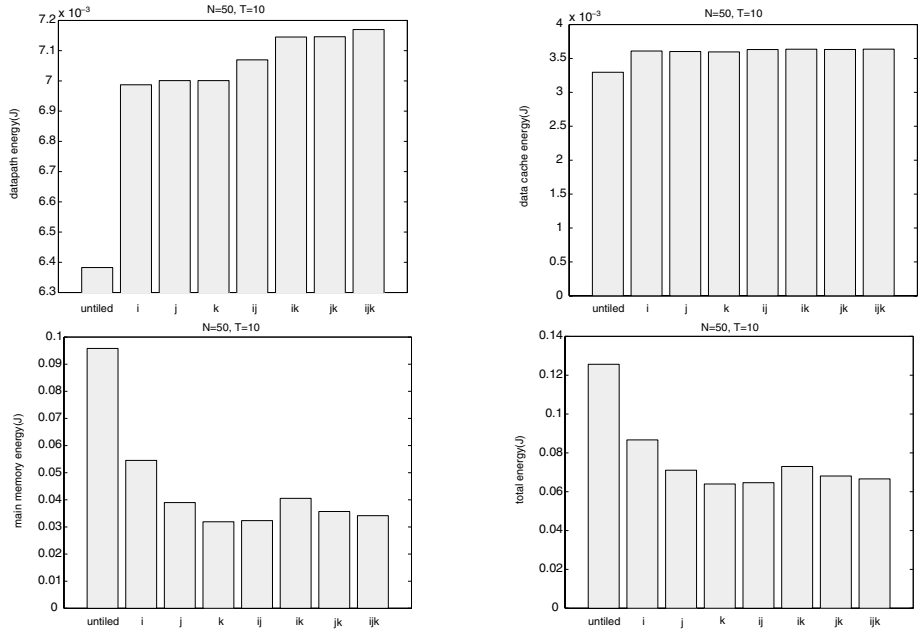


Fig. 1. Energy consumptions of different tiling strategies.

3.2 Sensitivity to the Tile Size and Input Size

While the tile size sensitivity issue has largely been addressed in performance-oriented studies [14, 4, 7, 15, 18], the studies that look at the problem from energy perspective are few [21]. Thus, we explored the influence of tile size on energy consumption. We summarize the key observations from our experiments [12]:

- Increasing the tile size reduces the datapath energy and instruction cache energy. This is because a large tile size (blocking factor) means smaller loop (code) overhead. However, as in the previous set of experiments, the overall energy behavior is largely determined by the energy spent in main memory.
- There is little change in data cache energy as we vary the tile size. This is because the number of accesses to the data cache is almost the same for all tile sizes (in a given tiling strategy).
- The energy consumption of a given tiling strategy depends to a large extent on the tile size. For instance, when N is 50 and both j and k loops are tiled, the overall energy consumption can be as small as 0.064J or as large as 0.128J depending on the tile size chosen. It was observed that, for each version of tiled code, there is a most suitable tile size beyond which the energy consumption starts to increase.
- For a given version, the best tile size from energy point of view was different from the best tile size from the performance (*execution cycles*) point of view. For example, as far as the execution cycles are concerned, the best tile size for the `ik` version was 10 (instead of 5).

Next, we varied the input size (N) and observed that *for each input size there exists a best tile size from energy point of view.* To be specific, the best possible tile sizes (among the ones we experimented with `ijk` version) for input sizes of 50, 100, 200, 300, and 400 are 10, 20, 20, 15, and 25, respectively. Thus, it is important for researchers to develop optimal tile size detection algorithms (for energy) similar to the algorithms proposed for detecting best tile sizes for performance (e.g., [4, 14, 24, 7]).

3.3 Sensitivity to the Cache Configuration

In this subsection, we evaluate the data cache energy consumption when the underlying cache configuration is modified. We experiment with different cache sizes and associativities as well as two energy-efficient cache architectures, namely, block buffering [22, 8] and sub-banking [22, 8].

In the block buffering scheme, the previously accessed cache line is buffered for subsequent accesses. If the data within the same cache line is accessed on the next data request, only the buffer needs to be accessed. This avoids the unnecessary and more energy consuming access to the entire cache data array. Thus, increasing temporal locality of the cache line through compiler techniques such as loop tiling can save more energy. In the cache sub-banking optimization,

the data array of the cache is divided into several sub-banks and only the sub-bank where the desired data is located is accessed. This optimization reduces the per access energy consumption and is *not* influenced by locality optimization techniques. We also evaluate cache configurations that combine both these optimizations. In such a configuration with block buffering and sub-banking, each sub-bank has an individual buffer. Here, the scope for exploiting locality is limited as compared to applying only block buffering as the number of words stored in a buffer is reduced. However, it provides the additional benefits of sub-banking for each cache access.

We first focus on traditional cache model and present in Figure 2 the energy consumed *only* in data cache for different cache sizes and associativities. We experiment with two different codes (with N=200), the untiled version and a blocked version where all three loops (i, j, and k) are tiled with a tile size of twenty. Our first observation is that the data cache energy is not too sensitive to the associativity but, on the other hand, is very sensitive to the cache size. This is because for a given code, the number of read accesses to the data cache is constant and, the cache energy per data access is higher for a larger cache. Increasing associativity also increases per access cost for cache (due to increased bit line and word line capacitances), but its effect is found to be less significant as compared to the increase in bit line capacitance due to increased cache sizes. As a result, embedded system designers need to determine minimum data cache size for the set of applications in question if they want to minimize data cache energy. Another observation is that for all cache sizes and associativities going from the untiled code to tiled code increases the data cache energy.

We next concentrate on cache line size and vary it between 8 bytes and 64 bytes for N=200 and T=50. The energy consumption of the `ijk` version for line sizes of 8, 16, 32, and 64 bytes were 0.226J, 0.226J, 0.226J, and 0.227J, respectively, indicating that (for this code) the energy consumption in data cache is relatively independent from the line size. It should also be mentioned that while increases in cache size and degree of associativity might lead to increases in data cache energy, they generally reduce the overall memory system energy by reducing the number of accesses to the main memory.

Finally, we focus on block buffering and sub-banking, and in Figure 3 give the data cache energy consumption for different combinations of block buffering (denoted `bb`) and sub-banking (denoted `sb`) for both the untiled and tiled (the `ijk` version) codes. *The results reveal that for the best energy reduction block buffering and sub-banking should be used together.* When used alone, neither sub-banking nor block buffering is much effective. The results also show that increasing the number of block buffers does not bring any benefit (as there is only one reference with temporal locality in the innermost loop). It should be noted that the energy increase caused by tiling on data cache can (to some extent) be compensated using a configuration such as `bb+sb`.

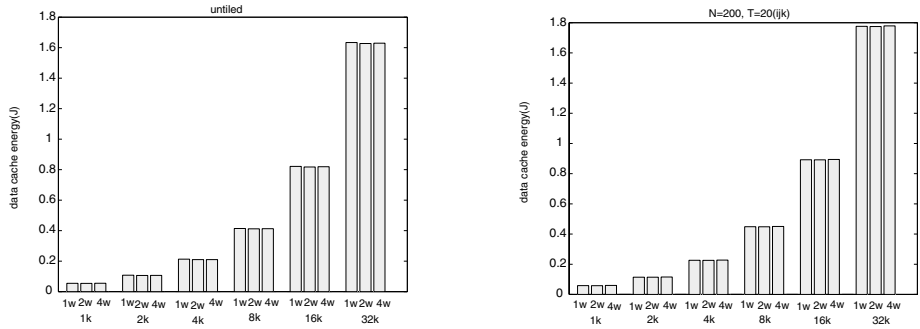


Fig. 2. Impact of cache size and associativity on data cache energy.

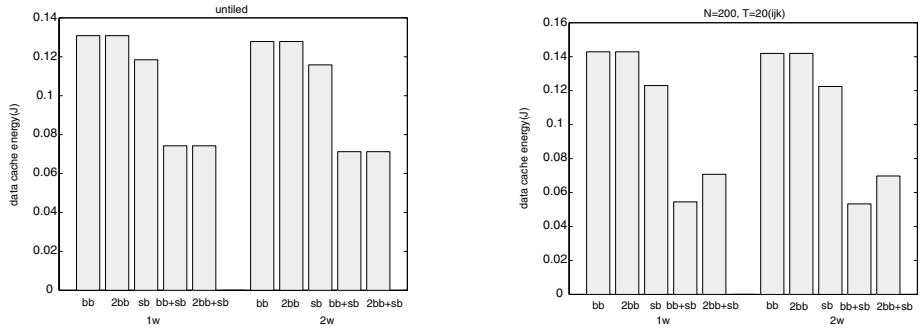


Fig. 3. Impact of block buffering (`bb`) and sub-banking (`sb`) on data cache energy.

3.4 Cache Miss Rates Vs. Energy

We now investigate the relative variations in miss rate and energy performance due to tiling. The following three measures are used to capture the correlation between the miss rates and energy consumption of the unoptimized (original) and optimized (tiled) codes.

$$\texttt{Improvement}_{\texttt{m}} = \frac{\text{Miss rate of the original code}}{\text{Miss rate of the optimized code}},$$

$$\texttt{Improvement}_{\texttt{e}} = \frac{\text{Memory energy consumption of the original code}}{\text{Memory energy consumption of the optimized code}},$$

$$\texttt{Improvement}_{\texttt{t}} = \frac{\text{Total energy consumption of the original code}}{\text{Total energy consumption of the optimized code}}.$$

In the following discussion, we consider four different cache configurations: 1K, 1-way; 2K, 4-way; 4K, 2-way; and 8K, 8-way. Given a cache configuration, Table 1 shows how these *three* measures vary when we move from the original version to an optimized version.

Table 1. Improvements in miss rate and energy consumption.

	1K, 1-way	2K, 4-way	4K, 2-way	8K, 8-way
$\texttt{Improvement}_\texttt{m}$	6.21	63.31	20.63	19.50
$\texttt{Improvement}_\texttt{e}$	2.13	18.77	5.75	2.88
$\texttt{Improvement}_\texttt{t}$	1.96	9.27	3.08	1.47

We see that in spite of very large reductions in miss rates as a result of tiling, the reduction in energy consumption is not as high. Nevertheless, it still follows the miss rate. We also made the same observation in other codes we used. We have found that $\texttt{Improvement}_\texttt{e}$ is smaller than $\texttt{Improvement}_\texttt{m}$ by a factor of 2 - 15. Including the datapath energy makes the situation worse for tiling (from the energy point of view), as this optimization in general increases the datapath energy consumption. Therefore, *compiler writers for energy aware systems can expect an overall energy reduction as a result of tiling, but not as much as the reduction in the miss rate.* We believe that some optimizing compilers (e.g., [20]) that estimate the number of data accesses and cache misses statically at compile time can also be used to estimate an approximate value for the energy variation. This variation is mainly dependent on the energy cost formulation parameterized by the number of hits, number of misses, and cache parameters.

3.5 Interaction with Other Optimizations

In order to see how loop tiling gets affected by other loop optimizations, we perform another set of experiments where we measure the energy consumption of tiling with linear loop optimization (loop interchange [26] to be specific) and loop unrolling [26]. Loop interchange modifies the original order of loops to obtain better cache performance. In our matrix-multiply code, this optimization converts the original loop order i, j, k (from outermost to innermost) to i, k, j, thereby obtaining spatial locality for arrays b and c, and temporal locality for array a, all in the innermost loop. We see from Figure 4 that tiling (in general) reduces the overall energy consumption of even this optimized version of the matrix-multiply nest. Note however that it increases the datapath energy consumption. Comparing these graphs with those in Figure 1, we observe that interchanged tiled version performs better than the pure tiled version, which suggests that *tiling should be applied in general after linear loop transformations for the best energy results.*

The interaction of tiling with loop unrolling is more complex. Loop unrolling reduces the iteration count by doing more work on a single loop iteration. We see from Figure 4 that untiled loop unrolling may not be a good idea as its energy consumption is very high. Applying tiling brings the energy consumption down. Therefore, *in circumstances where loop unrolling must be applied* (*e.g., to promote register reuse and/or to improve instruction level parallelism*), *we suggest to apply tiling as well to keep the energy consumption under control.*

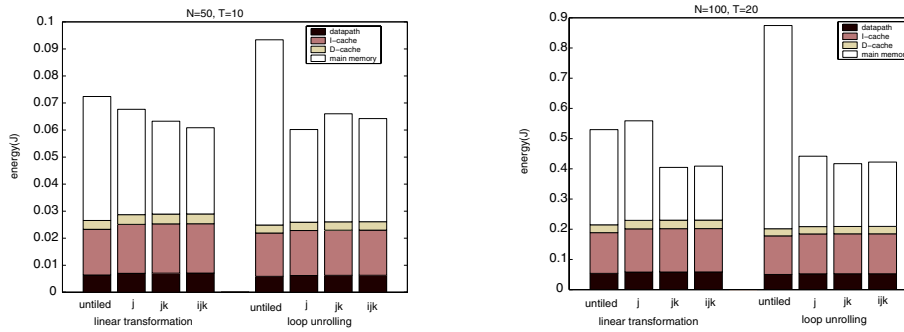


Fig. 4. Interaction of loop tiling with loop interchange and unrolling.

3.6 Analysis of Datapath Energy

We next zoom-in on the datapath energy, and investigate the impact of tiling on different components of the datapath as well as on different stages of the pipeline. Table 2 shows the breakdown of the datapath energy for the matrix-multiply code into different hardware components. For comparison purposes, we also give the breakdown for two other optimizations, loop unrolling (denoted `u`) and linear loop transformation (denoted `l`), as well as different combinations of these two optimizations and tiling (denoted `t`), using an input parameter of N=100. Each entry in this table gives the percentage of the datapath energy expended in the specific component. *We see that (across different versions) the percentages remain relatively stable. However, we note that all the optimizations increase the (percentage of) energy consumption in functional units due to more complex loop nest structures that require more computation in the ALU. The most significant increase occurs with tiling, and it is more than 26%.* These results also tell us that most of the datapath energy is consumed in register files and pipeline registers, and therefore the hardware designers should focus more on these units. Table 3, on the other hand, gives the energy consumption breakdown across five pipeline stages (the fetch stage `IF`, the instruction decode stage `ID`, the execution stage `EXE`, the memory access stage `MEM`, and the write-back stage `WB`). The entries under the `MEM` and `IF` stages here do not involve the energy consumed in data and instruction cache memory, respectively. We observe that most of the energy is spent in the `ID`, `EXE`, and `WB` stages. Also, the compiler optimizations in general increase the energy consumption in the `EXE` stage since that is the where the ALU sits; this increase is between 1% and 8% and also depends on the program being run.

3.7 Sensitivity to Technology Changes

The main memory has been a major performance bottleneck and has attracted a lot of attention [16]. Changes in process technology have made possible to embed a DRAM within the same chip as the processor core. Initial results using

Table 2. Datapath energy breakdown (in %s) in hardware components level.

Version	Register File	Pipeline Registers	Functional Units	Data-path Muxes
unoptimized	35.99	36.33	15.76	8.36
l	36.09	34.87	17.34	8.11
u	36.19	36.17	15.98	8.31
t	34.60	33.56	19.93	7.80
l+u	35.87	34.12	18.19	7.93
l+t	35.27	33.74	19.25	8.17
t+u	35.31	35.07	17.89	8.06
t+l+u	35.41	34.15	18.38	7.96

Table 3. Datapath energy breakdown (in %s) in pipeline stage level.

Version	IF	ID	EXE	MEM	WB
unoptimized	3.33	22.94	33.17	8.70	31.87
l	3.10	23.88	34.20	8.32	30.50
u	3.18	23.93	33.47	8.63	30.78
t	3.25	24.04	35.91	7.95	28.85
l+u	2.97	24.83	34.81	8.13	29.27
l+t	2.95	23.23	35.73	8.07	30.02
t+u	3.15	23.61	34.78	8.34	30.12
t+l+u	2.95	24.63	35.02	8.14	29.26

embedded DRAM (eDRAM) show an order of magnitude reduction in the energy expended in main memory [16]. Also, there have been significant changes in the DRAM interfaces [5] that can potentially reduce the energy consumption. For example, unlike conventional DRAM memory sub-systems that have multiple memory modules that are active for servicing data requests, the direct RDRAM memory sub-system delivers a full bandwidth with only one RDRAM module active. Also, based on the particular low power modes that are supported by the memory chips and based on how effectively they are utilized, the average per access energy cost for main memory can be reduced by up to two orders of magnitude [6].

In order to study the influence of changes in $\mathtt{E_m}$ due to these technology trends, we experiment with different $\mathtt{E_m}$ values that range from 4.95×10^{-9} (our default value) to 2.475×10^{-11}. We observe from Figure 5 that from $\mathtt{E_m} = 4.95 \times 10^{-9}$ on, the main memory energy starts to lose its dominance and instruction cache and datapath energies constitute the largest percentage. While this is true for both tiled and untiled codes, the situation in tiled codes is more dramatic as can be seen from the figure. For instance, when $\mathtt{E_m} = 2.475 \times 10^{-10}$, N=100, and T=10, the datapath energy is nearly 5.7 times larger than the main memory energy (which includes the energy spent in both data and instruction accesses), and the Icache energy is 13.7 times larger than the main memory energy. With the untiled code, however, these values are 0.98 and 2.43, respectively. *The challenge for future compiler writers for power aware systems then is to use tiling judiciously so that the energy expended in datapath and on-chip caches can be kept under control.*

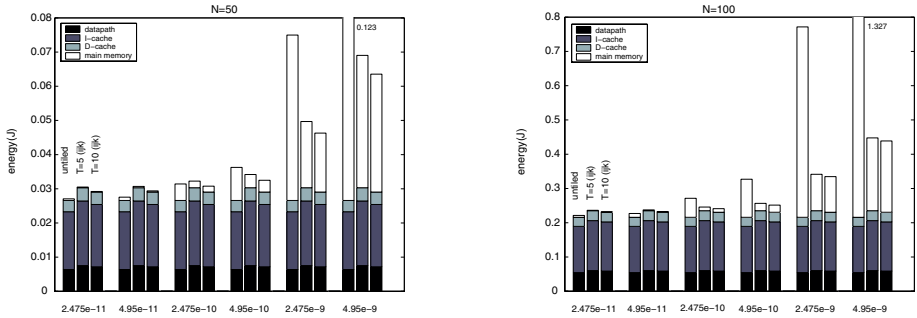


Fig. 5. Energy consumption with different $\mathtt{E_m}$ (J) values.

3.8 Other Codes

In order to increase our confidence in our observations on the matrix-multiply code, we also performed tiling experiments using several other loop nests that manipulate multi-dimensional arrays. The salient characteristics of these nests are summarized in Table 4. The first four loop nests in this table are from the Spec92/Nasa7 benchmark suite; `syr2k.1` is from Blas; and `htribk.2` and `qzhes.4` are from the Eispack library. For each nest, we used several tile sizes, input sizes, and per memory access costs, and found that the energy behavior of these nests are similar to that of the matrix-multiply. However, due to lack of space, we report here only the energy break-down of the untiled and two tiled codes (in a *normalized* form) using two representative $\mathtt{E_m}$ values (4.95×10^{-9} and 2.475×10^{-11}). In Figure 6, for each code, the three bars correspond to the untiled, tiled (`ijk`,T=5), and tiled (`ijk`,T=10) versions, respectively, from left to right. Note that while the main memory energy dominates when $\mathtt{E_m}$ is 4.95×10^{-9}, the instruction cache and datapath energies dominate when $\mathtt{E_m}$ is 2.475×10^{-11}.

Table 4. Benchmark nests used in the experiments. The number following the name corresponds to the number of the nest in the respective code.

nest	arrays	data size	tile sizes
`btrix.4`	two 4-D	21.0 MB	10 and 20
`vpenta.3`	one 3-D and five 2-D	16.6 MB	20 and 40
`cholesky.2`	one 3-D	10 MB	10 and 20
`emit.4`	one 2-D and one 1-D	2.6 MB	50 and 100
`htribk.2`	three 2-D	72 KB	12 and 24
`syr2k.1`	three 2-D	84 KB	8 and 16
`qzhes.4`	one 2-D	160 KB	15 and 30

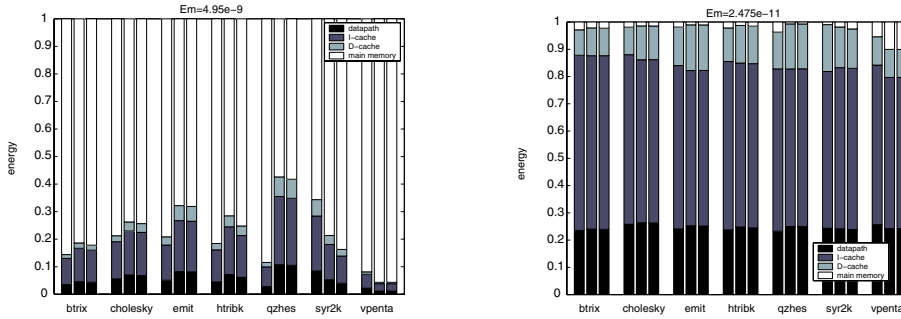


Fig. 6. Normalized energy consumption of example nested loops with two different $\mathtt{E_m}$ (J) values.

4 Related Work

Compiler researchers have attacked the locality problem from different perspectives. The works presented in Wolf and Lam [24], Li [15], Coleman and McKinley [4], Kodukula et al. [13], Lam et al. [14], and Xue and Huang [23], among others, have suggested tiling as a means of improving cache locality. In [24] and [15], the importance of linear locality optimizations before tiling is emphasized. While all of these studies have focussed on the performance aspect of tiling, in this paper, we investigate its energy behavior and show that the energy behavior of tiling may depend on a number of factors including tile size, input size, cache configuration, and per memory access cost.

Shiue and Chakrabarti [21] presented a memory exploration strategy based on three metrics, namely, processor, cycles, cache size, and energy consumption. They have found that increasing tile size and associativity reduces the number of cycles but does not necessarily reduce the energy consumption. In comparison, we focus on the entire system (including datapath and instruction cache) and study the impact of a set of parameters on the energy behavior of tiling using several tiling strategies. We also show that datapath energy consumption due to tiling might be more problematic in the future, considering the current trends in memory technology. The IMEC group [2] was among the first to work on applying loop transformations to minimize power dissipation in data dominated embedded applications.

In this paper, we utilize the framework that was proposed in [27]. This framework has been used to investigate the energy influence of a set of high-level compiler optimizations that include tiling, linear loop transformations, loop unrolling, loop fusion, and distribution [11]. However, the work in [11] accounts for only the energy consumed in data accesses and does not investigate tiling in detail. In contrast, our current work looks at tiling in more detail, investigating different tiling strategies, influence of varying tile sizes, and the impact of input sizes. Also, in this paper, we account for the energy consumed by the entire system including the instruction accesses. While the work in [12] studies differ-

ent tiling strategies, this paper focuses on impact of different tiling parameters, the performance versus energy consumption impact, and the interaction of tiling with other high-level optimizations.

5 Conclusions

When loop nest based computations process large amounts of data that do not fit in cache, tiling is an effective optimization for improving performance. While previous work on tiling has focussed exclusively on its impact on performance (execution cycles), it is critical to consider its impact on energy as embedded and mobile devices are becoming the tools for mainstream computation and start to take the advantage of high-level and low-level compiler optimizations.

In this paper, we study the energy behavior of tiling considering both the entire system and individual components such as datapath, caches, and main memory. Our results show that the energy performance of tiling is very sensitive to input size and tile size. In particular, selecting a suitable tile size for a given computation involves tradeoffs between energy and performance. We find that tailoring tile size to the input size generally results in lower energy consumption than working with a fixed tile size. Since the best tile sizes from the performance point of view are not necessarily the best tile sizes from the energy point of view, we suggest experimenting with different tile sizes to select the most suitable one for a given code, input size, and technology parameters. Also, given the current trends in memory technology, we expect that the energy increase in datapath due to tiling will demand challenging tradeoffs between prolonging battery life and limiting energy dissipated within a package.

References

[1] D. Burger, T. Austin, and S. Bennett. Evaluating future microprocessors: the SimpleScalar tool set. *Technical Report CS-TR-96-103,* Computer Science Dept., University of Wisconsin, Madison, July 1996.

[2] F. Catthoor, F. Franssen, S. Wuytack, L. Nachtergaele, and H. DeMan. Global communication and memory optimizing transformations for low power signal processing systems. In Proc. *the IEEE Workshop on VLSI Signal Processing,* pages 178–187, 1994.

[3] R. Y. Chen, R. M. Owens, and M. J. Irwin. Validation of an architectural level power analysis technique. In Proc. *the 35th Design Automation Conference (DAC),* June 1998.

[4] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In Proc. *the SIGPLAN'95 Conference on Programming Language Design and Implementation,* La Jolla, CA, June 1995.

[5] V. Cuppu, B. Jacob, B. Davis, and T. Mudge. A performance comparison of contemporary DRAM architectures. In Proc.*the 26th Int. Symp. Computer Architecture (ISCA),* May 1999, pp. 222–233.

[6] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin. Memory energy management using software and hardware directed power

mode control, *Technical Report CSE-00-004,* Dept. of Computer Science and Engineering, Penn State University, April 2000.

[7] K. Esseghir. *Improving Data Locality for Caches.* Master's Thesis, Dept. of Computer Science, Rice University, September 1993.

[8] K. Ghose and M. B. Kamble. Reducing power in superscalar processor caches using subbanking, multiple line buffers, and bit-line segmentation. In Proc. *1999 International Symposium Low Power Electronics and Design,* 1999, pages 70–75.

[9] M. Kamble and K. Ghose. Analytical energy dissipation models for low power caches. In Proc. *the International Symposium on Low Power Electronics and Design*, page 143, August 1997.

[10] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Improving locality using loop and data transformations in an integrated framework. In Proc. *the International Symposium on Microarchitecture,* Dallas, TX, December, 1998

[11] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye. Influence of compiler optimizations on system power. In Proc. *the 37th Design Automation Conference (DAC),* Los Angeles, California USA, June 5–9, 2000.

[12] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and H. S. Kim. Towards energy-aware iteration space tiling, In Proc. *the Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES),* Vancouver, B.C., June, 2000.

[13] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multilevel blocking. In Proc. *the SIGPLAN Conf. Programming Language Design and Implementation*, June 1997.

[14] M. Lam, E. Rothberg, and M. Wolf. The cache performance of blocked algorithms. In Proc. *the 4th Int. Conf. Arch. Supp. for Prog. Lang. & Oper. Sys.,* April 1991.

[15] W. Li. *Compiling for NUMA Parallel Machines.* Ph.D. Thesis, Computer Science Department, Cornell University, Ithaca, NY, 1993.

[16] Y. Nunomura et.al. M32R/D–Integrating DRAM and microprocessor, *IEEE MICRO*, November/December 1997

[17] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In Proc. *the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

[18] G. Rivera and C.-W. Tseng. A comparison of compiler tiling algorithms. In Proc. *the 8th International Conference on Compiler Construction (CC'99),* Amsterdam, The Netherlands, March 1999.

[19] K. Roy and M. Johnson. Software power optimization. In *Low Power Design in Deep Sub-micron Electronics,* Kluwer Academic Press, October 1996.

[20] V. Sarkar, G. R. Gao, and S. Han. Locality analysis for distributed shared-memory multiprocessors. In Proc. *the Ninth International Workshop on Languages and Compilers for Parallel Computing,* Santa Clara, California, August 1996.

[21] W-T. Shiue and C. Chakrabarti. Memory exploration for low power, embedded systems. In Proc. *the 36th Design Automation Conference (DAC),* New Orleans, Louisiana, 1999.

[22] C.-L. Su and A. M. Despain, Cache design trade-offs for power and performance optimization: A case study, In Proc. *ISLPED,* pp. 63–68, 1995.

[23] J. Xue and C.-H. Huang. Reuse-driven tiling for data locality. In *Languages and Compilers for Parallel Computing*, Z. Li et al., Eds., Lecture Notes in Computer Science, Volume 1366, Springer-Verlag, 1998.

[24] M. Wolf and M. Lam. A data locality optimizing algorithm. In Proc. *the ACM SIGPLAN 91 Conf. Programming Language Design and Implementation*, pages 30–44, June 1991.

[25] M. Wolf, D. Maydan, and D. Chen. Combining loop transformations considering caches and scheduling. In Proc. *International Symposium on Microarchitecture,* pages 274–286, Paris, France, December 1996.

[26] M. Wolfe. *High Performance Compilers for Parallel Computing*, Addison Wesley, CA, 1996.

[27] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. The design and use of SimplePower: A cycle-accurate energy estimation tool. In Proc. *the 37th Design Automation Conference (DAC),* Los Angeles, California USA, June 5–9, 2000.

Improving Offset Assignment for Embedded Processors

Sunil Atri[1], J. Ramanujam[1], and Mahmut Kandemir[2]

[1] Dept. Elec. & Comp. Engr., Louisiana State University
({sunil,jxr}@ee.lsu.edu)
[2] Dept. Comp. Sci. & Engr., The Pennsylvania State University
(kandemir@cse.psu.edu)

Abstract. Embedded systems consisting of the application program ROM, RAM, the embedded processor core, and any custom hardware on a single wafer are becoming increasingly common in application domains such as signal processing. Given the rapid deployment of these systems, programming on such systems has shifted from assembly language to high-level languages such as C, C++, and Java. The processors used in such systems are usually targeted toward specific application domains, e.g., digital signal processing (DSP). As a result, these embedded processors include application-specific instruction sets, complex and irregular data paths, etc., thereby rendering code generation for these processors difficult. In this paper, we present new code optimization techniques for embedded fixed point DSP processors which have limited on-chip program ROM and include indirect addressing modes using post-increment and decrement operations. We present a heuristic to reduce code size by taking advantage of these addressing modes. Our solution aims at improving the offset assignment produced by Liao et al.'s solution. It finds a layout of variables in RAM, so that it is possible to subsume explicit address register manipulation instructions into other instructions as a post-increment or post-decrement operation. Experimental results show the effectiveness of our solution.

1 Introduction

With the falling cost of microprocessors and the advent of very large scale integration, more and more processing power is being placed in portable electronic devices [5, 8, 9, 12]. Such processors (in particular, fixed-point DSPs and micro-controllers) can be found, for example in audio, video, and telecommunications equipment and have severely limited amounts of memory for storing code and data, since the area available for ROM and RAM is limited. This renders the efficient use of memory area very critical. Since the program code resides in the on-chip ROM, the size of the code directly translates into silicon area and hence the cost. The minimization of code size is, therefore, of considerable importance [1, 2, 4, 5, 6, 7, 8, 13, 14, 15, 16], while simultaneously preserving high levels of performance. However, current compilers for fixed-point DSPs generate code that is quite inefficient with respect to code size and performance. As a result, most application software is hand-written or at least hand-optimized, which is a very time consuming task [7]. The increase in developer productivity can therefore be directly linked to improvement in compiler techniques and optimizations.

S.P. Midkiff et al. (Eds.): LCPC 2000, LNCS 2017, pp. 158–172, 2001.

Many embedded processor architectures such as the TI TMS320C25 include indirect addressing modes with *auto-increment* and *auto-decrement* arithmetic. This feature allows address arithmetic instructions to be part of other instructions. Thus, it eliminates the need for explicit address arithmetic instructions wherever possible, leading to decreased code size. The memory access pattern and the placement of variables has a significant impact on code size. The auto-increment and auto-decrement modes can be better utilized if the placement of variables is performed after code selection. This delayed placement of variables is referred to as *offset assignment.*

This paper considers the *Simple Offset Assignment* (SOA) problem where there is just one address register. A solution to the problem assigns optimal frame-relative offsets to the variables of a procedure, assuming that the target machine has a single indexing register with only the indirect, auto-increment and auto-decrement addressing modes. The problem is modeled as follows. A basic block [10] is represented by an *access sequence*, which is a sequence of variables written out in the order in which they are accessed in the high level code. This sequence is in turn further condensed into a graph called the *access graph* with weighted undirected edges. The SOA problem is equivalent to a graph covering problem, called the *Maximum Weight Path Cover* (MWPC) problem. A solution to the MWPC problem gives a solution to the SOA problem. We present a new algorithm, called $Incremental\text{-}Solve$-SOA, for the SOA problem and compare its performance with previous work on the topic.

The remainder of this paper is organized as follows. We present a brief explanation of graphs and some additional required notation and background in Section 2. Then, in Section 3, we consider the problem of storage assignment, where the arithmetic permitted on the address register is plus or minus 1. We present our experimental results in Section 4. We conclude the paper with a summary in Section 5.

2 Background

We model the sequence of data accesses as weighted undirected graphs [7]. Each variable in the program corresponds to a vertex (or node) in the graph. An edge i, j indicates that variable i is accessed after j or vice-versa; the weight of an edge $w(i, j)$ denotes the number of times variables i and j are accessed successively.

Definition 1 *Two paths are said to be disjoint if they do not share any vertices.*

Definition 2 *A* disjoint path cover *(which will be referred to as just a 'cover') of a weighted graph $G(V, E)$ is a subgraph $C(V, E')$ of G such that, for every vertex v in C, deg(v) < 3 and there are no cycles in C. The edges in C may be a non-contiguous set [3].*

Definition 3 *The* weight *of a cover C is the sum of the weights of all the edges in C [5]. The cost of a cover C is the sum of the weights of all edges in G but not in C:*

$$cost(C) = \sum_{(e \in E) \wedge (e \notin C)} w(e)$$

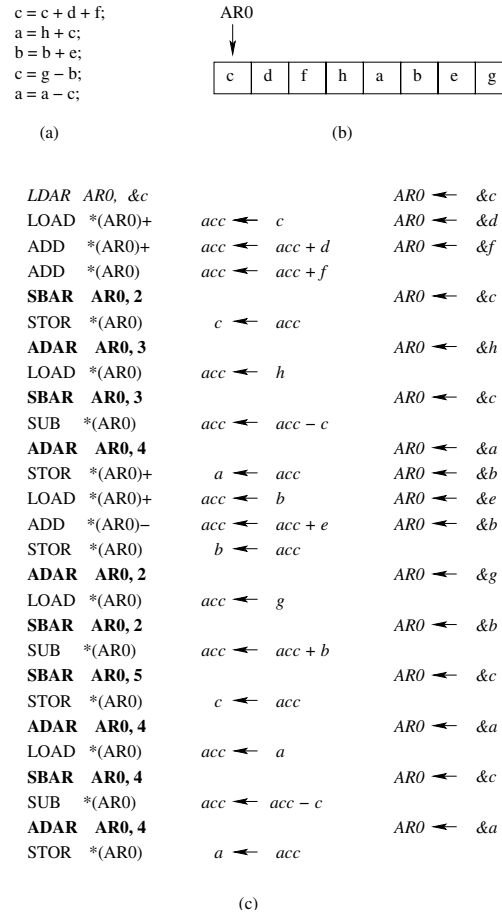


Fig. 1. Code example from [5,6].

2.1 Motivating Example

As mentioned earlier, many embedded processors provide register indirect addressing modes with auto-increment and auto-decrement arithmetic [5]. It is possible to use these modes for efficient sequential access of memory and improve code density. The placement of variables in memory has a large impact on the exploitation of auto-increment and auto-decrement addressing modes, which is in turn affected by the pattern in which variables are accessed. If the assignment of location of variables is done after code selection, then we get the freedom of assigning locations to variables depending on the order in which the variables are accessed. The placement of variables in storage has a considerable impact on code size and performance.

Consider the C code sequence shown in Figure 1(a), an example from [5]; let the placement of variables in memory be as in Figure 1(b). This assignment of variables to memory locations here is based on *first use*, i.e., as the variables are referred to in

the high level code, the variables are placed in memory. The assembly code for this section of C code is shown in Figure 1(c). The first column shows the assembly code, the second column shows the register transfer, and the third, the contents of the address pointer. The instructions in bold are the explicit address pointer arithmetic instructions, i.e., **SBAR**, Subtract Address Register and **ADAR**, *Add Address Register*. *The objective of the solution to the SOA problem is to find the minimal address pointer arithmetic instructions required using proper placement of variables in memory.* A brief explanation of Figure 1 follows.

The first instruction LDAR AR0, &c loads the *address* of the first variable 'c' into the address register AR0. The next instruction LOAD (AR0)+ loads the variable 'c' into the accumulator. This instruction shows the use of the auto-increment mode. Ordinarily, we would need an explicit pointer increment to get it to point to 'd' which is the next required variable, but it is subsumed into the LOAD instruction in the form of a post-increment operation, indicated by the trailing '+' sign. The pointer decrement operation can also be similarly subsumed by a post-decrement operation indicated by a trailing '-' sign for example as in the ADD *(AR0)- instruction. It can be seen, that the instructions in bold are the ones that do only address pointer arithmetic in Figure 1(a). *The number of such instructions in the generated code may be very high, as typically the high-level programmer does not consider the variable layout while writing the program.* AR0 is auto-incremented after the first LOAD instruction. Now, AR0 is pointing to 'd', as 'd' is the next variable required, so it can be accessed immediately without having to change AR0. Similarly for variable 'f', the next variable required is 'c', which is at a distance 2 from 'f'. Consider the STOR instruction that writes the result back to 'c', an explicit SBAR AR0, 2 instruction has to be used to set AR0 to point to 'c', because the address of 'f' and that of 'c' differ by two and auto-decrement cannot be used along with the previous ADD instruction. This can be seen in the other instances of ADAR and SBAR, where for every pair of accesses that do not refer to adjacent variables, either an SBAR or ADAR instruction must be used. In total, ten such instruction are needed to execute the code in Figure 1(a), given the offset assignment of Figure 1(b).

2.2 Assumptions in SOA

The simple offset assignment (SOA) problem is one of assigning a frame-relative offset to each local variable to *minimize* the number of address arithmetic instructions (ADAR and SBAR) required to execute a basic block. The cost of an assignment is hence defined by the number of such instructions. With a single address register, the initializing LDAR instruction is *not* included in this cost. We make the following assumptions for the SOA problem: (1) every data object is of size one word; (2) a single address register is used to address all variables in the basic block; (3) one-to-one mapping of variables to locations; (4) the basic block has a fixed evaluation order; (and) (5) special features such as address wrap-around are not considered.

2.3 Approach to the Problem

The SOA problem can be formulated as a *graph covering problem*, called the *Maximum Weight Path Covering Problem* (MWPC) [5, 6]. From a basic block, a graph, called the

access graph is derived, that shows the various variables and their relative adjacency and frequency of accesses. From the solution to the MWPC problem, a minimum cost assignment can be constructed.

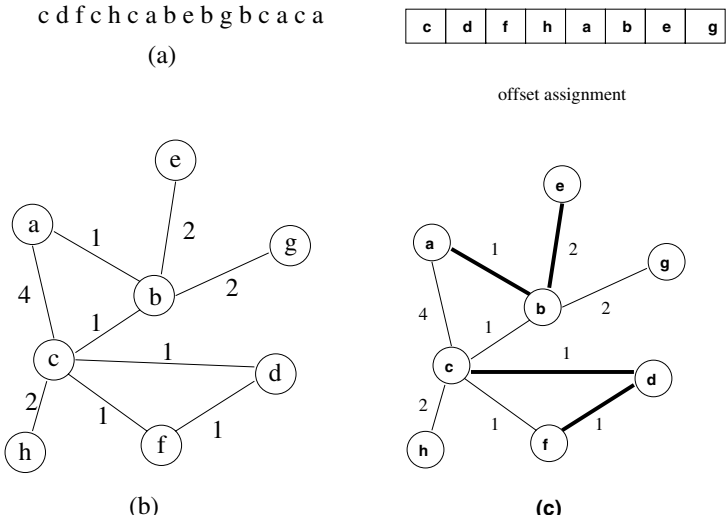


Fig. 2. (a) Access sequence; (b) Access graph; (c) Offset assignment and cover C (thick edges).

Given a code sequence S that represents a basic block, one can define a unique *access sequence* for that block [6]. In an operation '$z = x\ op\ y$', where 'op' is some binary operator, the access sequence is given by 'xyz'. The access sequence for an ordered set of operations is simply the concatenated access sequences for each operation in the appropriate order. For example, the access sequence for the C code example in Figure 1(a) is shown in Figure 2(a).

With the definition of cost given earlier, it can be seen that the cost is the number of consecutive accesses to variables that are not assigned to adjacent locations. The access sequence is the sequence of memory references made by a section of code and it can be obtained from the high-level language program. The access sequence can be summarized in an edge weighted, undirected graph. The access graph $G(V, E)$ is derived from an access sequence as follows. Each vertex $v \in V$ of the access graph corresponds to a unique variable in the basic block. An edge $e(u, v) \in E$ between vertices u and v exists with weight $w(e)$ if variables u and v are adjacent to each other $w(e)$ times in the access sequence. The order of the accesses is *not* significant as either auto-increment or auto-decrement can be performed. The access graph for Figure 2(a) is shown in Figure 2(b).

2.4 SOA and Maximum Weight Path Cover

Given the definitions earlier in this paper, if a maximum weight cover for a offset assignment graph is found, then that also means that the minimum cost assignment has also been found. Given a cover C of G the cost of every offset assignment implied by C is less than or equal to the cost of the cover [5]. Given an offset assignment A and an access graph G, there exists a disjoint path cover which implies A and which has the same cost as A. Every offset assignment implied by an optimal disjoint path cover is optimal. An example of a sub-optimal path cover is shown in Figure 2(c). The thick lines show the disjoint path cover and the corresponding offset assignment is also shown. The cost of this assignment is 10. This can be seen from the edges not in the cover.

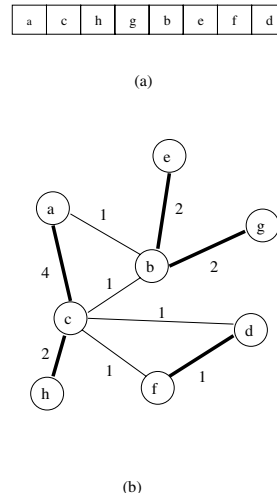


Fig. 3. Optimal offset assignment and cover C (thick edges).

3 An Incremental Algorithm for SOA

3.1 Previous Work

Bartley [2] and Liao [5, 6] studied the simple offset assignment problem. Liao formulated the simple offset assignment problem. The problem was modeled as a graph theoretic optimization problem similar to Bartley [2] and shown to be equivalent to the Maximum Weighted Path Cover (MWPC) problem. This problem is proven to be NP-hard. A heuristic solution to the above problem proposed by Liao will be explained in the following subsection. Consider the example shown earlier. Using Liao's algorithm we get an offset assignment as shown in Figure 3(a) which is implied by the access graph in Figure 3(b). The cover of the access graph is shown by the heavy edges, and in this case it is optimal. This can be seen from the graph itself. Picking any of the

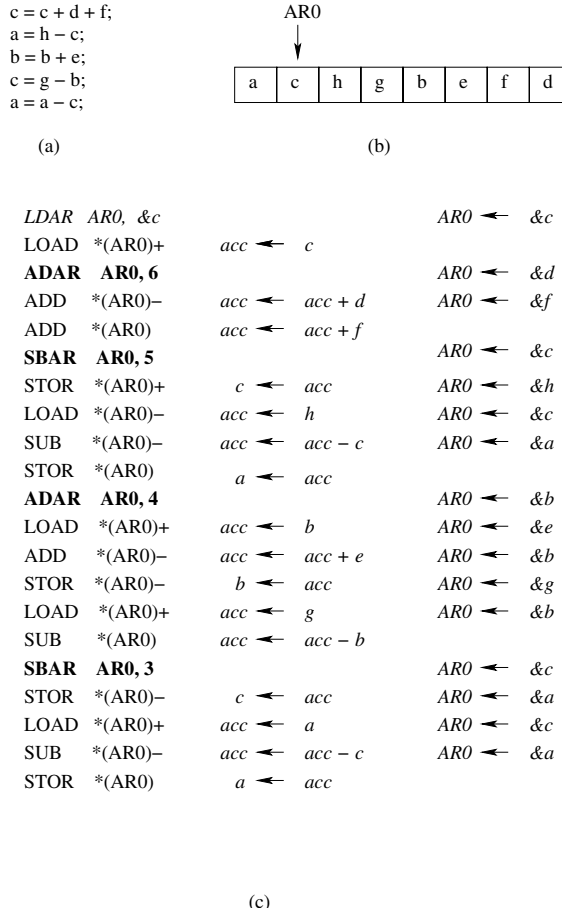


Fig. 4. Code after the optimized offset assignment.

four non-included edges will cause the dropping of some edge from the cover which will in turn increase the cost of the cover. The assembly code for the offset assignment implied by the cover is shown in Figure 4(c). The address arithmetic instructions are highlighted and there are four such instructions corresponding to the four edges not in the cover. For example because 'a' and 'b' could not be placed adjacent to each other, we need to use the instruction ADAR *(AR0) 4. The offset assignment that this section of code needs to use is shown in Figure 4(b), along with the C code (Figure 4(a)) for reference. Leupers and Marwedel [4] present a heuristic for choosing among different edges with the same weight in Liao's heuristic.

3.2 Liao's Heuristic for SOA

Because SOA and MWPC are NP-hard, a polynomial-time algorithm for solving these problems optimally is not likely to exist unless P=NP. Liao's heuristic for the simple

offset assignment problem is shown in Figure 5. This algorithm is similar to Kruskal's minimum spanning tree algorithm [3]. The heuristic is greedy, in the sense that it repeatedly selects the edge that seems best at the current moment.

```
1  // INPUT :  Access Sequence, L
2  // OUTPUT :  Constructed Assignment E'
3  Procedure Solve − SOA(L)
4  G(V, E) ← AccessGraph(L)
5  E_sort ← Sorted edges in E in descending order of weight
6  // Initialize C(V', E') the constructed cover
7  E' ← { }
8  V' ← V
9  while (|E'| < |V| − 1 and E_sort not empty) do
10   e ← first edge in E_sort
11   E_sort ← E_sort − e
12   if ((e does not cause a cycle in C) and
13      (e does not cause any vertex in V' to have degree > 2))
14      add e to E'
15   else
16      discard e from E_sort
17   endif
18 enddo
19 return E'
```

Fig. 5. Liao's maximum weight path cover heuristic [5].

Consider the algorithm $Solve$-SOA(L) in Figure 5. This algorithm takes as input a sequence 'L' which uniquely represents the high level code, and produces as output an offset assignment. In line 4, graph $G(V, E)$ is produced from the access sequence 'L'. Producing the access sequence takes $O(L)$ time. Line 5 produces a list of sorted edges in descending order of weight. $C(V', E')$ is the cover of the graph G, which starts with all the vertices included but no edges. The condition for the while statement makes sure that no more than $V - 1$ edges are selected, as that is the maximum needed for any cover. If the cover is disjoint, the order in which the disjoint paths are positioned does not matter as far as the cost of the offset assignment is concerned, because the cost of moving from one path to another will always have to be paid. The complexity of Liao's heuristic is $O(|E| \log |E| + |L|)$ [5], where $|E|$ is the number of edges in the access graph and $|L|$ is the length of the access sequence. Construction of the access sequence takes $O(|L|)$ time. The $(|E| \log |E|)$ term is due to the need to sort the edges in descending order of weight. The main loop of the algorithm runs for $|V|$ iterations. The test for a cycle in line 12 takes constant time, and the total time for the main loop is bounded by $O(E)$. The test for a cycle is achieved in constant time by using a special data structure proposed by Liao [5].

3.3 Improvement over Solve-SOA

Before suggesting the improvement, we want to point out two deficiencies in $Solve$-SOA. First, even though the edges are sorted in descending order of weight, the order of consideration of the edges of the same weight are ordered is not specified. We believe this to be important in deriving the optimal solution. Second, the maximum weight edge is always selected since this is a greedy approach. The proposed $Incremental$-$Solve$-SOA heuristic addresses both these cases. This algorithm takes as input an offset sequence, produced either by Liao's $Solve$-SOA or by some other means, and tries to include edges not previously included in the cover. Consider the example of Figure 6. The code sequence is shown in Figure 6(a) and the corresponding access sequence is in Figure 6(b). The access graph which in turn corresponds to this access sequence is shown in Figure 6(c). Let us now run Liao's $Solve$-SOA using the access sequence in Figure 6(b); one possible outcome is shown in Figure 7(a). The offset assignment associated with the cover is a, d, b, c, e or d, b, c, e, a. This is clearly a non-optimal solution. The cost of this assignment is 2. The optimal solution would be d, b, a, c, e. It is possible to have achieved the optimal cost of 1 by having considered either edge (a, b) or edge (a, c) before edge (b, c). But since $Solve$-SOA does not consider the relative positioning of the edges of the same weight in the graph, we get the cost of 2. The solution that is produced by the proposed $Incremental$-$Solve$-SOA is d, b, a, c, e as shown in Figure 7(b).

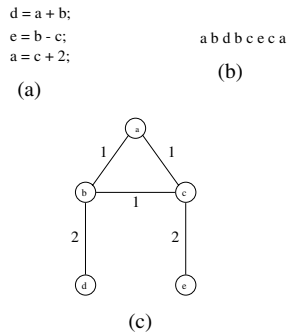


Fig. 6. An example where $Solve$-SOA could possibly return suboptimal results.

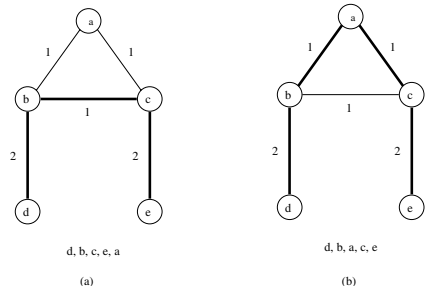


Fig. 7. Suboptimal and optimal cover of G.

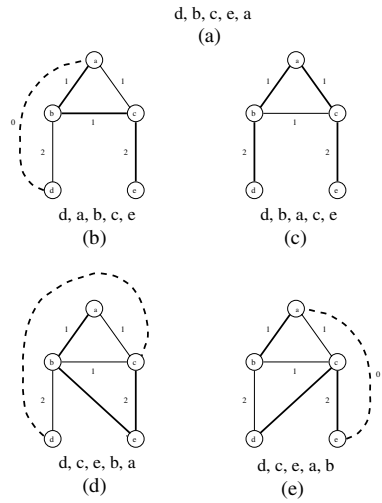


Fig. 8. Four different offset assignments.

Incremental-Solve-SOA Figure 9 shows the proposed algorithm. The algorithm picks the maximum weight edge not included in the cover and tries to include that. This is done as follows. Let the maximum weight edge not included in the cover be between two variables $a(n)$ and $a(n+x)$, in that order. We consider the case where we try to include that edge and see the effect on the cost of an assignment. There are four offset assignments when we try to bring two variables together not previously adjacent. The initial offset assignment is $...a(n-1)a(n)a(n+1)...a(n+x-1)a(n+x)a(n+x+1)....$ We consider the following four sequences that would result when edge $(a(n)a(n+x))$ is included in the cover:

(1) $...a(n-1)a(n+x)a(n)a(n+1)...a(n+x-1)a(n+x+1)...$
(2) $...a(n-1)a(n)a(n+x)a(n+1)...a(n+x-1)a(n+x+1)...$
(3) $...a(n-1)a(n+1)...a(n+x-1)a(n)a(n+x)a(n+x+1)...$
(4) $...a(n-1)a(n+1)...a(n+x-1)a(n+x)a(n)a(n+x+1)...$

The cost of each of these is evaluated and the best assignment, i.e., the one with the least cost of the four is chosen for the next iteration. A running minimum cost assignment, BEST, is used to store the best assignment discovered. This is returned at the end of the procedure.

Theorem 1 *The $Incremental$-$Solve$-SOA will either improve or return the same cost assignment.*

$Proof$: As different assignments with different costs are produced, a running minimum is maintained. If the minimum is the initial assignment that is the one considered again, and finally returned when all the edges are locked, or there are no non-zero edges available for inclusion. □

```
1  // INPUT :  Access Sequence AS, Initial Offset Assignment, OA
2  // OUTPUT :  Final Offset Assignmnet
3  Procedure Incremental-Solve-SOA(AS, OA)
4  G = (V, E) ← AccessGraph(AS)
5  BEST ← Initial offset assignment OA
6  repeat
7    E^U_sort ← Sorted list of unselected edges from BEST configuration
8    OUTER_FLAG ← FALSE
9    Unlock all edges in E^U_sort
10   INNER_BEST ← BEST
11   repeat
12     INNER_FLAG ← FALSE
13     e ← topmost edge from E^U_sort
14     (A_0, ..., A_3) ← The four possible assignments due to e
15     // An assignment is illegal if it involves changing a locked edge;
16     // Otherwise, an assignment is legal
17     S ← the set of legal assignments from (A_0, ..., A_3)
18     if (S has at least one legal assignment)
19       INNER_FLAG ← TRUE
20       CURRENT ← MinCost(S)
21       lock the edges that change
22       Delete the locked edges from E^U_sort ensuring that E^U_sort stays sorted
23       if (CostOf(CURRENT) < CostOf(INNER_BEST))
24         INNER_BEST ← CURRENT
25       endif
26     else (E^U_sort ≠ φ)
27       INNER_FLAG ← TRUE
28     endif
29   until (INNER_FLAG ≠ TRUE)
30   if (CostOf(INNER_BEST) < CostOf(BEST))
31     BEST ← INNER_BEST
32     OUTER_FLAG ← TRUE
33   endif
34 until (OUTER_FLAG ≠ TRUE)
35 return BEST
```

Fig. 9. Incremental-Solve-SOA

In the example, the initial assignment is d, b, c, e, a, that has a cost of 2. Let us try to include edge (b, a). The resulting four assignments for the initial assignment of d, b, c, e, a with cost = 2 are:

(1) d, a, b, c, e cost = 3
(2) d, b, a, c, e cost = 1
(3) d, c, e, b, a cost = 4
(4) d, c, e, a, b cost = 4

These four assignments are shown in Figure 8(b)-(e). Figure 8(a) shows the initial offset assignment for the purpose of comparison.

Detailed Explanation of the Incremental-Solve-SOA The input is an access sequence and the initial offset assignment, that we will attempt to improve upon. The output is the possibly improved offset assignment. In line 4, we call the function $AccessGraph$ to obtain the access graph from the access sequence. BEST is a data structure that stores an offset assignment along with its cost. It is initialized with the input offset assignment in line 5. Lines 6 thru 34 is the outer loop. The exit condition for this loop is the OUTER_FLAG being set to FALSE. Line 7 produces $\mathsf{E}^{\mathsf{U}}_{\mathsf{sort}}$ holds the sorted list of edges present in the access graph but not in the cover, in decreasing order of weight. This is done so as to be able to consider edges for inclusion in the order of decreasing weight. The edges carry a flag that is used for 'locking' or 'unlocking' edge. The 'lock' on an edge, if broken, is used to indicate an edge available for inclusion in the cover. Lines 11 thru 29 form the inner loop. The exit condition for this loop the INNER_FLAG being FALSE. In line 13, the top most edge is extracted from $\mathsf{E}^{\mathsf{U}}_{\mathsf{sort}}$ and the four assignments are produced as explained in the earlier section on reordering of variables. These four assignments are stored in $(A_0, ..., A_3)$. The cover formed by each is checked to see if a locked edges not included earlier in being included, or if a locked edge included in earlier is being excluded. The assignments where this does not happen are included in S. This is done line 17. Of these the legal assignments are stored in the set S in line 17. In line 18 if there is at least one assignment available, set S is not empty, then the minimum cost one of those is assigned to CURRENT in line 20 and INNER_BEST is set to TRUE. The edges which undergo transitions as explained earlier are locked in line 21. INNER_BEST maintains a running minimum cost assignment for the inner loop, and if the CURRENT cost is less than INNER_BEST, then that is made the INNER_BEST. $\mathsf{E}^{\mathsf{U}}_{\mathsf{sort}}$ is reassigned for the list of unselected and unlocked edges from the CURRENT cover in line 22. If no legal assignments could be found for the edge extracted from $\mathsf{E}^{\mathsf{U}}_{\mathsf{sort}}$ in line 17, and there is at least another edge available for consideration, then the INNER_FLAG is set to TRUE. This is done in lines 26 and 27. Once there are no more legal assignments and there are no more edges in $\mathsf{E}^{\mathsf{U}}_{\mathsf{sort}}$ available, we exit the inner loop and check if the cost of INNER_BEST is less than the BEST found. If there is an improvement we perform the whole process of the inner loop all over again. This is made possible be setting OUTER_FLAG to TRUE. If no improvement was found, then we exit the outer loop too and the BEST offset assignment discovered is returned in line 35.

3.4 Complexity of Incremental-Solve-SOA

As mentioned before, the running time of Liao's $Solve$-SOA heuristic is $O(|E| \log |E| + |L|)$, where $|E|$ is the number of edges in the access graph and $|L|$ is the length of the access sequence. The running time of the $Incremental\text{-}Solve$-SOA is $O(|E|)$ for the inner while loop. Sorting the edges in descending order of weight for E^U_{sort} takes $O(|E| \log |E|)$ time,and the marking of the edges is $O(|E|)$, the number of iterations of the outer loop in our experience runs for a constant number of times, an average of

2. So, the complexity of each outer loop iteration in practice is $O(|E| \log |E|)$. This is the same as Liao's, though in practice we need to incur a higher overhead; but, the use of our heuristic is justified by the fact that the code produced by this optimization will be executed many times whereas the compilation is done only once. Also, its use could possibly result in a smaller ROM area.

4 Experimental Results

We implemented both $Solve$-SOA and $Incremental$-$Solve$-SOA. The results shown in Table 1 are for the case where the initial offset assignment used was the result of using Liao's heuristic (If the initial offset assignment is an unoptimized one, the improvements will be much higher).

Table 1. Results from $Incremental$-$Solve$-SOA as compared to $Solve$-SOA.

Number of Variables	Size of Access Sequence	% Cases Improved	% Improvement in the Improved Cases
5	10	2.4	37.08
5	20	4.6	19.00
8	16	4.0	17.13
8	30	7.4	8.71
8	40	6.0	6.85
10	50	7.8	4.87
10	100	5.4	2.41
15	20	4.8	12.66
15	30	4.4	7.46
15	56	5.4	3.29
15	75	5.4	2.37
20	40	2.8	5.38
20	75	4.0	2.71
20	100	3.4	1.92

The experiments were performed by generating *random access sequences* and using these as input to Liao's heuristic. The offset sequence returned was then used, in turn, along with the access sequence in the $Incremental$-$Solve$-SOA to produce a possible change in the offset sequence. This change is guaranteed to be the same or better as reflected in Table 1. The third column shows the percentage improvement in the number of cases is of relevance here, as it shows an improvement in the cost of the $cover$ of the access graph. It is always possible to increase all the edge weights in the access graph by some constant value to achieve a higher magnitude improvement for the same change in cover, but the change in cover would still be the same.

In Table 1, the first column lists the number of variables, the second column lists the size of the access sequence. The third column shows the average improvement in

the number of cases. That is, for example in the first row, there was an improvement on an average of 2.4% of all the random access sequences considered. The fourth column shows, of the improved cases, the extant of improvement. For the first row that would be 37.08% improvement in 2.5% of the cases.

The overall average improvement (in the third column) is 5.23%. This figure reflects the cases in which $Incremental$-$Solve$-SOA was able to improve upon the cover of the access graph given the offset assignment produced from Liao's $Solve$-SOA as input. The improvement takes significance from the many times the code would be executed, and also that it would result in a saving of ROM area.

5 Conclusions

Optimal code generation is important for embedded systems in view of the limited area available for ROM and RAM. Small reductions in code size could lead to significant changes in chip area and hence reduction in cost. We looked at the Simple Offset Assignment (SOA) problem and proposed a heuristic which, if given as input, the offset assignment from Liao or some other algorithm will attempt to improve on that. It does this by trying to include the highest weighted edges not included in cover of an access graph. The proposed heuristic is quite simple and intuitive. Unlike algorithms that are used in different computer applications, it is possible to justify a higher running time for an algorithm designed for a compiler (especially for embedded systems), as it is run once to produce the code, which is repeatedly executed. In the case of embedded systems, there is the added benefit of savings in ROM area, possibly reducing the cost of the chip.

In addition, the first author's thesis [1] addressed two important issues: the first one is the use of commutative transformations to change the access sequence and thereby reducing the code size; the second deals with exploiting those cases where the post-increment or decrement value is allowed to be greater than one. We are currently exploring several issues. First, we are looking at the effect of statement reordering on code density. Second, we are evaluating the effect of variable life times and static single assignment on code density. In addition, reducing code density for programs with array accesses is an important problem.

Acknowledgments

The work of J. Ramanujam is supported in part by an NSF grant CCR-0073800 and by NSF Young Investigator Award CCR–9457768.

References

[1] S. Atri. *Improved Code Optimization Techniques for Embedded Processors.* M.S. Thesis, Department of Electrical and Computer Engineering, Louisiana State University, December 1999.

[2] D. Bartley. Optimizing Stack Frame Accesses for Processors with Restricted Addressing Modes. *Software - Practice and Experience*, 22(2):101-110, Feb. 1992.

[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivist. *Introduction to Algorithms.* MIT Press, 1990.

[4] R. Leupers and P. Marwedel. Algorithms for address assignment in DSP code generation. In *Proc. International Conference on Computer Aided Design,* pages 109–112, Nov. 1996.

[5] S. Y. Liao, *Code Generation and Optimization for Embedded Digital Signal Processors*, Ph.D. Thesis. MIT, June 1996.

[6] S. Y. Liao, S. Devadas, K. Keutzer and S. Tjiang, and A. Wang. Storage Assignment to Decrease code Size Optimization. In *Proc. 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation.* pages 186-195, June 1995.

[7] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, A. Wang, G. Araujo, A. Sudarsanam, S. Malik, V. Zivojnovic and H. Meyr. Code Generation and Optimization Techniques for Embedded Digital Signal Processors. In *Hardware/Software Co-Design*, Kluwer Acad. Pub., G. De Micheli and M. Sami, Editors, 1995.

[8] P. Marwedel and G. Goossens, editors. *Code Generation for Embedded Processors*, Kluwer Acad. Pub., 1995.

[9] G. De Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill 1994.

[10] S. S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, San Francisco, CA, 1997.

[11] D. A. Patterson and J. L. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[12] P. G. Paulin, M. Cornero, C. Liem *et al. Trends in Embedded System Technology, an Industrial Perspective.* Hardware/Software Co-Design, M. Giovanni M. Sami, editors. Kluwer Acad. Pub., 1996.

[13] Amit Rao and Santosh Pande. Storage assignment optimizations to generate compact and efficient code on embedded DSPs. *Proc. 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation.* pages 128–138, June 1999.

[14] A. Sudarsanam, S. Liao, and S. Devadas. Analysis and Evaluation of Address Arithmetic Capabilities if Custom DSP Architectures. In *Proceedings of 1997 ACM/IEEE Design Automation Conference.* pages 297-292, 1997.

[15] A. Sudarsanam and S. Malik. Memory Bank and Register Allocation in Software Synthesis for ASIPs. In *Proceedings of 1995 International Conference on Computer-Aided Design.* pages. 388-392, 1995.

[16] A. Sudarsanam, S. Malik, S. Tjiang, and S. Liao. Optimization of Embedded DSP Programs Using Post-pass Data-flow Analysis. In *Proceedings of 1997 International Conference on Acoustics, Speech, and Signal Processing.*

Improving Locality for Adaptive Irregular Scientific Codes

Hwansoo Han and Chau-Wen Tseng

Department of Computer Science, University of Maryland, College Park, MD 20742

Abstract. Irregular scientific codes experience poor cache performance due to their memory access patterns. In this paper, we examine two issues for locality optimizations for irregular computations. First, we experimentally find locality optimization can improve performance for parallel codes, but is dependent on the parallelization techniques used. Second, we show locality optimization may be used to improve performance even for adaptive codes. We develop a cost model which can be employed to calculate an efficient optimization frequency; it may be applied dynamically instrumenting the program to measure execution time per time-step iteration. Our results are validated through experiments on three representative irregular scientific codes.

1 Introduction

As scientists attempt to model more complex problems, computations with irregular memory access patterns become increasingly important. These computations arise in several application domains. In computational fluid dynamics (CFD), meshes for modeling large problems are sparse to reduce memory and computations requirements. In n-body solvers such as those arising in molecular dynamics, data structures are by nature irregular because they model the positions of particles and their interactions.

In modern microprocessors, memory system performance begins to dictate overall performance. The ability of applications to exploit locality by keeping references to cache becomes a major (if not the key) factor affecting performance. Unfortunately, irregular computations have characteristics which make it difficult to utilize caches efficiently.

Consider the example in Figure 1. In the irregular code, accesses to x are irregular, dictated by the contents of the index array idx. It is unclear whether spatial locality exists or can be exploited by the cache. In the adaptive irregular code, the irregular accesses to x change as the program proceeds. While the program iterates the outer loop, the condition variable *change* will become true, then the index array idx will have different values, changing the access pattern to x in the inner loop. Changes in access patterns make locality optimizations more difficult.

This research was supported in part by NSF CAREER Development Award #ASC9625531 in New Technologies, NSF CISE Institutional Infrastructure Award #CDA9401151, and NSF cooperative agreement ACI-9619020 with NPACI and NCSA.

S.P. Midkiff et al. (Eds.): LCPC 2000, LNCS 2017, pp. 173–188, 2001.

```
// Irregular                    // Adaptive irregular

do t = 1, time                  do t = 1, time
  do i = 1, M                     if (change) idx[] = ...
    ... = x[idx[i]]               do i = 1, M
                                    ... = x[idx[i]]
```

Fig. 1. Regular and Irregular Applications

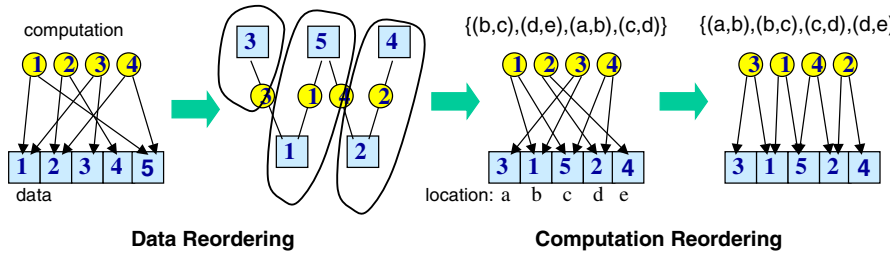


Fig. 2. Graph Partitioning & Lexicographic Sort

Researchers have demonstrated that the performance of irregular programs can be improved by applying a combination of computation and data layout transformations on irregular computations [3, 1, 4, 11, 12]. Results have been promising, but a number of issues have not been examined. Our paper makes the following contributions:

- Investigate the impact of locality optimizations on parallel programs. We find optimizations which yield better locality have greater impact on parallel performance than sequential codes, especially when parallelizing irregular reductions using local writes.
- Devise a heuristic for applying locality optimizations to adaptive irregular codes. We find a simple cost model may be used to accurately predict a desirable transformation frequency. An on-the-fly algorithm can apply the cost model by measuring the per-iteration execution time before and after optimizations.

The remainder of the paper begins with a discussion of our optimization framework. We then experimentally evaluate the effect of locality optimizations on parallel performance. Next, we evaluate the effect of adaptivity on locality optimizations, and provide both static and dynamic methods to apply locality optimizations based on a simple cost model. Finally, we conclude with a discussion of related work.

```
// 1 access                  // 2 accesses
global x(N)                  global x(N)
global idx1(M)               global idx1(M),idx2(M)

do i = 1, M                  do i = 1,M
   ... = x(idx1(i))             ... = x(idx1(i))
                                ... = x(idx2(i))
```

Fig. 3. Access Pattern of Irregular Computations

2 Locality Optimizations

2.1 Data & Computation Transformations

Irregular applications frequently suffer from high cache miss rates due to poor memory access behavior. However, program transformations may exploit dormant locality in the codes. The main idea behind these locality optimizations is to change computation order and or data layout at runtime, so that irregular codes can access data with more temporal and spatial locality.

Figure 2 gives an example of how program transformations can improve locality. Circles represent computations (loop iterations), squares represent data (array elements), and arrows represent data accesses. Initially memory accesses are irregular, but computation and/or data may be reordered to improve temporal and spatial locality.

Fortunately, irregular scientific computations are typically composed of loops performing *reductions* such as SUM and MAX, so loop iterations can be safely reordered [3, 4]. Data layouts can be safely modified as well, as long as all references to the data element are updated. Such updates are straightforward unless pointers are used.

2.2 Framework and Application Structure

A key decision is when computation and data reordering should be applied. For locality optimizations, we believe the number of distinct irregular accesses made by each loop iteration can be used to choose the proper optimization algorithm. Figure 3 shows examples of different access patterns. Codes may access either a single or multiple distinct elements of the array `x` on each loop iteration.

In the simplest case, each iteration makes only one irregular access to each array, as in the first code of Figure 3. The NAS integer sort (IS) and sparse matrix vector multiplication found in conjugate gradient (CG) fall into this category. Locality can be optimally improved by sorting computations (iterations) in memory order of array `x` [3, 4, 12]. Sorting computations may also be virtually achieved by sorting index array `idx1`.

More often, scientific codes access two or more distinct irregular references on each loop iteration. Such codes arise in PDE solvers traversing irregular meshes or N-body simulations, when calculations are made for pair of points. An example is shown in the second code of Figure 3. Notice that since each iteration accesses two array elements, computations can be viewed as edges connecting

data nodes, resulting in a graph as in Figure 2. Locality optimizations can then be mapped to a graph partitioning problem. Partitioning the graph and putting nodes in a partition close in memory can then improve spatial and temporal locality. Applying lexicographic sorting after partitioning captures even more locality. Finding optimal graph partitions is NP-hard, thus several optimization heuristics have been proposed [3, 1, 4, 8]. In our evaluation, we use two simple traversing algorithms called *Reverse Cuthill-Mckee* (RCM) and *Consecutive Packing* (CPACK). We also use three graph partitioning algorithms called *Recursive Coordinate Bisection* (RCB), *multi-level graph partitioning* (METIS), and *low overhead graph partitioning* (GPART).

3 Evaluation of Locality Optimization

3.1 Experimental Evaluation Environment

Our experimental results are obtained on a Sun HPC10000 which has 400 MHz UltraSparc II processors with 16K direct-mapped L1 and 4M direct-mapped L2 caches. Our prototype compiler is based on the Stanford SUIF compiler [15]. It can identify and parallelize irregular reductions using *pthreads*, but does not yet generate inspectors like Chaos [9]. As a result, we currently insert inspectors by hand for both the sequential and parallel versions of each program.

We examine three irregular applications, IRREG, NBF, and MOLDYN. Each application contains an initialization section followed by the main computation enclosed in a sequential time step loop. Statistics and timings are collected after the initialization section and the first iteration of the time step loop, in order to more closely match steady-state execution.

IRREG is a representative of iterative partial differential equation (PDE) solvers found in computational fluid dynamics (CFD) applications. In such codes, unstructured meshes are used to model physical structures. NBF is a kernel abstracted from the GROMOS molecular dynamics code. NBF computes a force between two molecules and applied to velocities of both molecules. MOLDYN is abstracted from the non-bonded force calculation in CHARMM, a key molecular dynamics application used at NIH to model macromolecular systems.

To test the effects of locality optimizations, we chose a variety of input data meshes. FOIL and AUTO are 3D meshes of a parafoil and GM Saturn automobile, respectively. The ratios of edges to nodes are between 7–10 for these meshes. MOL1 and MOL2 are small and large 3D meshes derived from semi-uniformly placed molecules of MOLDYN using a 1.5 angstrom cutoff radius. We applied these meshes to IRREG, NBF, and MOLDYN to test the locality effects. All meshes are initially sorted, so computation reordering is not required originally, but computation reordering is applied only after data reordering techniques are used, since data reordering makes computations out of order. FOIL and MOL1 roughly have 140K nodes and AUTO and MOL2 have roughly 440K nodes. Their *edges/nodes* ratios are 7–9.

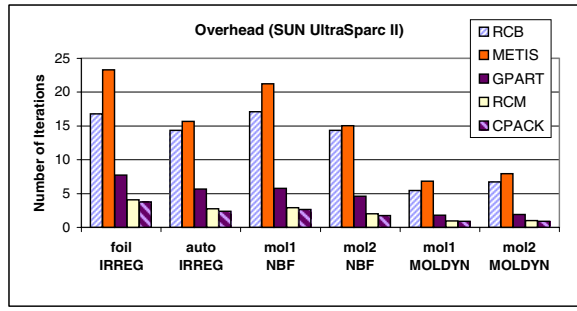


Fig. 4. Overhead of Data Reordering (relative to 1 iteration of ORIG)

3.2 Overhead of Optimizations

Each locality optimization has relative processing overhead. Figure 4 displays the costs of data reordering techniques measured relative to the execution time of one iteration of the time step loop.

The overhead includes the cost to update edge structures and transform other related data structures to avoid the extra indirect accesses caused by the data reordering. The overhead also includes the cost of computation reordering.

The least expensive data layout optimization is CPACK. RCM has almost same overhead as CPACK. In comparison, partitioning algorithms are quite expensive when used for cache optimizations. RCB and METIS are quite expensive when used for cache optimizations, on the order of 5–45 times higher than CPACK. The overhead of GPART is much less than METIS and RCB, but 3–5 times higher than CPACK.

3.3 Parallelizing Irregular Codes

Another issue we consider is the effect of locality optimizations on parallel execution. We find parallel performance is also improved, but the impact is dependent on the parallelization strategy employed by the application. We thus first briefly review two parallelization strategies.

The core of irregular scientific applications is frequently comprised of *reductions*, associative computations (e.g., SUM, MAX) which may be reordered and parallelized [14]. Compilers for shared-memory multiprocessors generally parallelize irregular reductions by having each processor compute a portion of the reduction, storing results in a local *replicated buffer*. Results from all replicated buffers are then combined with the original global data, using synchronization to ensure mutual exclusion [6, 14].

An example of the REPLICATEBUFS technique is shown in Figure 5. If large replicated buffers are to be combined, the compiler can avoid serialization by directing the run-time system to perform global accumulations in sections using a pipelined, round-robbin algorithm [6]. REPLICATEBUFS works well when the result of the reduction is to a scalar value, but is less efficient when the reduction

```
global x[nodes],y[nodes]
local  ybuf[nodes]
do t =               // time-step loop
  ybuf[] = 0         // init local buffer
  do i = {my_edges} // local computation
    n = idx1[i]
    m = idx2[i]
    force = f(x[m], x[n])
    ybuf[n] += force   // updates stored in
    ybuf[m] += -force  //   replicated ybuf
  reduce_sum(y, ybuf)  // combine buffers
```

Fig. 5. ReplicateBufs Example

```
global x[nodes],y[nodes]
inspect(idx1,idx2) // calc local_edges/cut_edges
do t =             // time-step loop
  do i = {local_edges} // both LHS's are local
    n = idx1[i]
    m = idx2[i]
    force = f(x[m], x[n])
    y[n] += force
    y[m] += -force
  do i = {cut_edges}   // one LHS is local
    n = idx1[i]
    m = idx2[i]
    force = f(x[m], x[n]) // replicated compute
    if(own(y[n]) y[n] += force
    if(own(y[m]) y[m] += -force
```

Fig. 6. LocalWrite Inspector Example

is to an array, since the entire array is replicated and few of its elements are effectively used.

An alternative method is LocalWrite, a compiler and run-time technique for parallelizing irregular reductions we previously developed [7]. LocalWrite avoids the overhead of replicated buffers and mutual exclusion during global accumulation by partitioning computation so that each processor only computes new values for locally-owned data. It simply applies to irregular computations the *owner-computes* rule used in distributed-memory compilers [10]. LocalWrite is implemented by having the compiler insert inspectors to ensure each processor only executes loop iterations which write to the local portion of each variable. Values of index arrays are examined at run time to build a list of loop iterations which modifies local data.

An example of LocalWrite is shown in Figure 6. Computation may be replicated whenever a loop iteration assigns the result of a computation to data belonging to multiple processors (*cut edge*). The overhead for LocalWrite should be much less than classic inspector/executors, because the LocalWrite inspector does not build communication schedules or perform address translation. Besides, LocalWrite does not perform global accumulation for the non-local data. Instead, LocalWrite replicates computation, avoiding expensive communications across processors.

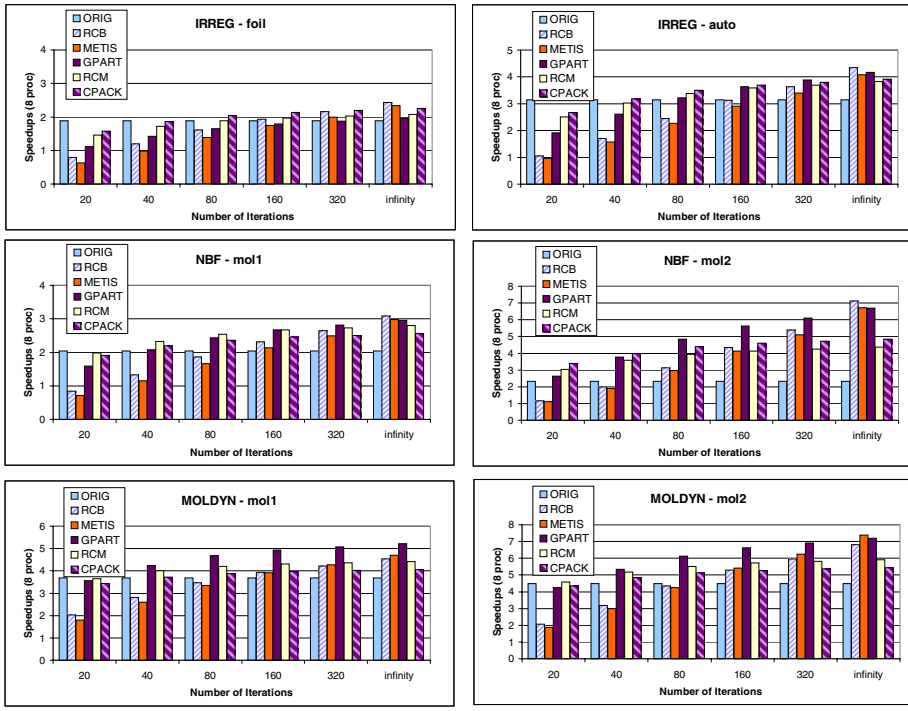


Fig. 7. Speedups on HPC 10000 (parallelized with ReplicateBufs)

The LocalWrite algorithm inspired our techniques for improving cache locality for irregular computations. Conventional compiler analysis cannot analyze, much less improves locality of irregular codes because the memory access patterns are unknown at compile time. The lightweight inspector in LocalWrite, however, can reorder the computations at run time to enforce local writes. It is only a small modification to change the inspector to reorder the computations for cache locality as well as local writes. We can use all of the existing compiler analysis for identifying irregular accesses and reductions (to ensure reordering is legal).

Though targeting parallel programs, our locality optimizations preprocess data sequentially. If overhead is too high, parallel graph partitioning algorithms may be used to reduce overhead for parallel programs.

3.4 Impact on Parallel Performance

Figure 7 and Figure 8 display 8-processor Sun HPC10000 speedups for each mesh, calculated versus the original, unoptimized program. Figure 7 shows speedups when applications are parallelized using ReplicateBufs and Figure 8 shows speedups when LocalWrite is used. We include overheads to show how performance changes as the total number of iterations executed by the program

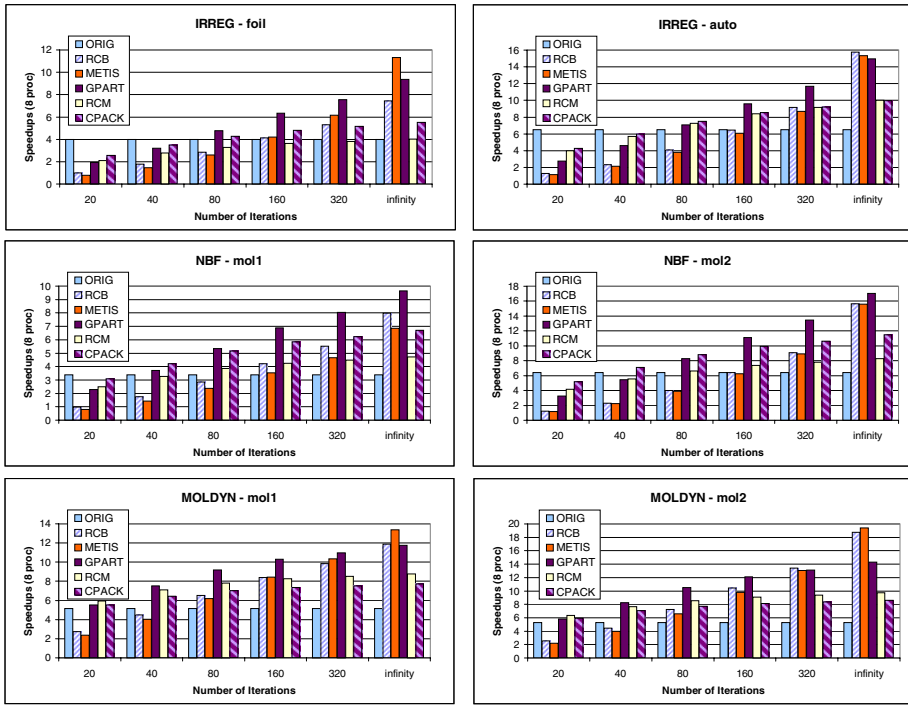


Fig. 8. Speedups on HPC 10000 (parallelized with LocalWrite)

increases. We see that when a sufficient number of iterations is executed, the versions optimized for locality achieved better performance. Locality optimizations thus also improve performance of parallel versions of each program.

In addition, we found that with locality optimizations, programs parallelized using LocalWrite achieved much better speedups than the original programs using ReplicateBufs. The LocalWrite algorithm tends to be more effective with larger graphs where duplicated computations are relatively few compared to computations performed locally. In general, the LocalWrite algorithm benefited more from the enhanced locality. Intuitively these results make sense, since the LocalWrite optimization can avoid replicated computation and communication better when the mesh displays greater locality.

Results show higher quality partitions become more important for parallel codes. Optimizations such as CPACK and RCM which yield almost the same improvements as RCB and METIS for uniprocessors [8] perform substantially worse on parallel codes. In comparison, GPART achieves roughly the same performance as the more expensive partitioning algorithms, even for parallel codes.

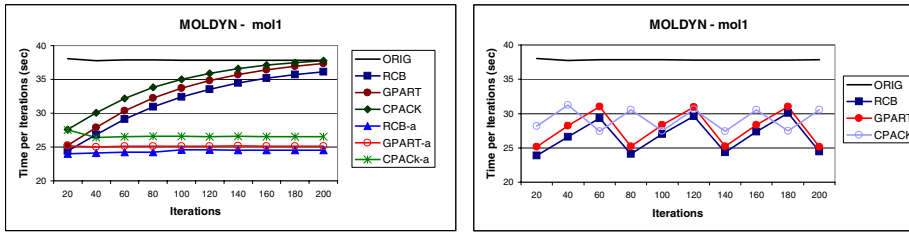


Fig. 9. Performance Changes in Adaptive Computations (overhead excluded)

4 Optimizing Adaptive Computations

A problem confronting locality optimizations for irregular codes is that many such applications are *adaptive*, where the data access pattern may change over time as the computation adapts to data. For instance, the example in Figure 1 is adaptive, since condition `change` may be satisfied on some iterations of the time-step loop, modifying elements of the index arrays `idx`. Fortunately, changing data access patterns reduces locality and degrades performance, but does not affect legality. Locality optimizations are not reapplied after every change, but only when it is deemed profitable.

4.1 Impact of Adaptation

To evaluate the effect of adaptivity on performance after computation/data reordering, we timed MOLDYN with input MOL1, periodically swapping 10% of the molecules randomly every 20 iterations to create an adaptive code. We realize this is not realistic, but it provides a testbed for our experiments. Adaptive behavior in MOLDYN is actually dependent on the initial position and velocity of molecules in the input data. Future research will need to conduct a more comprehensive study of adaptive behavior in scientific programs.

Results for our synthetic adaptive code is shown in Figure 9. In the first graph, the x-axis marks the passage of time in the computation, while the y-axis measures execution time. ORIG is the original program. RCB, METIS, GPART, and CPACK represent versions of the program where data and computation are reordered for improved locality exactly once at the beginning of program. In comparison, RCB–a, METIS–a, GPART–a, and CPACK–a represent the versions where data and computation are reordered whenever access patterns change. Data points represent the execution times for every 20 iterations of the program. Results show that without reapplying locality reordering, performance degrades and eventually matches with the unoptimized program. In comparison, Reapplying partitioning algorithms after every access pattern change can preserve the performance benefits of locality, if overhead is excluded.

In practice, however, we do need to take into account overhead. By periodically applying reordering, we can maximize the net benefit of locality reordering. Performance changes with periodic reordering are shown in the second graph in

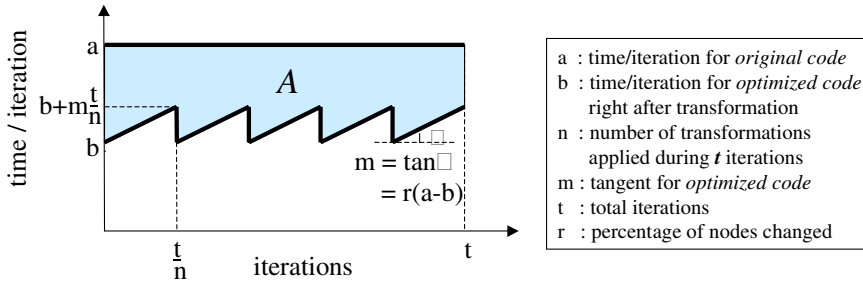


Fig. 10. Analytical Model for Adaptive Computations

$$G(n) = A - nO_v \qquad (n \geq 1) \tag{1}$$

$$where, \quad A = (a - (b + \frac{mt}{2n}))t \quad = \quad (a-b)t - \frac{mt^2}{2n}$$

$$O_v = \text{overhead of reordering}$$

$$n = \text{number of transformations applied}$$

$$n_0 = \left(\sqrt{\frac{m}{2O_v}}\right) t \qquad (G'(n_0) = 0) \tag{2}$$

$$G_{max} \quad = \quad \begin{cases} G(n_0) = (a-b)t - (\sqrt{2mO_v})t & (n_0 \geq 1) \\ G(1) \ = \ (a-b)t - \frac{mt^2}{2} - O_v & (n_0 < 1) \end{cases} \tag{3}$$

Fig. 11. Calculating Net Gain and Adaptation Frequency

Figure 9. We re-apply CPACK every 40 iterations, and RCB and GPART every 60 iterations. Results show performance begins to degrade, but recover after locality optimizations are reapplied. Even after overheads are included, the overall execution time is improved over not reapplying locality optimizations. The key question is what frequency of reapplying optimizations results in maximum net benefit. In the next section we attempt to calculate the frequency and benefit of reordering.

4.2 Cost Model for Optimizations

To guide locality transformations for adaptive codes, we need a cost model to predict the benefits of applying optimizations. We begin by showing how to calculate costs when all parameters such as optimization overhead, optimization benefit, access change frequency, and access change magnitude are known. Later we show how this method may be adopted to work in practice by collecting

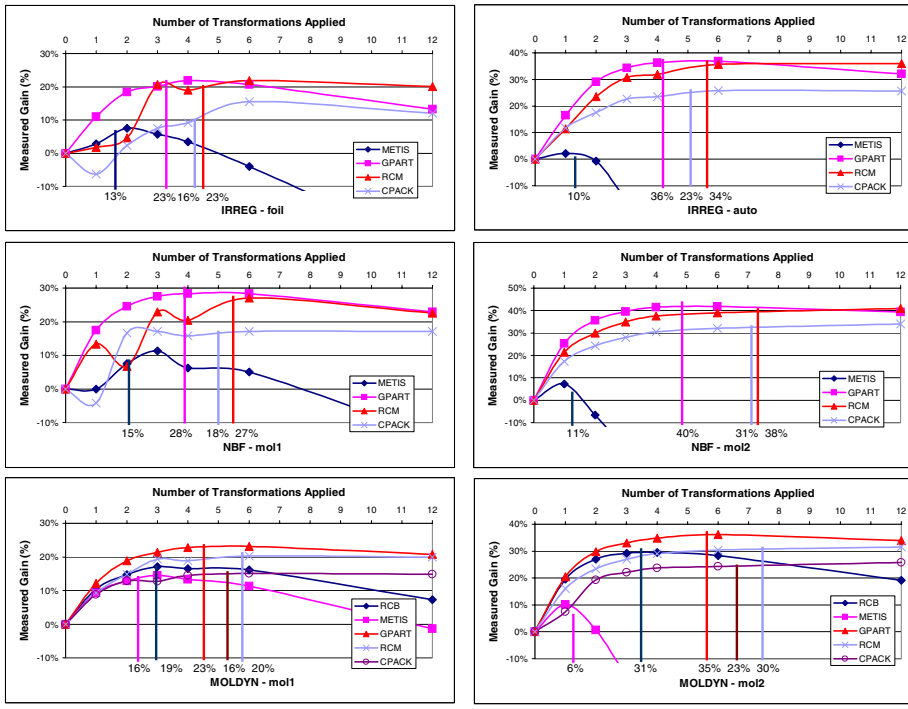


Fig. 12. Experimental Verification of Cost Model (vertical bars represent the numbers chosen by the cost model)

and fixing some parameters and gathering the remaining information on-the-fly through dynamic instrumentation.

First we present a simple cost model for computing the benefits of locality reordering when all information is available. We can use it to both predict improvements for different optimizations and decide how frequently locality transformations should be reapplied.

We consider models for the performance of two versions of a program, *original code* and *optimized code*. Figure 10 plots execution time per iteration, excluding the overhead. The upper straight line corresponds to the *original code* and the lower saw-tooth line corresponds to the *optimized code*. For the original code, we assume randomly initialized input data, which makes execution times per iteration stay constant after node changes. For the optimized code, locality optimization is performed at the beginning and periodically reapplied. For simplicity, we assume execution times per iteration increase linearly as the nodes change and periodically drop to the lowest (optimized) point after reordering is reapplied. Execution times per iteration do not include the overhead of the locality reordering, but we will take it into account later in our benefit calculation.

```
do t = 1, n           // time-step loop
  if (change)         // change accesses
    idx1[]=
    access_changed() // note access changes
  cost_model()        // apply cost model

  do i = 1, edges     // main computation
    ...
```

Fig. 13. Applying Cost Model Dynamically

Performance degradation rate (m) of the optimized code is set to $r(a - b)$, where r is the percentage of changed nodes. For example, if an adaptive code randomly changes 10% of nodes each iteration ($r = 0.1$), then the execution time per iteration becomes that of the original code after 10 iterations. The net performance gain ($G(n)$) from periodic locality transformations can be calculated as in Figure 11. Since the area below the line is the total execution time, the net performance gain will the area between two lines (A) minus the total overhead (nO_v). Taking the differential equation of $G(n)$, we can calculate the point where the net gain is maximized. In practice, data access patterns often change periodically instead of after every iteration, since scientists often accept less precision for better execution times. We can still directly apply our model in such cases, assuming one iteration in our model corresponds to a set of loop iterations where data access patterns change only on the last iteration.

To verify our cost model, we ran experiments with three kernels on a DEC Alpha 21064 processor. The kernels iterate 240 time steps, randomly swapping 10% of nodes every 20 iterations. We will use these adaptive programs for all the remaining experiments for adaptive codes. Results are shown in Figure 12. The y-axis represents the percentage of net performance gain over original execution time. The x-axis represents the number of transformations applied throughout the 240 time steps. Different curves correspond to different locality reordering, varying numbers of transformations applied. The vertical bars represent the numbers of transformations chosen by our cost model and the percentage numbers under the bars represent the predicted gain. The optimization frequency calculated by the cost model is not an integer and needs to be rounded to an integer in practice. Nonetheless, results show our cost model predicts precise performance gains, and always selects nearly the maximal point of each measured curve.

4.3 On-the-Fly Application of Cost Model

Though experiments show the cost model introduced in the previous section is very precise, it requires several parameters to calculate the frequency and net benefit of locality optimizations. In practice, some of the information needed is either expensive to gather (e.g., overhead for each optimization) or nearly impossible to predict ahead of time (e.g., rate of change in data access patterns).

In this section, we show how a simplified cost model can be applied by gathering information on-the-fly by inserting a few timing routines in the program. First, we apply only GPART as locality reordering. Thus, we only calculate the frequency of reordering, not the expected gain. Since GPART has low overhead

	IRREG		NBF		MOLDYN	
	FOIL	AUTO	MOL1	MOL2	MOL1	MOL2
Number of Transformations	4	4	4	4	4	6
Measured Gain (%)	20.7	36.0	27.1	41.5	22.1	33.9

Table 1. Performance Improvement for Adaptive Codes with On-the-fly Cost Model

but yields high quality ordering, it should be suitable for adaptive computations. Second, performance degradation rate (m) is gathered at run time (instead of calculating from the percentage of node changes) and adjusted by monitoring performance (execution time) of every iteration. The new rate is chosen so that actual elapsed time since the last reordering is the same as calculated elapsed time under the newly chosen linear rate.

Figure 13 shows an example code that uses on-the-fly cost model. At every iteration, `cost_model()` is called to calculate a frequency based on the information monitored so far. In our current setup, the cost model does not monitor the first iteration to avoid noisy information from cold start. On the second iteration, it applies GPART and monitors the overhead and the optimized performance (b). In the third iteration, it does not apply reordering and monitors the performance degradation. The cost model now has accumulated enough information to calculate the desired frequency of reapplying locality optimizations. From this point on, the cost model keeps monitoring performance of each iteration, adjusting degradation rate and produces new frequencies based on accumulated information. When the desired interval is reached, it re-applies GPART.

Since irregular scientific applications may change mesh connections periodically (e.g., every 20 iterations), an optimization for the on-the-fly cost model is detecting periodical access pattern changes. Rounding up the predicted optimization frequency so transformations are performed immediately after the next access pattern change can then fully exploit the benefit of locality reordering. For this purpose, the compiler can insert a call to *access_changed()* as shown in Figure 13. This function call notifies the cost model when access pattern changes. The cost model then decides whether access patterns periodically changes.

To evaluate the precision of our on-the-fly cost model, we performed experiments with the three kernels under the same situation. Table 1 shows the performance improvements over original programs that do not have locality reordering. The improvements closely match with the maximum improvements in Figure 12 and the numbers of transformations also match with the numbers in periodic reordering. Our on-the-fly cost model thus works well in practice with limited information.

5 Related Work

Researchers have investigated improving locality for irregular scientific applications. Das *et al.* investigated data/computation reordering for unstructured euler solvers [3]. They combined data reordering using RCM and lexicographical sort for computation reordering. to improve performance of parallel codes on an

Intel iPSC/860. Al-Furaih and Ranka studied partitioning data using METIS and BFS to reorder data in irregular codes [1]. They conclude METIS yields better locality, but they did not include computation reordering.

Ding and Kennedy explored applying dynamic copying (packing) of data elements based on loop traversal order, and show major improvements in performance [4]. They were able to automate most of their transformations in a compiler, using user provided information. For adaptive codes they re-apply transformations after every change. In comparison, we show partitioning algorithms can yield better locality, albeit with higher processing overhead. We also develop a more sophisticated on-the-fly algorithm for deciding when to re-apply transformations for adaptive codes. Ding and Kennedy also developed algorithms for reorganizing single arrays into multi-dimensional arrays depending on their access patterns [5]. Their technique might be useful for Fortran codes where data are often single dimension arrays, not structures as in C codes.

Mellor-Crummey *et al.* used a geometric partitioning algorithm based on space-filling curves to map multidimensional data to memory [11]. In comparison, our graph-based partitioning techniques are more suited for compilers, since geometric coordinate information for space-filling curves do not need to provided manually. When coordinate information is available, using RCB is better because space-filling curves cannot guarantee evenly balanced partition when data is unevenly distributed, which may cause significant performance degradation in parallel execution.

Mitchell *et al.* improved locality using bucket sorting to reorder loop iterations in irregular computations [12]. They improved the performance of two NAS applications (CG, and IS) and a medical heart simulation. Bucket sorting works only for computations containing a single irregular access per loop iteration. In comparison, we investigate more complex cases where two or more irregular access patterns exist. For simple codes lexicographic sort yields improvements similar to bucket sorting.

Researchers have previously proposed on-the-fly algorithms for improving load balance in parallel systems, but we are not aware of any such algorithm for guiding locality optimizations. Bull uses a dynamic system to improve load balance in loop scheduling [2]. Based on the previous execution time of each parallel loop iteration roughly equal amounts of computation are assigned.

Nicol and Saltz investigated algorithms to remap data and computation for adaptive parallel codes to reduce load imbalance [13]. They use a dynamic heuristic which monitors load imbalance on each iteration, predicting time lost to load imbalance. Data an computation is remapped using a greedy *stop-on-rise* heuristic, when a local minima is reached in predicted benefit. We adapt a similar approach for our on-the-fly optimization technique, but use it to guide transformations to improve locality instead of load imbalance.

6 Conclusions

In this paper, we propose a framework that guides how to apply locality optimizations according to the application access pattern. We find locality optimizations also improve performance for parallel codes, especially when combined

with parallelization techniques which benefit from locality. We show locality optimizations may be used to improve performance even for adaptive codes, using an on-the-fly cost model to select for each optimization how often to reorder data.

As processors speed up relative to memory systems, locality optimizations for irregular scientific computations should increase in importance, since processing costs go down while memory costs increase. For very large graphs, we should also obtain benefits by reducing TLB misses and paging in the virtual memory system. By improving compiler support for irregular codes, we are contributing to our long-term goal: making it easier for scientists and engineers to take advantage of the benefits of high-performance computing.

References

[1] I. Al-Furaih and S. Ranka. Memory hierarchy management for iterative graph structures. In *Proceedings of the 12th International Parallel Processing Symposium*, Orlando, FL, April 1998.

[2] J. M. Bull. Feedback guided dynamic loop scheduling: Algorithms and experiments. In *Proceedings of the Fourth International Euro-Par Conference (Euro-Par'98)*, Southhampton, UK, September 1998.

[3] R. Das, D. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy. The design and implementation of a parallel unstructured euler solver using software primitives. In *Proceedings of the 30th Aerospace Sciences Meeting & Exhibit*, Reno, NV, January 1992.

[4] C. Ding and K. Kennedy. Improving cache performance of dynamic applications with computation and data layout transformations. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.

[5] C. Ding and K. Kennedy. Inter-array data regrouping. In *Proceedings of the Twelfth Workshop on Languages and Compilers for Parallel Computing*, San Diego, CA, August 1999.

[6] M. Hall, S. Amarasinghe, B. Murphy, S. Liao, and M. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995.

[7] H. Han and C.-W. Tseng. Improving compiler and run-time support for adaptive irregular codes. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Paris, France, October 1998.

[8] H. Han and C.-W. Tseng. A comparison of locality transformations for irregular codes. In *Proceedings of the 5th Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers*, Rochester, NY, May 2000.

[9] R. v. Hanxleden and K. Kennedy. Give-N-Take — A balanced code placement framework. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, Orlando, FL, June 1994.

[10] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[11] J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving memory hierarchy performance for irregular applications. In *Proceedings of the 1999 ACM International Conference on Supercomputing*, Rhodes, Greece, June 1999.

[12] N. Mitchell, L. Carter, and J. Ferrante. Localizing non-affine array references. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Newport Beach , LA, October 1999.

[13] D. Nicol and J. Saltz. Dynamic remapping of parallel computations with varying resource demands. *IEEE Transactions on Computers*, 37(9):1073–1087, September 1988.

[14] W. Pottenger. The role of associativity and commutativity in the detection and transformation of loop level parallelism. In *Proceedings of the 1998 ACM International Conference on Supercomputing*, Melbourne, Australia, July 1998.

[15] R. Wilson et al. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.

Automatic Coarse Grain Task Parallel Processing on SMP Using OpenMP

Hironori Kasahara, Motoki Obata, and Kazuhisa Ishizaka

Waseda University
3-4-1 Ohkubo, Shinjuku-ku, Tokyo, 169-8555, Japan
{kasahara,obata,ishizaka}@oscar.elec.waseda.ac.jp

Abstract. This paper proposes a simple and efficient implementation method for a hierarchical coarse grain task parallel processing scheme on a SMP machine. OSCAR multigrain parallelizing compiler automatically generates parallelized code including OpenMP directives and its performance is evaluated on a commercial SMP machine. The coarse grain task parallel processing is important to improve the effective performance of wide range of multiprocessor systems from a single chip multiprocessor to a high performance computer beyond the limit of the loop parallelism. The proposed scheme decomposes a Fortran program into coarse grain tasks, analyzes parallelism among tasks by "Earliest Executable Condition Analysis" considering control and data dependencies, statically schedules the coarse grain tasks to threads or generates dynamic task scheduling codes to assign the tasks to threads and generates OpenMP Fortran source code for a SMP machine. The thread parallel code using OpenMP generated by OSCAR compiler forks threads only once at the beginning of the program and joins only once at the end even though the program is processed in parallel based on hierarchical coarse grain task parallel processing concept. The performance of the scheme is evaluated on 8-processor SMP machine, IBM RS6000 SP 604e High Node, using a newly developed OpenMP backend of OSCAR multigrain compiler. The evaluation shows that OSCAR compiler with IBM XL Fortran compiler version 5.1 gives us 1.5 to 3 times larger speedup than the native XL Fortran compiler for SPEC 95fp SWIM, TOMCATV, HYDRO2D, MGRID and Perfect Benchmarks ARC2D.

1 Introduction

The loop parallelization techniques, such as Do-all and Do-across, have been widely used in Fortran parallelizing compilers for multiprocessor systems[1, 2]. Currently, many types of Do-loop can be parallelized with various data dependency analysis techniques[3, 4] such as GCD, Benerjee's inexact and exact tests[1, 2], OMEGA test[5], symbolic analysis[6], semantic analysis and dynamic dependence test and program restructuring techniques such as array privatization[7], loop distribution, loop fusion, strip mining and loop interchange [8, 9].

S.P. Midkiff et al. (Eds.): LCPC 2000, LNCS 2017, pp. 189–207, 2001.

For example, Polaris compiler[10, 11, 12] exploits loop parallelism by using inline expansion of subroutine, symbolic propagation, array privatization[7, 11] and run-time data dependence analysis[12]. SUIF compiler parallelizes loops by using inter-procedure analysis[13, 14, 15], unimodular transformation and data locality optimization[16, 17]. Effective optimization of data localization is getting more important because of the increasing disparity between memory and processor speeds. Currently, many researches for data locality optimization using program restructuring techniques such as blocking, tiling, padding and data localization, are proceeding for high performance computing and single chip multiprocessor systems [16, 18, 19, 20].

However, these compilers cannot parallelize loops that include complex loop carrying dependences and conditional branches to the outside of a loop.

Considering these facts, the coarse grain task parallelism should be exploited to improve the effective performance of multiprocessor systems further in addition to the improvement of data dependence analysis, speculative execution and so on.

PROMIS compiler[21, 22] hierarchically combines Parafrase2 compiler[23] using HTG[24] and symbolic analysis techniques[6] and EVE compiler for fine grain parallel processing. NANOS compiler[25, 26] based on Parafrase2 has been trying to exploit multi-level parallelism including the coarse grain parallelism by using extended OpenMP API[27, 28]. OSCAR compiler has realized a multi-grain parallel processing [29, 30, 31] that effectively combines the coarse grain task parallel processing [29, 30, 31, 32, 33, 34], which can be applied for a single chip multiprocessor to HPC multiprocessor systems, the loop parallelization and near fine grain parallel processing[35]. In OSCAR compiler, coarse grain tasks are dynamically scheduled onto processors or processor clusters to cope with the runtime uncertainties caused by conditional branches by dynamic scheduling routine generated by the compiler. As the embedded dynamic task scheduler, the centralized dynamic scheduler[30, 31] in OSCAR Fortran compiler and the distributed dynamic scheduler[36] have been proposed.

A coarse grain task assigned to a processor cluster is processed in parallel by processors inside the processor cluster with the use of the loop, the coarse grain and near fine grain parallel processing hierarchically.

This paper describes the implementation scheme of a coarse grain task parallel processing on a commercially available SMP machine and its performance. Ordinary sequential Fortran programs are parallelized using by OSCAR compiler automatically and a parallelized program with OpenMP API is generated. In other words, OSCAR Fortran Compiler is used as a preprocessor which transforms a Fortran program into a parallelized OpenMP Fortran realizing static scheduling and centralized and distributed dynamic scheduling for coarse grain tasks depending on parallelism of the source program and performance parameters of the target machines. Parallel threads are forked only once at the beginning of the program and joined only once at the end to minimize fork/join overhead. Though OpenMP API is chosen as the thread creation method because of the

portability, the proposed implementation scheme can be used for other thread generation method as well.

The performance of the proposed coarse grain task parallel processing in OSCAR multigrain compiler is evaluated on IBM RS6000 SP 604e High Node 8 processors SMP machine.

In the evaluation, OSCAR multigrain compiler automatically generates coarse grain parallel processing codes using a subset of OpenMP directives supported by IBM XL Fortran version 5.1. The codes are compiled by XL Fortran and executed on 8 processors of RS6000 SP 604e High Node.

The rest of this paper is composed as follows. Section 2 introduces the coarse grain task parallel processing scheme. Section 3 shows the implementation method of the coarse grain task parallelization on a SMP. Section 4 evaluates the performance of this method on IBM RS6000 SP 604e High Node for several programs like Perfect Benchmarks and SPEC 95fp Benchmarks.

2 Coarse Grain Task Parallel Processing

Coarse grain task parallel processing uses parallelism among three kinds of macro-tasks, namely, Basic Block(BB), Repetition Block(RB or loop) and Subroutine Block(SB). Macro-tasks are generated by decomposition of a source program and assigned to processor clusters or processor elements and executed in parallel inter and/or intra processor clusters.

The coarse grain task parallel processing scheme in OSCAR multigrain automatic parallelizing compiler consists of the following steps.

1. Generation of macro-tasks from a source code of an ordinary sequential program
2. Generation of Macro-Flow Graph which represents a result of data dependency and control flow analysis among macro-tasks.
3. Generation of Macro-Task Graph by analysis of parallelism among macro-tasks using Earliest Executable Condition analysis [29, 32, 33].
4. If a macro-task graph has only data dependency edges, macro-tasks are assigned to processor clusters or processor elements by static scheduling. If a macro-task graph has both data dependency and control dependency edges, macro-tasks are assigned to processor clusters or processor elements at runtime by dynamic scheduling routine generated and embedded into the parallelized user code by the compiler.

In the following, these steps are briefly explained.

2.1 Generation of Macro-tasks

In the coarse grain task parallel processing, a source program is decomposed into three kinds of macro-tasks, namely, Basic Block(BB), Repetition Block(RB) and Subroutine Block(SB) as mentioned above.

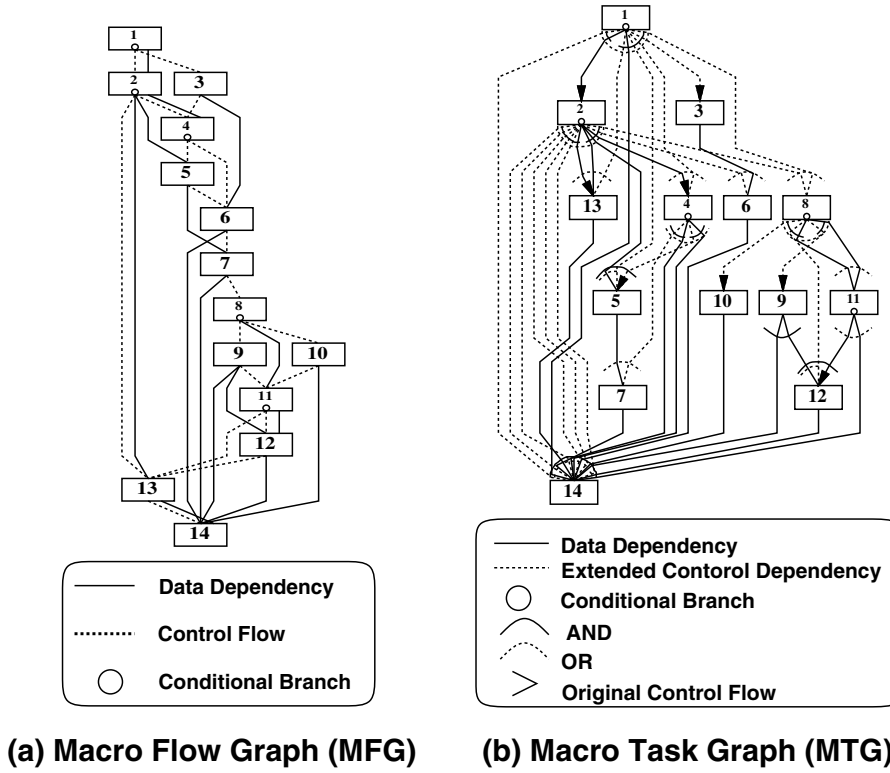


Fig. 1. Macro flow graph and macro-task graph

If there is a prallelizable loop, it is decomposed into smaller loops in the iteration direction and the decomposed partial loops are defined as different macro-tasks. The number of decomposed loops is decided considering the number of processor clusters or processor elements and cache or memory size.

RBs composed of a sequential loops having large processing cost and SBs, to which inline expansion can not be applied effectively, are decomposed into sub macro-tasks and the hierarchical coarse grain task parallel processing is applied as shown in Fig.2 explained later.

2.2 Generation of Macro-flow Graph

Next, the date dependency and control flow among macro-tasks for each nest level are analyzed hierarchically. The control flow and data dependency among macro-tasks are represented by macro-flow graph as shown in Fig.1(a).

In the figure, nodes represent macro-tasks, solid edges represent data dependencies among macro-tasks and dotted edges represent control flow. A small circle inside a node represents a conditional branch inside the macro-task. Though

arrows of edges are omitted in the macro-flow graph, it is assumed that the directions are downward.

2.3 Generation of Macro-task Graph

Though the generated macro-flow graph represents data dependencies and control flow, it does not represent parallelism among macro-tasks. To extract parallelism among macro-tasks from macro-flow graph, Earliest Executable Condition analysis considering data dependencies and control dependencies is applied. Earliest Executable Condition represents the conditions on which macro-task may begin its execution earliest. It is obtained assuming the following conditions.

1. If Macro-Task(MT)i data-depends on MTj, MTi cannot begin execution before MTj finishes execution.
2. If the branch direction of MTj is determined, MTi that control-depends on MTj can begin execution even though MTj has not completed its execution.

Then, the original form of Earliest Executable Condition is represented as follows;

(Macro-Task(MT)j, on which MTi is control dependent,
takes a branch that guarantees MTi will execute)
AND
(MTk($0 \leq k \leq |N|$), on which MTi is data dependent, completes execution
OR it is determined that MTk is not be executed),
where N is the number of predecessors of MTi

For example, the original form of Earliest Executable Condition of MT6 on Fig.1(b) is

(MT1 takes a branch that guarantees MT3 will be execute
OR MT2 takes a branch that guarantees MT4 will be execute)
AND
(MT3 completes execution
OR MT1 takes a branch that guarantees MT4 will be execute).

However, the completion of MT3 means MT1 already took the branch to MT3. Also, "MT2 takes a branch that guarantees MT4 will execute" means that MT1 already branched to MT2. Therefore, this condition is redundant and its simplest form is

(MT3 completes execution
OR MT2 takes a branch that guarantees MT4 will execute).

Earliest Executable Condition of macro-task is represented in a macro-task graph as shown in Fig.1(b).

In the macro-task graph, nodes represent macro-tasks. A small circle inside nodes represents conditional branches. Solid edges represent data dependencies.

Dotted edges represent extended control dependencies. Extended control dependency means ordinary normal control dependency and the condition on which a data dependence predecessor of MTi is not executed.

Solid and dotted arcs connecting solid and dotted edges have two different meanings. A solid arc represents that edges connected by the arc are in AND relationship. A dotted arc represents that edges connected by the arc are in OR relation ship.

In MTG, though arrows of edges are omitted assuming downward, an edge having arrow represents original control flow edges, or branch direction in macro-flow graph.

2.4 Generation of Scheduling Routine

In the coarse grain task parallel processing, the dynamic scheduling and the static scheduling are used for assignment of macro-tasks to processor clusters or processor elements. In the dynamic scheduling, MTs are assigned to processor clusters or processor elements at runtime to cope with runtime uncertainties like conditional branches. The dynamic scheduling routine is generated and embedded into user program by compiler to eliminate the overhead of OS call for thread scheduling.

Though generally dynamic scheduling overhead is large, in OSCAR compiler the dynamic scheduling overhead is relatively small since it is used for the coarse grain tasks assignment. There are two kinds of schemes for dynamic scheduling, namely, a centralized dynamic scheduling, in which the scheduling routine is executed by a processor element, and a distributed scheduling, in which the scheduling routine is distributed to all processors.

Also, in static scheduling, assignment of macro-tasks to processor clusters or processor elements is determined at compile-time inside auto parallelize compiler if macro-task graph has only data dependency edges. Static scheduling is useful since it allows us to minimize data transfer and synchronization overheard without run-time scheduling overhead.

3 Implementation of Coarse Grain Task Parallel Processing Using OpenMP

This section describes an implementation method of the coarse grain task parallel processing using OpenMP for SMP machines.

Though macro-tasks are assigned to processor clusters or processor elements in the coarse grain task parallel processing in OSCAR compiler, OpenMP only supports the thread level parallel processing. Therefore, the coarse grain parallel processing is realized by corresponding a thread to a processor element, and a thread group to a processor cluster.

Though OpenMP is used as a method of the thread generation in this implementation because of its high portability, the proposed scheme can be used with other thread creation methods as well.

3.1 Generation of Threads

In the proposed coarse grain task parallel processing using OpenMP, threads are generated by PARALLEL SECTIONS directive only once at the beginning of the execution of program.

Generally, upper level threads fork nested threads to realize nested or hierarchical parallel processing.

However, the proposed scheme realizes this hierarchical parallel processing with single level thread generation by writing all hierarchical behavior, or by embedding hierarchical scheduling routines, in each section between PARALLEL SECTIONS and END PARALLEL SECTIONS. This scheme allows us to minimize thread fork and join overhead and to implement hierarchical coarse grain parallel processing without any language extension.

3.2 Macro-task Scheduling

This section describes code generation scheme using static and dynamic scheduling to assign macro-tasks to threads or thread groups hierarchically.

In the coarse grain task parallel processing by OSCAR compiler, the macro-tasks are assigned to threads or thread groups at run-time and/or at compilation-time. OSCAR compiler can choose the centralized dynamic scheduling and/or the distributed dynamic scheduling scheme in addtion to static scheduling. These scheduling methods are suitably used considering parallelism of the source program, a number of processors, data transfer and syncronization overhead of a target multiprocessor system with their any combinations. In the centralized dynamic scheduling, scheduling code is assigned to a single thread. In the distributed dynamic scheduling, scheduling code is distributed to before and after each task assuming exclusive access to the scheduling tables. More concretely, the compiler chooses static scheduling for a macro-task graph with data dependence edges and dynamic scheduling for a macro-task graph with control dependence edges in each layer, or nest level. Also, centralized scheduler is usually chosen for a processor, or a thread group, and distributed scheduler is chosen for a processor cluster with low mutual exclusion overhead to shared scheduling information.

Those scheduling methods can be hierarchically combined freely depending on program parallelism, the number of processors available for the program layer, syncronization overhead and so on.

Centralized Dynamic Scheduling. In centralized scheduling scheme, one thread in a parallel processing layer choosing centralized scheduling serves as centralized scheduler, which assigns macro-tasks to thread groups, namely, processor clusters or processor elements. This thread is called scheduler or Centralized Scheduler.

The behavior of *CS* written in OpenMP "SECTION" is shown in the following.

step1 Receive a completion or branch signal from each macro-task.
step2 Check Earliest Executable Condition, and enqueue ready macro-tasks, which satisfy this condition, to a ready task queue.

step3 Find a processor cluster, or a thread group, to which a ready macro-task should be assigned according to the priority of Dynamic CP method.[29]
step4 Assign macro-task to the processor cluster or the processor element. If the assigned macro-task is "End MT"(EMT), the centralized scheduler finishes scheduling routine in the layer.
step5 Jump to step1.

In the centralized scheduling scheme, ready macro-tasks are initially assigned to thread groups. Each thread group executes these macro-tasks at the beginning. When macro-task finishes execution or determines branch direction, it sends signal to the centralized scheduler. Therefore the centralized scheduler busy waits for these signals at the start.

If the centralized scheduler receives these signals, it searches new executable, or ready, macro-tasks by checking Earliest Executable Condition for each macro-task.

If a ready macro-task is found, the centralized scheduler finds a thread group, or a thread, to which a macro-task should be assigned. The centralized scheduler assigns macro-task and goes back to the signal waiting routine.

On the other hand, slave threads execute macro-tasks assigned by the centralized scheduler. Their behavior is summarized in following.

step1 Wait macro-task assignment information from the centralized scheduler
step2 Execute the assigned macro-task
step3 Go back to step1

So, at the beginning of OpenMP "SECTION" or a thread, slave thread executes the busy/wait code for waiting assignment information from the centralized scheduler if a macro-task isn't assigned initially.

In the dynamic scheduling mode, since every thread or thread group has possibility to execute all macro-tasks, the whole code including all macro-tasks is copied into each OpenMP "SECTION" for each slave thread.

Figure 2 shows an image of generated OpenMP code for every thread. Sub macro-tasks generated inside of Macro-Task(MT)3, a macro-task in the 1st layer, are represented as macro-tasks in the 2nd layer, MT3_1, MT3_2, and so on. The 2nd layer macro-tasks are executed on "k" threads for program execution and one thread to serve as a centralized dynamic scheduler. Moreover, MTs like MT3_2_1 and MT3_2_2 in the 3rd layer are generated inside MT3 in the 2nd layer. Figure 2 exemplifies a code image in a case where the 3rd layer MTs like MT3_2_1 and MT3_2_2 are dynamically scheduled to threads by the distributed scheduler. The details of distributed scheduling are described later.

After the completion of the execution of a macro-task, the slave threads go back to the routine to wait for the assignment of a macro-task by a centralized scheduler.

Also, the compiler generates a special macro-task called "End MT"(EMT) in each layer. As shown in the 2nd layer in Fig.2, the EndMT is written at the end of all OpenMP "SECTION", and *CS* assigns EMT to the all thread groups when

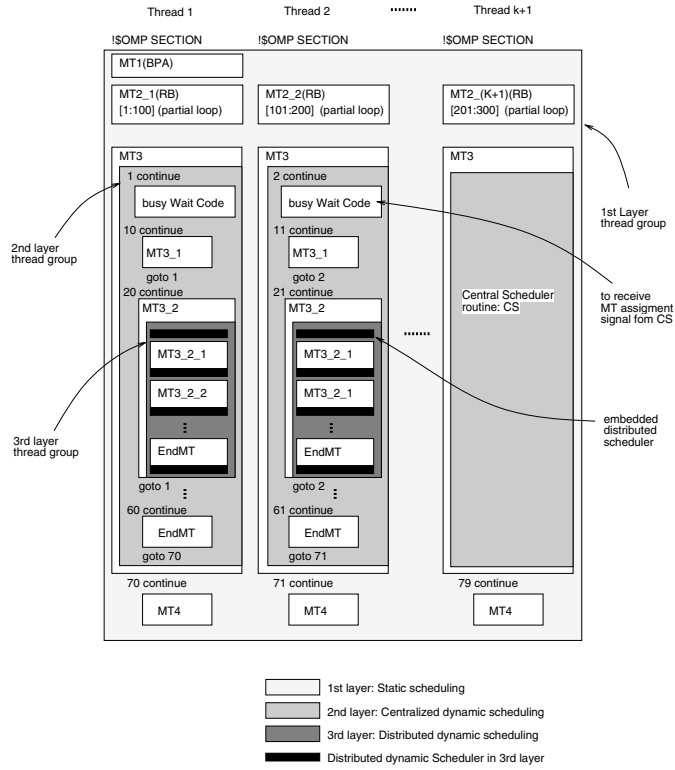


Fig. 2. Code image ($k$threads)

all of the threads executing the same hierarchy are finished. After assigning EMT to the all threads, *CS* finishes scheduling routine in the layer. Each thread group jumps to outside of its hierarchy. If the hierarchy is a top layer, the program finishes the execution. If there exists an upper layer, threads continue to execute the upper layer macro-tasks.

In the example shown in Fig.2, thread1 finishes the execution in 2nd layer by "goto 70" after "End MT" and jumps out to a statement"70 continue" in the upper layer.

Distributed Dynamic Scheduling. Each thread group or processor cluster schedules a macro-task to itself and executes the macro-task in the layer where the distributed dynamic scheduler is chosen.

In the distributed scheduling scheme, all shared data for scheduling are assigned onto shared memory and accessed exclusively.

step1 Search executable, or ready, macro-tasks that satisfy Earliest Executable Condition by the completion or a branch of the macro-task and enqueue

the ready macro-tasks to the ready queue with exclusive access to data on shared memory required for dynamic scheduling.

step2 Choose a macro-task, which the thread should execute next, considering Dynamic CP algorithm's priority.

step3 Execute the macro-task

step4 Update the Earliest Executable Condition table exclusively.

step5 Go back to step1

For example, the 3rd layer shown in Fig.2 uses the distributed dynamic scheduling scheme. In this example, the distributed scheduling is applied to inside of MT3_2 in second layer, and two thread groups that consist of one thread are realized in this layer by only executing the thread code generated by compiler.

Static Scheduling Scheme. If a macro-task graph in a target hierarchy has only data dependencies, the static scheduling is applied to reduce data transfer, synchronization and scheduling overheads.

In the static scheduling, the assignment of macro-tasks to processor clusters or processor elements is determined at compile-time. Therefore, each OpenMP "SECTION" needs only the macro-tasks that should be executed in the order predetermined by static scheduling algorithms CP/DT/MISF, DT/CP and ETF/CP. In other words, the compiler generates different program to each threads as shown in the first layer of Fig.2. When this static scheduling is used, it is assumed that each thread is binded to a processor.

In the OSCAR compiler, all of those algorithms are applied to the same macro-task graph, and the best schedule is automatically chosen.

At runtime, each thread group needs to synchronize and transfer shared data among other thread groups in the same hierarchy to satisfy the data dependency among macro-tasks.

To realize the data transfer and synchronization, the following code generation scheme is used.

If a macro-task assigned to the thread group is Basic Block, only one thread in the thread group executes the Basic Block(BB) or Basic Pseudo Assignment(BPA). Therefore, the code for MT1(BPA) is written in an OpenMP "SECTION" for one thread as shown in the first layer in Fig.2. OpenMP "SECTIONS" for the other threads don't have the code for MT1(BPA).

A parallel loop, for example RB2, is decomposed into smaller loops like MT2_1 to MT2_(K+1) at compile-time and defined as independent MTs as shown in Fig.2. Each thread has a code for assigned RBs, or decomposed partial parallel loops. In this case, since the partial loops are different macro-tasks, barrier synchronization at the end of the original parallel loops is not required.

If a macro-task is a sequential loop to which data localization cannot be applied, only one thread or thread group executes it. The sequential loop, or a RB, assigned to a processor cluster has large execution cost, its body hierarchically decomposed into coarse grain tasks. In Fig.2, MT3 is a sequential loop and decomposed into MT3_1, MT3_2, and so on.

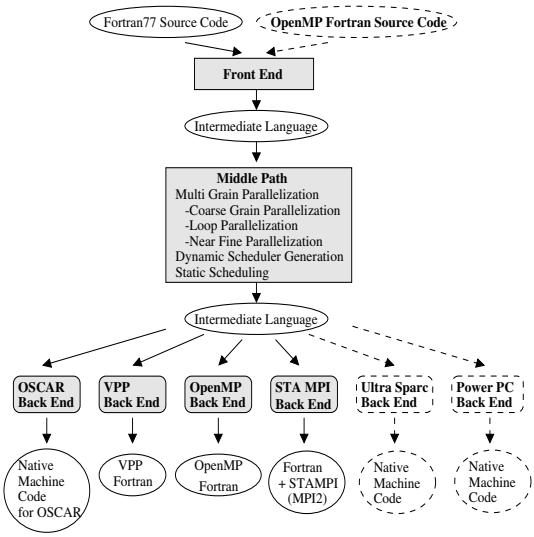


Fig. 3. Overview of OSCAR Fortran Compiler

4 Performance Evaluation

This section describes the optimization for exploiting coarse grain task parallelization by OSCAR Fortran Compiler and its performance for several programs in Perfect benchmarks and SPEC 95fp benchmarks on IBM RS6000 SP 604e High Node 8 processor SMP.

4.1 OSCAR Fortran Compiler

Figure 3 shows the overview of OSCAR Fortran Compiler. It consists of Front End, Middle Path and Back Ends. OSCAR Fortran Compiler has various Back Ends for different target multiprocessor systems like OSCAR distributed/shared memory multiprocessor system[37], Fujitsu's VPP supercomputer, UltraSparc, PowerPC, MPI-2 and OpenMP. OpenMP Back End used in this paper, which generates the parallelized Fortran source code with OpenMP directives. In other words, OSCAR Fortran Compiler is used as a preprocessor that transforms from an ordinary sequential Fortran program to OpenMP Fortran program for SMP machines.

4.2 Evaluated Programs

The programs used for performance evaluation are ARC2D in Perfect Benchmarks, SWIM, TOMCATV, HYDRO2D, MGRID in SPEC 95fp Benchmarks. ARC2D is an implicit finite difference code for analyzing fluid flow problems and

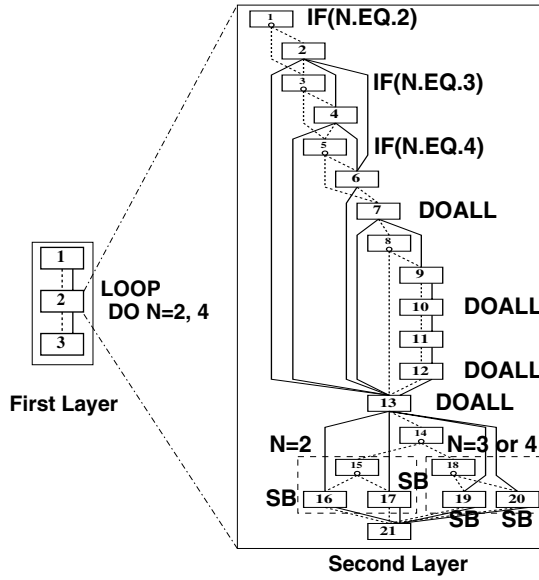


Fig. 4. Macro-flow graph of subroutine STEPFX in ARC2D

solves Euler equations. SWIM solves the system of shallow water equations using finite difference approximations. TOMCATV is a vectorized mesh generation program. HYDRO2D is a vectorizable Fortran program with double precision floating-point arithmetic. MGRID is the Multi-grid solver in 3D potential field.

As an example of the exploitation of coarse grain parallelism, parallelization of ARC2D is briefly explained. ARC2D has about 4500 statements including 40 subroutines. More than 90% of the execution time is spent in subroutine INTEGR. In the subroutine INTEGR, subroutines FILERX, FILERY, STEPFX, STEPFY consume 62% of the total execution time. Here, as an example, subroutine STEPFX is treated since it consumes 30% of the total execution time.

Macro Flow Graph of subroutine STEPFX is shown in Fig.4. The first layer in Fig.4 consists of three macro-tasks. Macro-Task(MT) 2 in the first layer is a sequential loop having three iterations with large processing time.

To minimize the processing time of MT2, the second layer macro-tasks are defined for a loop body of MT2, such as twelve Basic Blocks (MT1∼6, 8, 9, 11, 14, 15, 18), four DOALL loops (MT7, 10, 12, 13) and four Subroutine Blocks (MT16, 17, 19, 20).

In the second layer of Fig.4, MT groups, namely, MTs 1 and 2, MTs 3 and 4, MTs 5 and 6, MTs 15, 16 and 17 and MTs 18, 19 and 20, are executed depending on a value of loop index N. For example, MT2 is executed by the result of conditional branch inside MT1 (a Basic Block) when the value of loop index N is 2. MT4 is executed when N=3. MT6 is executed when N=4. By the

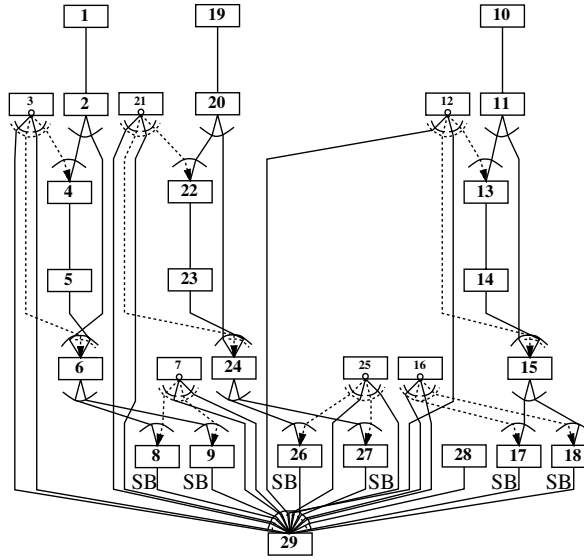


Fig. 5. Optimized macro-task graph of subroutine STEPFX in ARC2D

result of conditional branch inside MT14 depending on loop index, MT15,16 and 17 are executed when N=2, and MT18,19 and 20 are executed when N=3 and 4. Also, MT15 and 18 in the second layer of Fig.4 are Basic Blocks having conditional branches depending on the input variable from files. By the result of conditional branches inside MT 15 and 18, either MTs 16 and 19 or MTs 17 and 20 are executed.

At first, loop unrolling is applied to MT2 having 3 iterations, in the first layer of Fig.4. As the result of loop unrolling, OSCAR compiler can remove conditional branches inside MT 1, 3, 5 and 14 in the second layer of Fig.4 depending on loop index N. By loop unrolling, SB16, 17, 19 and 20 in the second layer are transformed to SB8, 9, 17, 18, 26 and 27 in Fig.5 . Either macro-task group composed of SBs 8, 17 and 26 or SBs 9, 18 and 27 in Fig.5 is executed by conditional branches inside MT7, 16 and 25. As the result of interprocedure analysis to these subroutine blocks, subroutine blocks inside each group can be processed in parallel. Figure 5 shows the optimized Macro-Task Graph.

OSCAR compiler applies the similar optimizations to other subroutines called from subroutine INTEGR. In addition, the inline expansion are applied to the subroutine calls in subroutine INTEGR for extracting more coarse grain parallelism.

Figure 6 shows the optimized Macro-Task Graph for the subroutine INTEGR.

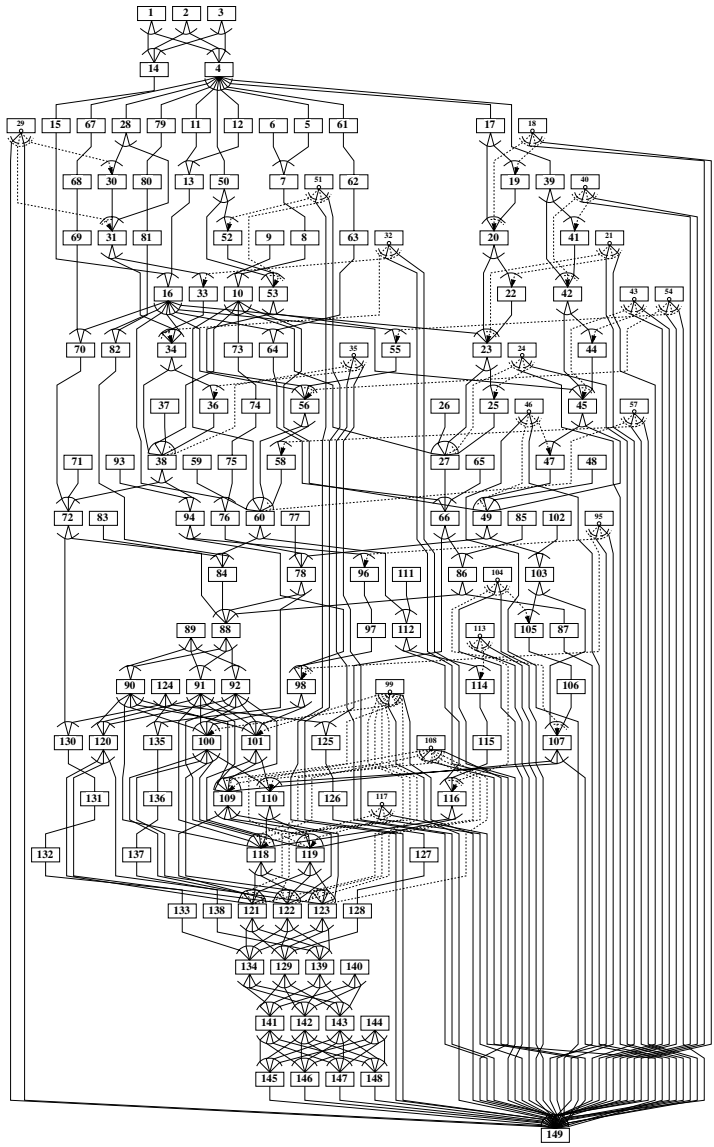


Fig. 6. Optimized macro-task graph of subroutine INTEGR in ARC2D.

4.3 Architecture of IBM RS6000 SP

RS6000 SP 604e High Node used for the evaluation is a SMP server having eight PowerPC 604e (200 MHz). Each processor has 32 KB L1 instruction and data caches and 1 MB L2 unified cache. The shared main memory is 1 GB.

4.4 Performance on RS6000 SP 604e High Node

In this evaluation, a coarse grain parallelized program automatically generated by OSCAR compiler is compiled by IBM XL Fortran compiler version 5.1[38] and executed on 1 through 8 processors of RS6000 SP 604e High Node. The performance of OSCAR compiler with XL Fortran compiler is compared with native IBM XL automatic parallelizing Fortran compiler[39]. In the compilation by a XL Fortran, maximum optimization option "-qsmp=auto -O3 -qmaxmem=-1 -qhot" is used.

Figure 7(a) shows speed-up ratios for ARC2D by the proposed coarse grain task parallelization scheme in OSCAR compiler and the automatic loop parallelization by XL Fortran compiler. The sequential processing time for ARC2D was 77.5s and parallel processing time by XL Fortran version 5.1 compiler using 8 processors was 60.1s. On the other hand, the execution time of coarse grain parallel processing using 8 processors by OSCAR Fortran compiler combined with XL Fortran compiler was 23.3s. In other words, OSCAR compiler gave us 3.3 times speed up against sequential processing time and 2.6 times speed up against native XL Fortran compiler for 8 processors. The number of loop iterations consuming large execution time in ARC2D has a small number of iterations. Therefore, only use of the loop level parallelism cannot attain large performance improvement even if more than 4 or 5 processors are used. The performance difference between OSCAR compiler and XL Fortran compiler in Fig.7(a) come from the coarse grain parallelism detected by OSCAR compiler.

Next, Fig.7(b) shows speed-up ratio for SWIM. The sequential execution time of SWIM was 551s. While the automatic loop parallel processing time using 8 processors by XL Fortran needed 112.7s and 4.9 times speed-up was attained, coarse grain task parallel processing by OSCAR Fortran compiler required only 61.1s and gave us 9.0 times speed-up by the effective use of distributed caches against the sequential execution time and 1.8 times speed-up compared with XL Fortran compiler.

Figure 7(c) shows speed-up ratio for TOMCATV. The sequential execution time of TOMCATV was 691s. The parallel processing time using 8 processors by XL Fortran was 484s and 1.4 times speed-up against sequential execution time. On the other hand, the coarse grain parallel processing using 8 processors by OSCAR Fortran compiler was 154s and gave us 4.5 times speed-up against sequential execution time. OSCAR Fortran compiler also gave us 3.1 times speed up compared with XL Fortran compiler using 8 processors.

Figure 7(d) shows speed-up in HYDRO2D. The sequential execution time of Hydro2d was 1036s. While XL Fortran gave us 4.7 times speed-up (221s) using 8 processors compared with the sequential execution time, OSCAR Fortran compiler gave us 8.1 times speed-up (128s) compared with sequential execution time.

Finally, Fig.7(e) shows speed-up ratio for MGRID. The sequential execution time of MGRID was 658s. For this application, XL Fortran compiler attains 4.2 times speed-up, or processing time of 157s, using 8 processors. Also, OSCAR compiler achieved 6.8 times speed up, or 97.4s. Namely, OSCAR Fortran

compiler gave us 1.6 times speed-up compared with XL Fortran compiler for 8 processors.

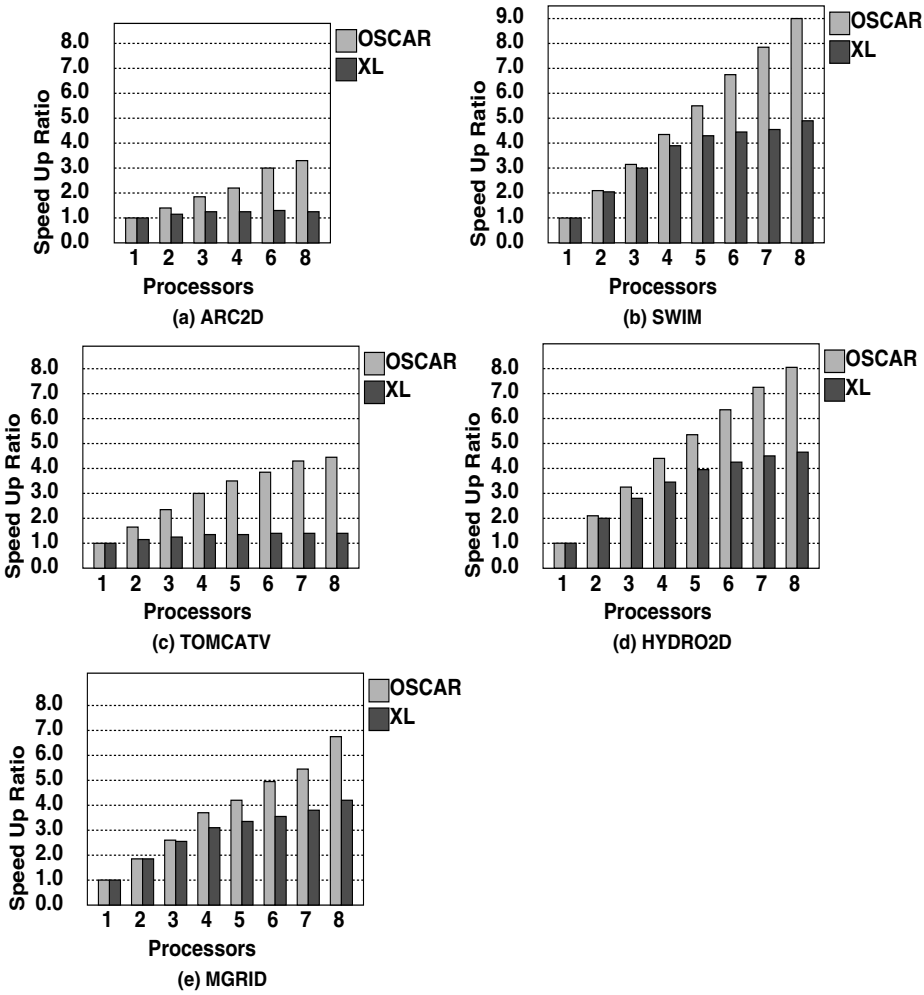


Fig. 7. Speed-up of several benchmarks on RS6000

5 Conclusions

This paper has presented the implementation scheme of the automatic coarse grain task parallel processing using OpenMP API as an example of realization and its performance on an off the shelf SMP machine.

OSCAR compiler generates coarse grain parallelized code which forks threads only once at the beginning of a program and joins only once at the end to minimize the overhead though hierarchical coarse grain task parallelism are automatically exploited.

In the performance evaluation, OSCAR compiler with XL Fortran compiler gave us scalable speed up for application programs in Perfect and SPEC 95fp benchmarks and significant speed-up compared with native XL Fortran compiler, such as 2.6 times for ARC2D, 1.8 times for SWIM, 3.1 times for TOMCATV, 1.7 times for HYDRO2D and 1.6 times for MGRID when the 8 processors are used. In other words, OSCAR Fortran compiler can boost the performance of XL Fortran compiler, which is one of the best commercially available loop parallelizing compilers for IBM RS6000 SP 604e High Node, easily using coarse grain parallelism with low overhead.

Currently, the authors are planning to evaluate the proposed coarse grain task parallel processing scheme on other SMP machines using OpenMP and improving the implementation scheme to further reduce dynamic scheduling overhead and data transfer overhead using data localization scheme to use distributed cache efficiently.

References

[1] M. Wolfe. High Performance Compilers for Parallel Computing. *Addison-Wesley*, 1996.

[2] U. Banerjee. Loop Parallelization. *Kluwer Academic Pub.*, 1994.

[3] U. Barnerjee. Dependence Analysis for Supercomputing. *Kluwer Pub.*, 1989.

[4] P. Petersen and D. Padua. Static and Dynamic Evaluation of Data Dependence Analysis. *Proc. Int'l conf. on supemputing*, Jun. 1993.

[5] W. Pugh. The OMEGA Test: A Fast and Practical Integer Programming Algorithm for Dependence Alysis. *Proc. Supercomputing'91*, 1991.

[6] M. R. Haghighat and C. D. Polychronopoulos. *Symbolic Analysis for Parallelizing Compliers.* Kluwer Academic Publishers, 1995.

[7] P. Tu and D. Padua. Automatic Array Privatization. *Proc. 6th Annual Workshop on Languages and Compilers for Parallel Computing*, 1993.

[8] M. Wolfe. Optimizing Supercompilers for Supercomputers. *MIT Press*, 1989.

[9] D. Padua and M. Wolfe. Advanced Compiler Optimizations for Supercomputers. *C.ACM*, 29(12):1184–1201, Dec. 1986.

[10] Polaris. http://polaris.cs.uiuc.edu/polaris/.

[11] R. Eigenmann, J. Hoeflinger, and D. Padua. On the Automatic Parallelization of the Perfect Benchmarks. *IEEE Trans. on parallel and distributed systems*, 9(1), Jan. 1998.

[12] L. Rauchwerger, N. M. Amato, and D. A. Padua. Run-Time Methods for Parallelizing Partially Parallel Loops. *Proceedings of the 9th ACM International Conference on Supercomputing, Barcelona, Spain*, pages 137–146, Jul. 1995.

[13] M. W. Hall, B. R. Murphy, S. P. Amarasinghe, S. Liao, , and M. S. Lam. Interprocedural Parallelization Analysis: A Case Study. *Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing (LCPC95)*, Aug. 1995.

[14] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, 1996.
[15] S. Amarasinghe, J. Anderson, M. Lam, and C. Tseng. The SUIF Compiler for Scalable Parallel Machines. *Proc. of the 7th SIAM conference on parallel processing for scientific computing*, 1995.
[16] M. S. Lam. Locallity Optimizations for Parallel Machines. *Third Joint International Conference on Vector and Parallel Processing*, Nov. 1994.
[17] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam. Data and Computation Transformations for Multiprocessors. *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Processing*, Jul. 1995.
[18] H. Han, G. Rivera, and C.-W. Tseng. Software Support for Improving Locality in Scientific Codes. *8th Workshop on Compilers for Parallel Computers (CPC'2000)*, Jan. 2000.
[19] G. Rivera and C.-W. Tseng. Locality Optimizations for Multi-Level Caches. *Super Computing '99*, Nov. 1999.
[20] A. Yoshida, K. Koshizuka, M. Okamoto, and H. Kasahara. A Data-Localization Scheme among Loops for each Layer in Hierarchical Coarse Grain Parallel Processing. *Trans. of IPSJ*, 40(5), May. 1999.
[21] PROMIS. http://www.csrd.uiuc.edu/promis/.
[22] C. J. Brownhill, A. Nicolau, S. Novack, and C. D. Polychronopoulos. Achieving Multi-level Parallelization. *Proc. of ISHPC'97*, Nov. 1997.
[23] Parafrase2. http://www.csrd.uiuc.edu/parafrase2/.
[24] M. Girkar and C. Polychronopoulos. Optimization of Data/Control Conditions in Task Graphs. *Proc. 4th Workshop on Languages and Compilers for Parallel Computing*, Aug. 1991.
[25] X. Martorell, E. Ayguade, N. Navarro, J. Corbalan, M. Gozalez, and J. Labarta. Thread Fork/Join Techniques for Multi-level Parllelism Exploitation in NUMA Multiprocessors. *ICS'99 Rhodes Greece*, 1999.
[26] E. Ayguade, X. Martorell, J. Labarta, M. Gonzalez, and N. Navarro. Exploiting Multiple Levels of Parallelism in OpenMP: A Case Study. *ICPP'99*, Sep. 1999.
[27] OpenMP: Simple, Portable, Scalable SMP Programming http://www.openmp.org/.
[28] L. Dagum and R. Menon. OpenMP: An Industry Standard API for Shared Memory Programming. *IEEE Computational Science & Engineering*, 1998.
[29] H. K. et al. A Multi-grain Parallelizing Compilation Scheme on OSCAR. *Proc. 4th Workshop on Languages and Compilers for Parallel Computing*, Aug. 1991.
[30] M. Okamoto, K. Aida, M. Miyazawa, H. Honda, and H. Kasahara. A Hierarchical Macro-dataflow Computation Scheme of OSCAR Multi-grain Compiler. *Trans. IPSJ*, 35(4):513–521, Apr. 1994.
[31] H. Kasahara, M. Okamoto, A. Yoshida, W. Ogata, K. Kimura, G. Matsui, H. Matsuzaki, and H.Honda. OSCAR Multi-grain Architecture and Its Evaluation. *Proc. International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*, Oct. 1997.
[32] H. Kasahara, H. Honda, M. Iwata, and M. Hirota. A Macro-dataflow Compilation Scheme for Hierarchical Multiprocessor Systems. *Proc. Int'l. Conf. on Parallel Processing*, Aug. 1990.
[33] H. Honda, M. Iwata, and H. Kasahara. Coarse Grain Parallelism Detection Scheme of Fortran programs. *Trans. IEICE (in Japanese)*, J73-D-I(12), Dec. 1990.

[34] H. Kasahara. Parallel Processing Technology. *Corona Publishing*, Tokyo (in Japanese), Jun. 1991.
[35] H. Kasahara, H. Honda, and S. Narita. Parallel Processing of Near Fine Grain Tasks Using Static Scheduling on OSCAR. *Proc. IEEE ACM Supercomputing'90*, Nov. 1990.
[36] J. E. Moreira and C. D. Polychronopoulos. Autoscheduling in a Shared Memory Multiprocessor. *CSRD Report No.1337*, 1994.
[37] H. Kasahara, S. Narita, and S. Hashimoto. OSCAR's Architecture. *Trans. IEICE (in Japanese)*, J71-D-I(8), Aug. 1988.
[38] IBM. *XL Fortran for AIX Language Reference*.
[39] D. H. Kulkarni, S. Tandri, L. Martin, N. Copty, R. Silvera, X.-M. Tian, X. Xue, and J. Wang. XL Fortran Compiler for IBM SMP Systems. *AIXpert Magazine*, Dec. 1997.

Compiler Synthesis of Task Graphs for Parallel Program Performance Prediction

Vikram Adve[1] and Rizos Sakellariou[2]

[1] Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, U.S.A.
vadve@cs.uiuc.edu
[2] Department of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, U.K.
rizos@cs.man.ac.uk

1 Introduction

Task graphs and their equivalents have proved to be a valuable abstraction for representing the execution of parallel programs in a number of different applications. Perhaps the most widespread use of task graphs has been for performance modeling of parallel programs, including quantitative analytical models [3, 19, 25, 26, 27], theoretical and abstract analytical models [14], and program simulation [5, 13]. A second important use of task graphs is in parallel programming systems. Parallel programming environments such as PYRROS [28], CODE [24], HENCE [24], and Jade [20] have used task graphs at three different levels: as a programming notation for expressing parallelism, as an internal representation in the compiler for computation partitioning and communication generation, and as a runtime representation for scheduling and execution of parallel programs. Although the task graphs used in these systems differ in representation and semantics (e.g., whether task graph edges capture purely precedence constraints or also dataflow requirements), there are close similarities. Perhaps most importantly, they all capture the parallel structure of a program separately from the sequential computations, by breaking down the program into computational "tasks", precedence relations between tasks, and (in some cases) explicit communication or synchronization operations between tasks.

If task graph representations could be constructed automatically, via compiler support, for common parallel programming standards such as Message-Passing Interface (MPI), High Performance Fortran (HPF), and OpenMP, the techniques and systems described above would become available to a much wider range of programs than they are currently. Within the context of the POEMS project [4], we have developed a task graph based application representation that is used to support modeling of the end-to-end performance characteristics of a large-scale parallel application on a large parallel system, using a combination of analytical, simulation and hybrid models, and models at multiple levels of resolution for individual components. This paper describes how parallelizing compiler technology can be used to automate the process of constructing this task graph representation for HPF programs compiled to MPI (and, in the near

S.P. Midkiff et al. (Eds.): LCPC 2000, LNCS 2017, pp. 208–226, 2001.

future, for existing MPI programs directly). In particular, this paper makes the following contributions:

- We describe compiler techniques to derive a static, *symbolic* task graph representing the MPI code generated for a given HPF program. A key aspect of this process is the use of symbolic integer sets and mappings to capture a number of dynamic task instances or edge instances as a single node or edge at compile time. These techniques allow the compiler to describe sophisticated computation partitionings and communication and synchronization patterns in symbolic terms.
- We describe how standard analysis techniques can be used to condense the task graph and simplify control flow, whenever less than fully detailed information suffices (as in many performance modeling applications in practice).
- Finally, we describe an approach to instantiate a dynamic task graph representation from the static task graph, based on a novel use of code generation from symbolic integer sets.

In addition to the above techniques, which to our knowledge are new, the compiler also uses standard techniques to compute symbolic scaling functions for task computation times and message communication volumes.

The techniques described above have been implemented in the Rice dHPF compiler system, which compiles HPF programs to MPI for message-passing systems using aggressive techniques for computation partitioning and communication optimization [1, 6, 22]. This implementation was recently used in a joint project with the parallel simulation group at UCLA to improve the scalability of simulation of message passing programs [5]. In that work, we showed how compiler information captured in the task graph can be used to reduce the memory and time requirements for simulating message-passing programs in detail. In the context of the present paper, these results illustrate the potential importance of automatically constructing task graphs for widely used programming standards.

The next section briefly describes the key features of our static and dynamic task graph representations. Section 3 is the major technical section, which presents the compiler techniques described above to construct the task graph representations. Section 4 provides some results about the structure of the compiler-generated task graphs for simple programs and illustrates how task graphs have been used to improve the scalability of simulation, as mentioned above. We conclude with a brief overview of related work (Section 5) and a discussion of future plans (Section 6).

2 Background: The Task Graph Representation

The POEMS project [4] aims to create a performance modeling environment for the end-to-end modeling of large parallel applications on complex parallel and distributed systems. The wide range of modeling techniques supported by POEMS, and the goal of integrating multiple modeling paradigms make it challenging, if not impossible, for the end-user to generate the required workload

information manually for a large-scale application. Thus, since the conception of the project, it has been deemed essential to use compiler support to simplify and partially automate the process of constructing the workload information. To achieve this, we have designed a common task graph based program representation that provides a *uniform* platform for capturing the parallel structure of a program as well as its associated workloads for different modeling techniques. This representation uses two flavors of a task graph, the *static task graph* and the *dynamic task graph*. Its design is described in detail in [2] and is briefly summarized here. The specific information we aim to collect for a given program includes: (1) The detailed computation partitioning and communication structure of the program, described in symbolic terms. (2) Source code for individual tasks to support source-code-driven uses such as detailed program-driven simulation of memory hierarchy performance. (3) Scaling functions that describe how computation and communication scale as a function of program inputs and processor configuration. (4) Optionally, the detailed dynamic behavior of the parallel program, for a specified program input and processor configuration.

The Static Task Graph: The static task graph (STG) captures the static parallel structure of a program and is defined only by the program *per se*. Thus, it is independent of runtime input values, intermediate program results, and processor configuration. Each node (or task) of the graph may represent one of the following main types: control flow statements such as loops and branches, procedure calls, communication, or pure computation. Edges between nodes may denote control flow within a processor or synchronization between different processors (due to communication tasks). For example, the STG for a simple parallel program is shown in Figure 1, and is explained in more detail in the next section.

A key aspect of the STG is that each node represents a set of instances of the task, one per processor that executes the task at runtime. Similarly, an edge in the STG actually represents a set of edge instances connecting pairs of dynamic node instances. We use symbolic integer sets to describe the set of instances for a given node, e.g., a task executed by P processors would be described by the set: $\{[p] : 0 \leq p \leq P-1\}$, and symbolic integer mappings to describe the edge instances, e.g., an edge from a `SEND` task on processor p to a `RECV` task on processor $q = p+1$ (i.e., each processor sends data to its right neighbor, if any) would be described by the mapping: $\{[p] \rightarrow [q] : q = p+1 \wedge 0 \leq p < P-1\}$. This kind of mapping enables precise symbolic representations of arbitrary regular communication patterns. Irregular patterns (i.e., data-dependent patterns that cannot be determined until runtime) have to be represented as an all-to-all communication, which is the best that can be done statically.

To capture high level communication patterns where possible (e.g., shift, pipeline, broadcast, etc. [21]) we group the communication operations in the program into related groups, each describing a single "logical communication event". A communication event descriptor, kept separate from the STG, captures all information about a single communication event. This includes the communication pattern, the set of communication tasks involved, and a symbolic

expression for the communication size. The CPU components of each communication event are represented explicitly as communication tasks in the STG, allowing us to use task graph edges between these tasks to explicitly capture the synchronization implied by the underlying communication calls. The number of communication nodes and edges depends on the communication pattern and also on the type of message passing calls used. This technique does not work for MPI receive operations that use a wildcard message tag (because the matching send cannot be easily identified). It does work for receive operations that use a wildcard for the sending processor, but the symbolic mapping on the communication edges may be an all-to-all mapping (for the processors that execute the send and receive statements). Making the communication tasks explicit in the STG has proved valuable also because it allows us to describe arbitrary interleavings (i.e., overlaps) of communication and computation tasks on individual processors and across processors.

In addition to the symbolic sets and mappings above, each node and communication event in the STG includes a symbolic scaling function that describes how the task computation time or the message size scales as a function of program variables. Finally, note that the STG of a program containing multiple procedures is represented as a number of unconnected graphs, each corresponding to a single procedure. Each call site is represented by a CALL task that identifies the called procedure by name.

The Dynamic Task Graph: The dynamic task graph (DTG) is a directed acyclic graph that captures the execution behavior of a program on a given input and given processor configuration. This representation is important for detailed performance modeling because it corresponds closely with the actual execution behavior being modeled by a particular program performance model (whether using detailed simulation or abstract analytical models).

The nodes of a dynamic task graph are computational tasks and individual communication tasks. In particular, the DTG does not contain control flow nodes (loops, branches, jumps, and jump targets). It can be thought of as being instantiated from the static task graph by unrolling all the loops, resolving all the branches, and instantiating all the instances of parallel tasks, edges, and communication events.

There are two approaches to making this representation tractable for large-scale programs, and these approaches can be combined: (1) we can condense tasks allocated to a process between synchronization points so that only (relatively) coarse-grain parallel tasks are explicitly represented, and (2) if necessary, we can compute the dynamic task graph "on the fly," rather than precomputing it and storing it offline. We describe techniques to automatically condense the task graph in Section 3.2. The approach to instantiate the task graph on-the-fly is outside the scope of this paper, but is a direct extension of the compile-time instantiation of the DTG described in Section 3.3.

3 Compiler Techniques for Synthesizing the Task Graphs

As noted in the Introduction, there are three major aspects to synthesizing our task graph representation for a parallel program: (1) synthesizing the static, symbolic task graph; (2) condensing the task graph; and (3) optionally instantiating a dynamic task graph representing an execution on a particular program input. Each of these steps relies on information about the message-passing program gathered by the compiler, although for many programs the third step can be performed purely by inspecting the static task graph, as explained in Section 3.3. These steps are described in detail in the following three subsections.

3.1 Synthesizing the Static Task Graph

Four key steps need be performed in synthesizing the static task graph (STG): (1) generating computation and control-flow nodes; (2) generating communication tasks for each logical communication event; (3) generating symbolic sets describing the processors that execute each task, and symbolic mappings describing the pattern of communication edges; and (4) eliminating excess control flow edges.

Generating computation and control-flow nodes in the STG can be done in a single preorder traversal of the internal representation for each procedure; in our case, the representation is an Abstract Syntax Tree (AST). STG nodes are created as appropriate statements are encountered in the AST. Thus, program statements, such as `DO`, `IF`, `CALL`, `PROGRAM/FUNCTION/SUBROUTINE`, `STOP/RETURN`, trigger the creation of a single node in the graph; encountering one of the first two also leads to the creation of an ENDDO-NODE or an ENDIF-NODE, a THEN-NODE and an ELSE-NODE, respectively. Any contiguous sequence of other computation statements *that are executed by the same set of processors* are grouped into a single computational task (contiguous implies that they are not interrupted by any of the above statements or by communication).

Identifying statements that are computed by the same set of processors is a critical aspect of the above step. This information is derived from the computation partitioning phase of the compiler and is translated into a symbolic integer set [1] that is included with each task. By having a general representation of the set of processors associated with each task, our representation can describe sophisticated computation partitioning strategies. The explicit set representation also enables us to check equality by direct set operations, in order to group statements into tasks. These processors sets are also essential for the fourth step listed above, namely eliminating excess control-flow edges between tasks, so as to expose program parallelism. In particular, a control flow edge is retained between two tasks *only if* the intersection of their processor sets is not empty. Otherwise, the sink node is connected to its most immediate ancestor in the STG for which the result of this intersection is a non-empty set.

When the first communication statement for a logical communication event is encountered, the communication event descriptor and all the communication tasks that are pertinent to this single event are built. The processor mappings for the synchronization between the tasks are also built at this time.

```
CHPF$ DISTRIBUTE A(*,BLOCK)
      DO I=2,N
         DO J=1,M-1
            A(I,J) = A(I-1,J+1)
         ENDDO
      ENDDO
```

(a) HPF source code fragment

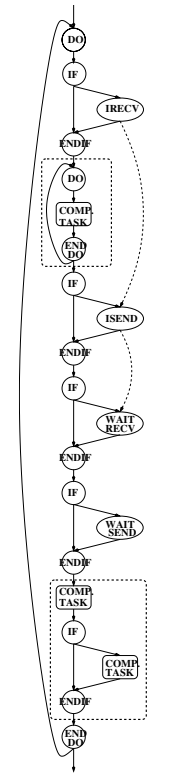
```
blk = block size per processor
DO I=2,N
    IF (myid < P-1)
        irecv B(i, myid*blk+blk+1) from myid+1

    ! Execute local iterations of j-loop
    DO J=myid*blk+1, min(myid*blk+blk-1, M-1)
        A(I,J) = A(I-1,J+1)
    ENDDO

    IF (myid > 0) isend B(i, myid*blk+1) to myid-1
    IF (myid < P-1) wait-recv
    IF (myid > 0) wait-send

    ! Execute non-local iterations of j-loop
    J=myid*blk+blk
    IF (J <= M-1)
        A(I,J) = A(I-1,J+1)
ENDDO
```

(b) Unoptimized MPI code generated by dHPF



(c) Static task graph

Fig. 1. An example of generating the communication tasks.

For an explicit message-passing program, the computation partitioning information can be derived by analyzing the control-flow expressions that depend on process id variables. The communication pattern information has to be extracted by recognizing the communication calls syntactically, analyzing their arguments, and identifying the matching send and receive calls. In principle, both the control-flow and the communication can be written in a manner that is too complex for the compiler to decipher, and some message passing programs will probably not be analyzable. But in most of the programs we have looked at, the control-flow idioms for partitioning the computation and the types of message passing operations that are used are fairly simple. We believe that the required analysis to construct the STG would be feasible with standard interprocedural symbolic analysis techniques [16].

To illustrate the communication information built by the compiler, consider the simple HPF example, which, along with the MPI parallel code generated by the dHPF compiler, are shown on the left-hand side of Figure 1. The parallelization of the code requires the boundary values of array `A` along the j dimension to

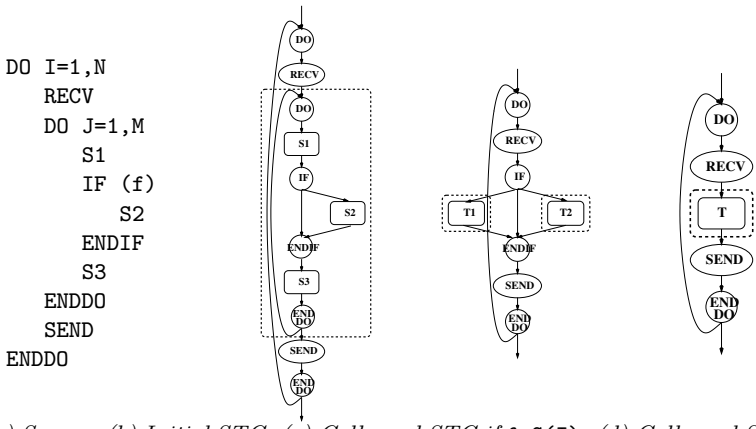
be communicated inside the `I` loop. (In practice, the compiler pipelines the communication in larger blocks by strip-mining the `I` loop [17] but that is omitted to simplify the example.) The corresponding STG is shown on the right-hand side of the figure. Solid lines represent control flow edges and dashed lines represent interprocessor synchronization. In this example, the compiler uses the non-blocking MPI communication primitives. The two dashed lines show that the `wait-recv` operation cannot complete until the `isend` is executed, and the `isend` cannot complete until the `irecv` is issued by the receiver (the latter is true because our target MPI library uses sender-side buffering).[1] Also, the compiler interleaves the communication tasks and computation so as to overlap waiting time at the `isend` with the computation of local loop iterations, i.e., the iterations that do not read or write any off-processor data. The use of explicit communication tasks within the task graph allows this overlap to be captured precisely in the task graph. The dashed edge between the `isend` and the `wait-recv` tasks is associated with the processor mapping: $\{[p_0] \rightarrow [q_0] : q_0 = p_0 - 1 \wedge 0 \leq q_0 < p - 1\}$, denoting that each processor receives data from its "right" neighbor, except the rightmost boundary processor. The other dashed edge has the inverse mapping, i.e., $q_0 = p_0 + 1$.

Finally, the compiler constructs the symbolic scaling functions for each task and communication event, using direct symbolic analysis of loop bounds and message sizes. For a communication event, the scaling function is simply an expression for the message size. For each DO-NODE the scaling function describes the number of iterations executed by each processor, as a function of processor id variables and other symbolic program variables. In the simple example above, the scaling functions for the two DO nodes are `N-1` and `min(myid*blk+blk-1,M-1) - (myid*blk+1) + 1`, respectively. For a computational task, the scaling function is a single parameter representing the workload corresponding to the task. At this stage of the task graph construction, no further handling (such as symbolic iteration counting for more complex loop bounds), takes place.

3.2 Condensing Nodes of the Static Task Graph

The process described above produces a first-cut version of the STG. For many typical modeling studies of parallel program performance, however, a less detailed graph will be sufficient. For instance, a coarse-grain modeling approach could assume that all operations of a single process between two communication points constitute a single task. In order to add this functionality, the compiler traverses the STG and marks contiguous nodes, connected by a control-flow edge, that do not include communication. Such sequences of nodes are then collapsed and replaced in the STG by a single *condensed* task. Note that such a task will have a single point of entry and a single point of exit. For example, the two large

[1] More precisely, the `isend` task should be broken into two tasks, one that performs local initialization and does not depend on the `irecv`, and a second one that can only be initiated once the `irecv` has been issued but does not block the local computation on the sending node. This would be simply require introducing additional communication tasks into the task graph.



(a) Source; (b) Initial STG; (c) Collapsed STG if `f=G(I)`*; (d) Collapsed STG if* `f=Y(J)`

Fig. 2. Collapsing Branches of the Static Task Graph.

dotted rectangles in Figure 1 (c) correspond to sequences of tasks that can be collapsed into a single condensed task.

To preserve precision, the computation of the scaling function of the new condensed task is of particular importance. Ignoring conditional control flow for the moment, this scaling function is the symbolic sum of the scaling functions of the individual collapsed tasks, each multiplied by the symbolic number of iterations of the surrounding loops, where appropriate (only if these loops are also collapsed).

In cases with conditional control flow, tasks can sometimes be condensed with no loss of accuracy, using sophisticated compiler analysis. For example, no accuracy will be lost in cases where all dynamic instances of the resulting collapsed task have identical computational time; that is, the workload expressions are free of any input parameters whose value changes for different instances of that task. In other cases, condensing would result in some loss of accuracy, and the goals of the modeling study should be used to dictate the degree to which tasks are collapsed together.

To illustrate, consider the code shown in Figure 2 (a). Let w_1, w_2, w_3 represent the workload (i.e., the scaling function) for statements `S1`, `S2` and `S3`, respectively. The initial version of the STG is shown in Figure 2 (b); the nodes inside the dotted rectangle are candidates for collapsing. Assuming that the function `f` in the `IF` depends on at least one of `I` or `J`, we distinguish between:

– If `f` is a function of `I` only, the `IF` statement can be moved outside the `J` loop and the `J` loop can be collapsed with no loss of accuracy. In this case, we are left with two separate tasks, representing the two possible versions of the `J` loop, as shown in part (c) of the figure. These two tasks have scaling functions given by $M \times (w_1 + w_2 + w_3)$, $M \times (w_1 + w_3)$, respectively.

- If `f` is a function of `J` only, the code can be condensed into a single task as shown in Figure 2(d). The scaling function of the task T would be $M \times w$, where w is the workload inside the `J` loop body per iteration of the `I` loop.
- Finally, if `f` is a function of both `I` and `J`, we can condense the code only by introducing a branching probability parameter. If $p(\texttt{I})$ represents the probability that `S2` will be executed for a given value of `I`, then the entire code inside the dotted rectangle can be condensed into a single task (as in part (d)) with a combined scaling function given by $\texttt{M} \times (w_1 + p(\texttt{I}) \times w_2 + w_3)$. Since this probabilistic expression for execution time can lead to inaccuracies, the decision to condense the task graph in such cases should depend on the goals of the modeling study.

The three cases can be differentiated using well-known but aggressive dataflow analysis. We note that the first two cases correspond directly to loop unswitching and identifying loop-invariant code respectively, except that only the static task graph is modified and the code itself is not transformed. A key point to note is that in the first two cases, there is no resulting loss of accuracy in condensing the task graph. For example, in the ASCI benchmark Sweep3D [4] used in Section 4, the one significant branch is in fact of the first type, which can be pulled out of the task and enclosing loops (the analysis would have to be interprocedural because the enclosing loops are not in the same procedure as the branch).

3.3 Instantiating the Dynamic Task Graph

As noted in Section 2, the dynamic task graph (DTG) is essentially an instantiation of the STG representing a single execution for a particular input. The DTG is an acyclic graph containing no control-flow nodes. The time for instantiating the DTG grows linearly with the number of task instances in the execution of the program, but much less computation per task is usually required for the instantiation than for the actual execution. This is an optional step that can be performed when required for detailed performance prediction.

The information required to instantiate the DTG varies significantly across programs. For a regular, non-adaptive code, the parallel execution behavior of the program can usually be determined directly from the program input (in which we include the processor configuration parameters). In such cases, the DTG can be instantiated directly from the STG once the program input is specified. In general, and particularly in adaptive codes, the parallel execution behavior (and therefore the DTG) may depend on *intermediate computational results* of the program. For example, this could happen in a parallel n-body problem if the communication pattern changed as the positions of the bodies evolved during the execution of the program. In the current work, we focus on the techniques needed to instantiate the DTG in the former case, i.e., that of regular non-adaptive codes. These techniques are also valuable in the latter case, but they must be applied at runtime when the intermediate values needed are known. The issues to be faced in that case are briefly discussed later in this section.

There are two main aspects to instantiating the DTG: (1) enumerating the outcomes of all the control flow nodes, effectively by unrolling the DO nodes and

resolving the dynamic instances of the branch nodes; and (2) enumerating the dynamic instances of each node and edge in the STG. These are discussed in turn below. Of these, the second step is significantly more challenging in terms of the compile-time techniques required, particularly for sophisticated message passing programs with general computation partitioning strategies and communication patterns.

Interpreting control-flow in the static task graph Enumerating the outcomes of all the control flow nodes in an execution can be accomplished by a symbolic interpretation of the control flow of the program for each process. First, we must enumerate loop index values and resolve the dynamic instances of branch conditions that have not been collapsed in the STG. This requires evaluating the values of these symbolic expressions. We can perform this evaluation directly *at compile-time* when these quantities are determined solely by input values, surrounding loop index variables, and processor id variables. Under these conditions, we know all the required variable values in the expressions, as follows. The input variable values are specified externally. The loop index values are explicitly enumerated for all DO nodes that are retained in the static task graph. The processor id variables are explicitly enumerated for each parallel task using the symbolic processor sets, as discussed below. Therefore, we can evaluate the relevant symbolic expressions for enumerating the control-flow outcomes.

We assumed above that key symbolic quantities were determined solely by input values, surrounding loop index variables, and processor id variables. These requirements only apply to those loop bounds and branch conditions that are retained in the collapsed static task graph (i.e., which affect the parallel task graph structure of the code), and *not* to loops and branches that have been collapsed because they only affect the internal computational results of a task. With the exception of a few common algorithmic constructs, we find these requirements to be satisfied by a fairly large class of regular scientific applications. For example, in a collection of codes including the three NAS application benchmarks (SP, BT and LU), an ASCI benchmark Sweep3D [4], and other standard data-parallel codes such as Erlebacher [1] and the SPEC benchmark Tomcatv, the sole exceptions were terminating conditions testing convergence in the outermost timestep loops. In such cases, we would rely on the user to specify a fixed number of time steps for which the program performance would be modeled.

More generally, and particularly for adaptive codes, we expect the parallel structure to depend on intermediate computational results. This would require generating the DTG on the fly, e.g., when performing program-driven simulation during which the actual computational results would be computed. In this case, the most efficient approach to synthesizing the DTG would be to use program slicing to isolate the computations that do affect the parallel control flow. (This is very similar to the use of slicing for optimizing parallel simulation as described in Section 4.) These extensions are outside the scope of this paper.

Enumerating the symbolic sets and mappings The second challenge is that we must enumerate all instances of each parallel task and each communication edge

between tasks. These instances are described by symbolic sets and mappings respectively. In the context of complex computation partitioning strategies and arbitrary communication patterns, this presents a much more difficult problem at compile-time (i.e., without executing the actual program).

```
CHPF$ distribute rsd(*,block,block,*)
CHPF$ distribute u(*,block,block,*)
CHPF$ distribute flux(*,block,block,*)
      do k = 2, nz-1
        do j = jst, jend
CHPF$    INDEPENDENT, NEW(flux)
         do indep_dummy_loop = 1, 1
           do i = 1, nx
             do m = 1, 5    !  ON HOME rsd(m,i-1,j,k) ⋃ rsd(m,i+1,j,k)
               flux(m,i,j,k) = F( u(m,i,j,k) )
             enddo
           enddo
           do i = ist, iend
             do m = 1, 5    !  ON HOME rsd(m,i,j,k)
               rsd(m,i,j,k) = G( flux(m,i-1,j,k), flux(m,i+1,j,k) )
             enddo
           enddo
         enddo
        enddo
      enddo
```

(a) Source code and computation partitioning

$$
\begin{aligned}
ProcsThatSend = \quad & \{[p_0,p_1] : \texttt{jst} \le 17p_1 + 17, \texttt{jend}, 65 \wedge 0 \le p_0 \le 3 \wedge 0 \le p_1 \le 3 \\
& \quad \wedge 17p_0 < \texttt{nx} \wedge 17p_1 < \texttt{jend} \wedge 3 \le \texttt{nz}\} \\
& \bigcup \{[p_0,p_1] : \texttt{jst} \le 17p_1 + 17, \texttt{jend}, 65 \wedge 0 \le p_0 \le 3 \wedge 0 \le p_1 \le 3 \\
& \quad \wedge 17p_0 < \texttt{nx} \wedge 17p_1 < \texttt{jend} \wedge 3 \le \texttt{nz} \wedge 2 \le \texttt{nx} \wedge 17p_1 < \texttt{jend}\} \\
SendToRecvProcsMap = \quad & \{[p_0,p_1] \rightarrow [q_0,q_1] : q_1 = p_1 \wedge 0, q_0 - 1 \le p_0 \le 3 \\
& \quad \wedge \texttt{jst} \le 17p_1 + 17, 65, \texttt{jend} \wedge 0 \le p_1 \le 3 \\
& \quad \wedge 17p_0 < \texttt{nx} \wedge 3 \le \texttt{nz} \wedge 17q_0 \le \texttt{nx} \wedge 17p_1 < \texttt{jend}\} \\
& \bigcup \{[p_0,p_1] \rightarrow [q_0,q_1] : 0, p_0 - 1 \le q_0 \le p_0 \le 3 \\
& \quad \wedge \texttt{jst} \le 17p_1 + 17, 65, \texttt{jend} \wedge 0 \le p_1 \le 3 \wedge 17p_0 < \texttt{nx} \\
& \quad \wedge 3 \le \texttt{nz} \wedge 2 + 17q_0 \le \texttt{nx} \wedge 17p_1 < \texttt{jend}\}
\end{aligned}
$$

(b) Processor sets and task mappings

```
subroutine ProcsThatSend(nx, nz, jst, jend)
integer nx, nz, jst, jend
if (jend >= 1 && jst <= jend && nz >= 3 && jst <= 65) then
  do p0 = 0,  min(intDiv(nx-1,17), 3)
    do p1 = max(intDiv(jst-17+16,17), 0),  min(intDiv(jend-1,17), 3)
      !  Emit SEND task for processor (p0,p1)
    enddo
  enddo
endif
```

(c) Parameterized code to enumerate the set `ProcsThatSend`

Fig. 3. Example from NAS LU illustrating processor sets and task mappings for communication tasks. Problem size: $65 \times 65 \times 65$; Processor configuration: 4×4.

For example, consider the loop fragment from the NAS LU benchmark shown in Figure 3. The compiler automatically chooses a sophisticated computation partitioning, denoted by the ON HOME descriptors for each statement in the figure. For example, the ON HOME descriptor for the assignment to `flux` indicates that the instance of the statement in iteration `(k,j,i,m)` should be executed by the processors that own either of the array elements `rsd(m,i-1,j,k)` or `rsd(m,i+1,j,k)`. This means that each boundary iteration of the statement will be replicated among the two adjacent processors. This replication eliminates the need for highly expensive inner-loop communication for the privatizable array `flux` [6]. Communication is still required for each reference to array u, all of which are coalesced by the compiler into a single logical communication event. The communication pattern is equivalent to two SHIFT operations in opposite directions. Part (b) of the figure shows the set of processors that must execute the SEND communication task ($ProcsThatSend$), as well as the mapping between processors executing the SEND and those executing the corresponding `wait-recv` ($SendToRecvProcsMap$). (Note that both these quantities are described in terms of symbolic integer sets, parameterized by the variables `jst`, `jend`, `nx` and `nz`.) Each of these sets combines the information for both SHIFT operations. Correctly instantiating the communication tasks and edges for such communication patterns in a *pattern-driven* manner can be difficult and error-prone, and would be limited to some predetermined class of patterns that is unlikely to include such complex patterns.

Instead, we develop a novel and general solution to this problem that is based on an unusual use of code generation from integer sets. In ordinary compilation, dHPF and other advanced parallelizing compilers use code generation from integer sets to synthesize loop nests that are executed *at runtime*, e.g., for a parallel loop nest or for packing and unpacking data for communication [1, 7, 8, 11]. If we could invoke the same capability but execute the generated loop nests *at compile-time*, we could use the synthesized loop nests to enumerate the required tasks and edges.

Implementing this approach, however, proved to be a non-trivial task. Most importantly, each of the sets is parameterized by several variables, including input variables and loop index variables (e.g., the two sets above are parameterized by `jst`, `jend`, `nx` and `nz`). This means that the set must be enumerated separately for each combination of these variable values that occurs during the execution of the original program. We solve this problem as follows. We first generate a subroutine for each integer set that we want to enumerate, and make the parameters arguments of the subroutine. Then (still at compile-time), we compile, link, and invoke this code in a separate process. The desired combinations of variable values for each node and edge are automatically available when interpreting the control-flow of the task graph as described earlier. Therefore, during this interpretation, we simply invoke the desired subroutine in the other process to enumerate the ids for a node or the id pairs for an edge.

To illustrate this approach, Figure 3(c) shows the subroutine generated to enumerate the elements of the set $ProcsThatSend$ described earlier. The loop

nest in this subroutine is generated directly from the symbolic integer set in part (b) of the figure. This loop nest enumerates the instances of the SEND task, which in this case is one task per processor executing the SEND. This subroutine is parameterized by `jst`, `jend`, `nx` and `nz`. In this example, these are all unique values determined by the program input. In general, however, these could depend on loop index variables of some outer loop and the subroutine has to be invoked for each combination of values of its arguments.

Overall, the use of code generated from symbolic sets enables us to support a broad class of computation partitionings and communication patterns in a uniform manner. This approach fits nicely with the core capabilities of advanced parallelizing compilers.

4 Status and Results

We have successfully implemented the techniques described above in the dHPF compiler. We have extended the dHPF compiler to synthesize a static task graph for the MPI code generated by dHPF, including the symbolic processor sets and mappings for communication tasks and edges, and the scaling functions for loop nodes. In computing the condensed static task graph, we collapse all DO-NODEs or sequences of computational tasks that do not contain any communication or any IF-NODEs (We would rely on user intervention to collapse IF-NODEs). We also compute the combined scaling function for the collapsed tasks.

We have also partially implemented the support to instantiate dynamic task graphs at compile-time. In particular, we are able to enumerate the task instances and control-flow edges. We also synthesize the code from symbolic integer-sets required to enumerate the edge mappings at compile-time. We do not yet link in this code to enumerate the edges at compile-time.

Because of the aggressive computation partitioning and communication strategies used by dHPF, capturing the resulting MPI code requires the full generality of our task graph representation. This gives us confidence that we can synthesize task graphs for a wide range of explicit message-passing programs as well (including all the ones we have examined so far).

In order to illustrate the size of the static task graph generated and the effectiveness of condensing the task graph, Table 1 lists some particulars for the STG produced by the dHPF compiler for three HPF benchmarks: Tomcatv (from SPEC92), jacobi (a simple 2D Jacobi iterator PDE solver), and expl (Livermore Loop # 18). The effect of condensing the task graph on reducing the number of loops (DO-NODE) and computational tasks (COMP-TASK) can be observed. After condensing, most of the remaining tasks are either IF-NODEs and dummy nodes (e.g., ENDIF-NODE, etc.) or communication tasks (which are never condensed), since we opted for a detailed representation of communication behavior, rather than compromise on the accuracy of the representation. The compiler generated task graphs for the above can be found at `http://www.cs.man.ac.uk/~rizos/taskgraph/`

	tomcatv		jacobi		expl	
Lines of source HPF program	227		64		94	
Lines of output parallel MPI program	1850		1156		3722	
	1st pass	condensed	1st pass	condensed	1st pass	condensed
Total number of tasks	247	193	122	83	225	174
# COMM-NODE	54	54	36	36	114	114
# DO-NODE	18	3	13	1	17	3
# COMP-TASK	39	20	16	5	23	6

Table 1. Size of STG for various example codes before and after condensing.

The most important application of our compiler-synthesized task graphs to date has been for improving the state of the art of parallel simulation of message-passing programs [5]. Those results are briefly summarized here because they provide the best illustration of the correctness and benefits of the compiler-synthesized task graphs. This work was performed in collaboration with the parallel simulation group at UCLA, using MPI-Sim, a direct-execution parallel simulator for MPI programs [10].

The basic strategy in using the STG for program simulation is to generate an abstracted MPI program from the STG where all computation corresponding to a computational task is eliminated, *except those computations whose results are required* to determine the control-flow, communication behavior, and task scaling functions. We refer to the eliminated computations as *redundant computations* (from the point of view of performance prediction), and we use the task scaling functions to generate simple symbolic estimates for their execution time. The simulator can avoid simulating the redundant computations, and simply needs to advance the simulation clock by an amount equal to the estimated execution time of the computation. The simulator can even avoid allocating memory for program data that is referenced only in redundant computations.

The key to implementing this strategy lies in identifying non-redundant computations. To do so, we must first identify the values in the program that determine program performance. These are exactly those variable values that appear in the control-flow expressions, communication descriptors, and scaling functions (both for task times and for communication volume) of the STG. Thus, using the STG makes these values very easy to identify. We can then use a standard program slicing algorithm [18] to isolate the computations that affect these values. We then generate the simplified MPI code by including all the control-flow that appears in the static task graph, all the communication calls, and the non-redundant computations identified by program slicing. All the remaining (i.e., redundant) computations in each computational task are replaced with a single call to a special simulator delay function which simply advances the simulator clock by a specified amount. The argument to this function is a symbolic expression for the estimated execution time of the redundant computation. Note that the simulator continues to simulate the communication behavior in detail.

We have applied this methodology both to HPF programs (compiled to MPI by the dHPF compiler), and also to existing MPI programs (in the latter case,

Benchmark	% Error in prediction vs. measurement				
	#Procs = 4	8	16	32	64
Tomcatv	−5.44	15.75	11.79	8.50	9.27
Sweep3D	−7.01	−4.97	9.02	9.80	5.13
NAS SP, class A	−2.59		−1.24	7.11	6.10
NAS SP, class C			0.09	−14.01	−1.58

Table 2. Validation of the compiler-generated task graphs using MPI-Sim.

generating the abstracted MPI program by hand). The benchmarks include an HPF version of Tomcatv, and MPI versions of Sweep3D (a key ASCI benchmark) and NAS SP. Table 2 shows the percentage error in the execution times predicted by MPI-Sim using the simplified MPI code, compared with direct program measurement. As can be seen, the error was less than 16% in all cases tested, and less than 10% in most of these cases. This is important because the simplified MPI program can be thought of as simply an executable representation of the static task graph itself. These results show that the task graph abstraction very accurately captures the properties of the program that determine performance. We believe that the errors observed could be further reduced by applying more sophisticated techniques for estimating the execution time of redundant computations, particularly with simple estimates of cache behavior.

The benefits of using the task graph based simulation strategy were extremely impressive. For these benchmarks, the optimized simulator requires factors of 5 to 2000 less memory and up to a factor of 10 less time to execute than the original simulator. These dramatic savings allow us to simulate systems and problem sizes 10 to 100 times larger than is possible with the original simulator. Also, they have allowed us to simulate MPI programs for parallel architectures with hundreds of processors *faster than real-time*, and have made it feasible to simulate execution of programs on machines with 10,000+ processors. These results are described in more detail in [5].

5 Related Work

There is a large body of work on the use of task graphs for various aspects of parallel systems but very little work on synthesizing task graphs for general-purpose parallel programs. The vast majority of performance models that use task graphs as inputs generally do not specify how the task graph should be constructed but assume that this has been done [3, 14, 19, 25, 27]. The various compiler-based systems that use task graphs, namely PYRROS [28], CODE [24], HENCE [24], and Jade [20] construct task graphs by assuming special information from the programmer. In particular, PYRROS, CODE and HENCE all assume that the programmer specifies the task graph explicitly (CODE and HENCE actually use a graphical programming language to do so). In Jade, the programmer specifies input and output variables used by each task and the compiler uses this information to deduce the task dependences for the program.

The PlusPYR project [12] has developed a task graph representation that has some similarities with ours (in particular, symbolic integer sets and mappings for describing task instances and communication and synchronization rules), along with compiler techniques to synthesize these task graphs. The key difference from our work is that they start with a limited class of *sequential* programs (annotated to identify the tasks) and use dependence analysis to compute dependences between tasks, and then derive communication and synchronization rules from these dependences. Therefore, their approach is essentially a form of simple automatic parallelization. In contrast, our goal is to generate task graphs for existing parallel programs with no special program annotations and with explicit communication. A second major difference is that they assume a simple parallel execution model in which a task receives all inputs from other tasks in parallel and sends all outputs to other tasks in parallel. In contrast, we capture much more general communication behavior in order to support realistic HPF and MPI programs.

Parashar et al. [26] construct task graphs for HPF programs compiled by the Syracuse Fortran 90D compiler, but they are limited to a very simple, loosely synchronous computational model that would not support many message-passing and HPF programs in practice. In addition, their interpretive framework for performance prediction uses functional interpretation for instantiating a dynamic task graph, which is similar to our approach for instantiating control-flow. Like the task graph model, their interpretation and performance estimation are significantly simplified (compared with ours) because of the loosely synchronous computational model. For example, they do not need to capture sophisticated communication patterns and computation partitionings, as we do using code generation from integer sets.

Dikaiakos et al. [13] developed a tool called FAST that constructs task graphs from user-annotated parallel programs, performs advanced task scheduling and then uses abstract simulation of message passing to predict performance. The PACE project [25] proposes a language and programming environment for parallel program performance prediction. Users are required to identify parallel subtasks and computation and communication patterns. Finally, Fahringer [15], Armstrong and Eigenmann [9], Mendes and Reed [23] and many others have developed symbolic compile-time techniques for estimating execution time, communication volume and other metrics. The communication and computation scaling functions available in our static task graph are very similar to the symbolic information used by these techniques, and could be directly extended to support their analytical models.

6 Conclusion and Future Plans

In this paper, we described a methodology for automating the process of synthesizing task graphs for parallel programs, using sophisticated parallelizing compiler techniques. The techniques in this paper can be used *without* user intervention to construct task graphs message-passing programs compiled from HPF

source programs, and we believe they extend directly to existing message-passing (e.g., MPI) programs as well. Such techniques can make a large body of existing research based on task graphs and equivalent representations applicable for these widely used programming standards. Our immediate goals for the future are: (1) to demonstrate that the techniques described in this paper can be applied to message-passing programs (using MPI), by extracting the requisite computation partitioning and communication information; and (2) to couple the compiler-generated task graphs with the wide range of modeling approaches being used within the POEMS project, including analytical, simulation and hybrid models.

Acknowledgements: The authors would like to acknowledge the valuable input that several members of the POEMS project have provided to the development of the application representation. We are particularly grateful to the UCLA team and especially Ewa Deelman for her efforts in validating the simulations of the abstracted MPI codes generated by the compiler. This work was carried out while the authors were with the Computer Science Department at Rice University.

References

[1] V. Adve and J. Mellor-Crummey. Using Integer Sets for Data-Parallel Program Analysis and Optimization. In *Proc. of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

[2] V. Adve and R. Sakellariou. Application Representations for Multi-Paradigm Performance Modeling of Large-Scale Parallel Scientific Codes. *International Journal of High Performance Computing Applications*, 14(4), 2000.

[3] V. Adve and M. K. Vernon. A Deterministic Model for Parallel Program Performance Evaluation. Technical Report CS-TR98-333, Computer Science Dept., Rice University, December 1998. Also available at `http://www-sal.cs.uiuc.edu/~vadve/Papers/detmodel.ps.gz`.

[4] V. S. Adve, R. Bagrodia, J. C. Browne, E. Deelman, A. Dube, E. Houstis, J. R. Rice, R. Sakellariou, D. Sundaram-Stukel, P. J. Teller, and M. K. Vernon. POEMS: End-to-End Performance Design of Large Parallel Adaptive Computational Systems. *IEEE Trans. on Software Engineering*, 26(11), November 2000.

[5] V. S. Adve, R. Bagrodia, E. Deelman, T. Phan, and R. Sakellariou. Compiler-Supported Simulation of Highly Scalable Parallel Applications. In *Proceedings of Supercomputing '99*, Portland, Oregon, November 1999.

[6] V. Adve, G. Jin, J. Mellor-Crummey, and Q. Yi. High Performance Fortran Compilation Techniques for Parallelizing Scientific Codes. In *Proceedings of SC98: High Performance Computing and Networking*, Orlando, FL, November 1998.

[7] S. Amarasinghe and M. Lam. Communication optimization and code generation for distributed memory machines. In *Proc. of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, Albuquerque, NM, June 1993.

[8] C. Ancourt and F. Irigoin. Scanning polyhedra with do loops. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, April 1991.

[9] B. Armstrong and R. Eigenmann. Performance Forecasting: Towards a Methodology for Characterizing Large Computational Applications. In *Proc. of the Int'l Conf. on Parallel Processing*, pages 518–525, August 1998.

[10] R. Bagrodia, E. Deelman, S. Docy, and T. Phan. Performance prediction of large parallel applications using parallel simulation. In *Proc. 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, GA, 1999.

[11] P. Banerjee, J. Chandy, M. Gupta, E. Hodges, J. Holm, A. Lain, D. Palermo, S. Ramaswamy, and E. Su. The Paradigm compiler for distributed-memory multicomputers. *IEEE Computer*, 28(10):37–47, October 1995.

[12] M. Cosnard and M. Loi. Automatic Task Graph Generation Techniques. *Parallel Processing Letters*, 5(4):527–538, December 1995.

[13] M. Dikaiakos, A. Rogers, and K. Steiglitz. FAST: A Functional Algorithm Simulation Testbed. In *International Workshop on Modelling, Analysis and Simulation of Computer and Telecommunication Systems – Mascots '94*, 1994.

[14] D. L. Eager, J. Zahorjan, and E. D. Lazowska. Speedup versus Efficiency in Parallel Systems. *IEEE Trans. on Computers*, C-38(3):408–423, March 1989.

[15] T. Fahringer and H. Zima. A static parameter based performance prediction tool for parallel programs. In *Proceedings of the 1993 ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.

[16] P. Havlak. *Interprocedural Symbolic Analysis*. PhD thesis, Dept. of Computer Science, Rice University, May 1994. Also available as CRPC-TR94451 from the Center for Research on Parallel Computation and CS-TR94-228 from the Rice Department of Computer Science.

[17] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Evaluation of compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proc. of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[18] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. on Programming Languages and Systems*, 12:26–60, 1990.

[19] A. Kapelnikov, R. R. Muntz, and M. D. Ercegovac. A Modeling Methodology for the Analysis of Concurrent Systems and Computations. *Journal of Parallel and Distributed Computing*, 6:568–597, 1989.

[20] M. S. Lam and M. Rinard. Coarse-Grain Parallel Programming in Jade. In *Proc. Third ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 94–105, Williamsburg, VA, April 1991.

[21] J. Li and M. Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Trans. on Parallel and Distributed Systems*, 2(3):361–376, July 1991.

[22] J. Mellor-Crummey and V. Adve. Simplifying control flow in compiler-generated parallel code. *International Journal of Parallel Programming*, 26(5), 1998.

[23] C. Mendes and D. Reed. Integrated Compilation and Scalability Analysis for Parallel Systems. In *Proc. of the Int'l Conference on Parallel Architectures and Compilation Techniques*, Paris, October 1998.

[24] P. Newton and J. C. Browne. The CODE 2.0 Graphical Parallel Programming Language. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[25] E. Papaefstathiou, D. J. Kerbyson, G. R. Nudd, T. J. Atherton, and J. S. Harper. An Introduction to the Layered Characterization for High Performance Systems. Research Report 335, University of Warwick, December 1997.

[26] M. Parashar, S. Hariri, T. Haupt, and G. Fox. Interpreting the Performance of HPF/Fortran 90D. In *Proceedings of Supercomputing '94*, Washington, D.C., November 1994.

[27] A. Thomasian and P. F. Bay. Analytic Queueing Network Models for Parallel Processing of Task Systems. *IEEE Trans. on Computers*, C-35(12):1045–1054, December 1986.

[28] T. Yang and A. Gerasoulis. PYRROS: Static task scheduling and code generation for message passing multiprocessors. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

Optimizing the Use of High Performance Software Libraries*

Samuel Z. Guyer and Calvin Lin

The University of Texas at Austin, Austin, TX 78712

Abstract. This paper describes how the use of software libraries, which is prevalent in high performance computing, can benefit from compiler optimizations in much the same way that conventional programming languages do. We explain how the compilation of these informal languages differs from the compilation of more conventional languages. In particular, such compilation requires precise pointer analysis, domain-specific information about the library's semantics, and a configurable compilation scheme. We describe a solution that combines dataflow analysis and pattern matching to perform configurable optimizations.

1 Introduction

High performance computing, and scientific computing in particular, relies heavily on software libraries. Libraries are attractive because they provide an easy mechanism for reusing code. Moreover, each library typically encapsulates a particular domain of expertise, such as graphics or linear algebra, and the use of such libraries allows programmers to think at a higher level of abstraction. In many ways, libraries are informal domain-specific languages whose only syntactic construct is the procedure call. This procedural interface is significant because it couches these informal languages in a familiar form without imposing new syntax. Unfortunately, libraries are not viewed as languages by compilers. With few exceptions, compilers treat each invocation of a library routine the same as any other procedure call. Thus, many optimization opportunities are lost because the semantics of these informal languages are ignored.

As a trivial example, an invocation of the C standard math library's exponentiation function, `pow(a,b)`, can be simplified to `1` when its second argument is `0`. This paper argues that there are many such opportunities for optimization, if only compilers could be made aware of a library's semantics. These optimizations, which we term *library-level* optimizations, include choosing specialized library routines in favor of more general ones, eliminating unnecessary library calls, moving library calls around, and customizing the implementation of a library routine for a particular call site.

Figure 1 shows our system architecture for performing library-level optimizations [13]. In this approach, annotations capture semantic information about library routines. These annotations are provided by a library expert and placed in a separate file from the source code. This information is read by our compiler, dubbed the Broadway compiler, which performs source-to-source optimizations of both the library and application code.

* This work was supported in part by NSF CAREERS Grant ACI-9984660, DARPA Contract #F30602-97-1-0150 from the US Air Force Research Labs, and an Intel Fellowship.

S.P. Midkiff et al. (Eds.): LCPC 2000, LNCS 2017, pp. 227–243, 2001.

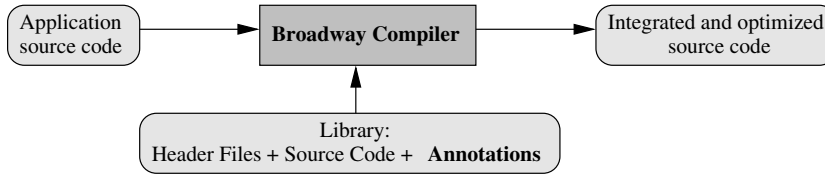


Fig. 1. Architecture of the Broadway Compiler system

This system architecture offers three practical benefits. First, because the annotations are separate from the library source, our approach applies to existing libraries and existing applications. Second, the annotations describe the library, not the application, so the application programmer does nothing more than use the Broadway Compiler in place of a standard C compiler. Finally, the non-trivial cost of writing the library annotations can be amortized over many applications.

This architecture also provides an important conceptual benefit: a clean separation of concerns. The compiler encapsulates all compiler analysis and optimization machinery, while the annotations describe all library knowledge and domain expertise. Together, the annotations and compiler free the applications programmer to focus on application design rather than on performing manual library-level optimizations.

The annotation language faces two competing goals. To provide simplicity, the language needs to have a small set of useful constructs that apply to a wide range of software libraries. At the same time, to provide power, the language has to convey sufficient information for the Broadway compiler to perform a wide range of optimizations. The remainder of this paper will focus on the annotation language and its requirements.

This paper makes the following contributions. (1) We define the distinguishing characteristics of library-level optimizations. (2) We describe the implications of these characteristics for implementing library-level optimizations in a compiler. (3) We present formulations of dataflow analysis and pattern matching that address these implications. (4) We extend our earlier annotation language [13] to support the configuration of the dataflow analyzer and pattern matcher.

2 Opportunities

This section characterizes library-level optimizations and their requirements.

Conceptually similar to traditional optimizations. Library-level optimizations are conceptually similar to traditional optimizations, which can be grouped into the following classes of optimizations. (1) *Eliminate redundant computations:* Examples include partial redundancy elimination, common subexpression elimination, loop-invariant code motion, and value numbering. (2) *Perform computations at compile-time:* This may be as simple as constant folding or as complex as partial evaluation (3) *Exploit special cases:* Examples include algebraic identities and simplifications, as well as strength reduction. (4) *Schedule code:* For example, exploit non-blocking loads and asynchronous

I/O operations to hide the cost of long-latency operations. (5) *Enable other improvements:* Use transformations such as inlining, cloning, loop transformations, and lowering the internal representation. These same categories form the basis of our library-level optimization strategy. In some cases, the library-level optimizations are identical to their classical counterparts. In other cases we need to reformulate the optimizations for the unique requirements of libraries.

Significant opportunities exist. Most libraries receive no support from traditional optimizers. For example, the code fragments in Figure 2 illustrate the untapped opportunities of the standard C Math Library. A conventional C compiler will perform three optimizations on the built-in operators: (1) strength reduction on the computation of `d1`, (2) loop-invariant code motion on the computation of `d3`, and (3) replacement of division with bitwise right-shift in the computation of `int1`. The resulting optimized code is shown in the middle fragment. However, there are three analogous optimization opportunities on the math library computations that a conventional compiler will not discover: (4) strength reduction of the power operator in the computation of `d2`, (5) loop-invariant code motion on the cosine operator in the computation of `d4`, and (6) replacement of sine divided by cosine with tangent in the computation of `d5`. The code fragment on the right shows the result of applying these optimizations.

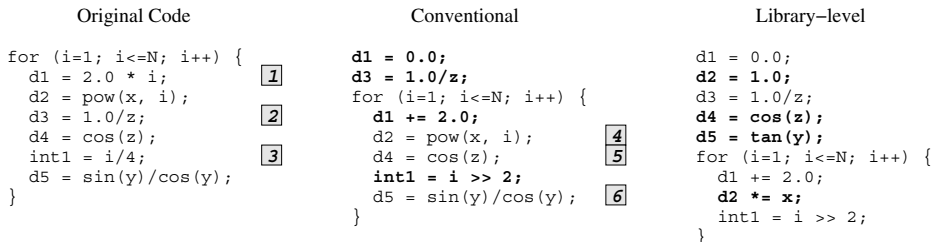


Fig. 2. A conventional compiler optimizes built-in math operators, but not math library operators.

Significantly, each application of a library-level optimization is likely to yield much greater performance improvement than the analogous conventional optimization. For example, removing an unnecessary multiplication may save a few cycles, while removing an unnecessary cosine computation may save hundreds or thousands of cycles.

Specialized routines are difficult to use. Many libraries provide a *basic* interface which provides basic functionality, along with an *advanced* interface that provides specialized routines that are more efficient in certain circumstances [14]. For example, the MPI message passing interface [11] provides 12 variations of the basic point-to-point communication operation. These advanced routines are typically more difficult to use than the basic versions. For example, MPI's Ready Send routines assume that the communicating processes have already been somehow synchronized. These specialized rou-

tines represent an opportunity for library-level optimization, as a compiler would ideally translate invocations of basic routines to specialized routines.

Domain-specific analysis is required. Most libraries provide abstractions that can be useful for performing optimizations. For example, the PLAPACK parallel linear algebra library [19] manipulates linear algebra objects indirectly though handles called *views*. A view consists of data, possibly distributed across processors, and an index range that selects some of the data. While most PLAPACK procedures are designed to accept any type of view, the actual parameters often have special distributions. Recognizing and exploiting these special distributions can yield significant performance gains [2]. For example, in some cases, calls to the general-purpose `PLA_Trsm()` routine can be replaced with calls to a specialized routine, `PLA_Trsm_Local()`, which assumes the matrix *view* resides completely on a single processor. This customized routine can run as much as three times faster [13].

The key to this optimization is to analyze the program to discover the special case matrix distributions. While compilers can perform many kinds of dataflow analysis, most compilers have no notion of "matrix," let alone PLAPACK's particular notion of matrix distributions. Thus, to perform this kind of optimization, there must be a mechanism for telling the compiler about the relevant abstractions and for facilitating program analysis in those terms.

Challenges. To summarize, while there are many opportunities for library-level optimization, there are also significant challenges. First, while library-level optimizations are conceptually similar to traditional optimizations, library routines are typically more complex than primitive language operators. Second, a library typically embodies a high-level domain of computation whose abstractions are not represented in the base language, and effective optimizations are often phrased in terms of these abstractions. Third, the compiler cannot be hardwired for every possible library because each library requires unique analyses and optimizations. Instead, all of these facilities need to be configurable. The next two sections address these issues in more detail.

3 Dependence Analysis

Almost any kind of program optimization requires a model of dataflow dependences to preserve the program's semantics. The use of pointers, and particularly pointer-based data structures, can greatly complicate dependence analysis, as pointers can make it difficult to determine which memory objects are actually modified. While many solutions to the pointer analysis problem have been proposed, we now argue that optimization at the library level requires the most precise and most aggressive.

3.1 Why Libraries Need Pointers

Libraries use pointers for two main reasons. The first is to overcome the limitations of the procedure call mechanism, as pointers allow a procedure to return more than one value. The second reason is to build and manipulate complex data structures that

represent the library's domain-specific programming abstractions. It is important to understand these structures because data dependences may exist between internal components of these data structures which, if violated, could change the program's behavior. As an example, consider the following PLAPACK routine:

```
PLA_Obj_horz_split_2(size, A, &A_upper, &A_lower);
```

This routine logically splits the matrix into two pieces by returning objects that represent ranges of the original matrix index space. Internally, the library defines a *view* data structure that consists of minimum and maximum values for the row and column indices, and a pointer to the actual matrix data. To see how this complicates analysis, consider the following code fragment:

```
PLA_Obj A, A_upper, A_lower, B;
PLA_Create_matrix(num_rows, num_cols, &A);
PLA_Obj_horz_split_2(size, A, &A_upper, &A_lower);
B = A_lower;
```

The first line declares four variables of type `PLA_Obj`, which is an opaque pointer to a *view* data structure. The second line creates a new matrix (both *view* and data) of the given size. The third line creates two *views* into the original data by splitting the rows into two groups, upper and lower. The fourth line performs a simple assignment of one *view* variable to another. Figure 3 shows the resulting data structures graphically.

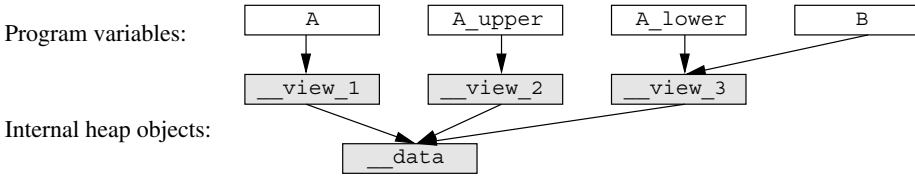


Fig. 3. Library data structures have complex internal structure.

The shaded objects are never visible to the application code, but accesses and modifications to them are still critical to preserving program semantics. For example, regardless of whether `A`, `A_lower`, or `B` is used, the compiler cannot change the order of library calls that update the data.

3.2 Pointer Analysis for Library-Level Optimizations

Pointer analysis attempts to determine whether there are multiple ways to access the same piece of memory. We categorize the many approaches to pointer analysis along the following dimensions: (1) points-to versus alias representation, (2) heap model, and (3) flow and context sensitivity. This section describes the characteristics of our compiler's pointer analysis and explains why they are appropriate for library-level optimization.

Representation. Heap allocation is common for library code because it allows data structures to exist throughout the life of the application code and it supports complex and dynamic structures. By necessity, components of these structures are connected by pointers, so a points-to relation provides a more natural model than alias pairs. Furthermore, in C the only true variable aliasing occurs between the fields of a union; all other aliases occur through pointers and pointer expressions.

Heap Model. The heap model determines the granularity and naming of heap allocated memory. Previous work demonstrates a range of possibilities including (1) one object to represent the whole heap [9], (2) one object for each connected data structure [12], (3) one object for each allocation call site [4], and (4) multiple objects for a malloc call site with a fixed limit ("k-limiting") [6]. In the matrix split example above, we need a model of the heap that distinguishes the *views* from the data. Thus, neither of the first two approaches is sufficiently precise. We choose approach (3) because it is precise, without the unnecessary complexity of the k-limiting approach.

Often, a library will provide only one or two functions that allocate new data structures. For example, the above `PLA_matrix_create` function creates all matrices in the system. Thus, if we associate only one object (or one data structure, e.g., one *view* and one data object) with the call site, we cannot distinguish between individual instances of the data structure. In the example, we would have one object to represent all *views* created by the split operation, preventing us from distinguishing between `__view_2` and `__view_3`. Therefore, we create a new object in the heap model for each unique execution path that leads to the allocation statement.

This naming system leads to an intuitive heap model where objects in the model often represent conceptual categories, such as "the memory allocated by foo()." Note that when allocation occurs in a loop, all of the objects created during execution of the loop are represented by one object in the heap model.

Context and Flow Sensitivity. Libraries are mechanisms for software reuse, so library calls often occur deep in the application's call graph, with the same library functions repeatedly invoked from different locations. Without context and flow sensitivity, the analyzer merges information from the different call sites. For example, consider a simple PLAPACK program that makes two calls to the split routine with different matrices:

```
PLA_Obj_horz_split_2(size, A, &A_upper, &A_lower);
PLA_Obj_horz_split_2(size, B, &B_upper, &B_lower);
```

Context insensitive analysis concludes that all four outputs might point to either `A`'s data or `B`'s data. While this information is conservatively correct, it severely limits optimization opportunities by creating an unnecessary data dependence. Any subsequent analyses that use this information suffers the same merging of information. For example, if the state of `A` is unknown, then analysis cannot safely infer that `B_upper` and `B_lower` contain all zeros, even if analysis concludes that `B` contains all zeros before the split.

The more reuse that occurs in a system, the more important it is to keep information separate. Thus, we implement full context sensitivity. While this approach is complex,

recent research shows that efficient implementations are possible [22]. Recent work also shows that more precise pointer analysis not only causes subsequent analyses to produce more precise results, but also causes them to run faster [18].

3.3 Annotations for Dependence Analysis

Our annotation language provides a mechanism for explicitly describing how a routine affects pointer structures and dataflow dependences. This information is integrated into the Broadway compiler's dataflow and pointer analysis framework. The compiler builds a uniform representation of dependences, regardless of whether they involve built-in operators or library routines. When annotations are available, our compiler reads that information directly. For the application and for libraries that have not been annotated, our compiler analyzes the source code using the pointer analysis described above.

```
procedure PLA_Obj_horz_split_2( obj, height, upper, lower)
{
  on_entry { obj    --> __view_1, DATA  of __view_1 -->__data }
  access { __view_1, height }
  modify { }
  on_exit  { upper  --> new __view_2, DATA  of  __view_2 --> __data,
             lower  --> new __view_3, DATA  of  __view_3 --> __data }
}
```

Fig. 4. Annotations for pointer structure and dataflow dependences.

Figure 4 shows the annotations for the PLAPACK matrix split routine. In some cases, a compiler could derive such information from the library source code. However, there are situations when this is impossible or undesirable. Many libraries encapsulate functionality for which no source code is available, such as low-level I/O or interprocess communication. Moreover, the annotations allow us to model abstract relationships that are not explicitly represented through pointers. For example, a file descriptor logically refers to a file on disk through a series of operating system data structures. Our annotations can explicitly represent the file and its relationship to the descriptor, which might make it possible to recognize when two descriptors access the same file.

Pointer Annotations: `on_entry` *and* `on_exit`. To convey the effects of the procedure on pointer-based data structures, the `on_entry` and `on_exit` annotations describe the pointer configuration before and after execution. Each annotation contains a list of expressions of the following form:

[*label* `of`] *identifier* `-->` [`new`] *identifier*

The `-->` operator, with an optional label, indicates that the object named by the identifier on the left logically points to the object named on the right. In the `on_entry` annotations, these expressions describe the state of the incoming arguments and give names to the internal objects. In the `on_exit` annotations, the expressions can create new objects (using the `new` keyword) and alter the relationships among existing objects.

In Figure 4, the `on_entry` annotation indicates that the formal parameter `obj` is a pointer and assigns the name `__view_1` to the target of the pointer. The annotation also says that `__view_1` is a pointer that points to `__data`. The `on_exit` annotation declares that the split procedure creates two new objects, `__view_2` and `__view_3`. The resulting pointer relationships correspond to those in Figure 3.

Dataflow Annotations: `access` *and* `modify`. The `access` and `modify` annotations declare the objects that the procedure accesses or modifies. These annotations may refer to formal parameters or to any of the internal objects introduced by the pointer annotations. The annotations in Figure 4 show that the procedure uses the `length` argument and reads the input *view* `__view_1`. In addition, we automatically add the accesses and modifications implied by the pointer annotations: a dereference of a pointer is an access, and setting a new target is a modification.

3.4 Implications

As described in Section 2, most library-level optimizations require more than just dependence information. However, simply having the proper dependence analysis information for library routines does enable some classical optimizations. For example, by separating accesses from modifications, we can identify and remove library calls that are dead code. To perform loop invariant code motion or common subexpression elimination, the compiler can identify purely functional routines by checking whether the objects accessed are different from those that are modified.

4 Library-Level Optimizations

Our system performs library-level optimizations by combining two complementary tools: dataflow analysis and pattern-based transformations. Patterns can concisely specify local syntactic properties and their transformations, but they cannot easily express properties beyond a basic block. Dataflow analysis concisely describes global context, but cannot easily describe complex patterns. Both tools have a wide range of realizations, from simple to complex, and by combining them, each tool is simplified. Both tools are configurable, allowing each library to have its own analyses and transformations.

The patterns need not capture complex context-dependent information because such information is better expressed by the program analysis framework. Consider the following fragment from an application program:

```
PLA_Obj_horz_split_2( A, size, &A_upper, &A_lower);
...
if ( is_Local(A_upper) ) { ... } else { ... }
```

The use of `PLA_Obj_horz_split_2` ensures that `A_upper` resides locally on a single processor. Therefore, the condition is always true, and we can simplify the subsequent `if` statement by replacing it with the then-branch. The transformation depends on two conditions that are not captured in the pattern. First, we need to know that the `is_Local`

function does not have any side-effects. Second, we need to track the library-specific *local* property of `A_upper` through any intervening statements to make sure that it is not invalidated. It would be awkward to use patterns to express these conditions.

Our program analysis framework fills in the missing capabilities, giving the patterns access to globally derived information such as data dependences and control-flow information. We use *abstract interpretation* to further extend these capabilities, allowing each library to specify its own dataflow analysis problems, such as the matrix distribution analysis needed above. This approach also keeps the annotation language itself simple, making it easier to express optimizations and easier to get them right. The following sequence outlines our process for library-level optimization:

1. **Pointers and dependences.** Analyze the code using combined pointer and dependence analysis, referring to annotations when available to describe library behavior.
2. **Classical optimizations.** Apply traditional optimizations that rely on dependence information only.
3. **Abstract interpretation.** Use the dataflow analysis framework to derive library-specific properties as specified by the annotations.
4. **Patterns**. Search the program for possible optimization opportunities, which are specified in the annotations as syntactic patterns.
5. **Enabling conditions.** For patterns that match, the annotations can specify additional constraints, which are expressed in terms of data dependences and the results of the abstract interpretation.
6. **Actions.** When a pattern satisfies all the constraints, the specified code transformation is performed.

4.1 Configurable Dataflow Analysis

This section describes our configurable interprocedural dataflow analysis framework. The annotations can be used to specify a new analysis pass by defining both the library-specific flow value and the associated transfer functions. For every new analysis pass, each library routine is supplied a transfer function that represents its behavior.

The compiler reads the specification and runs the given problem on our general-purpose dataflow framework. The framework takes care of propagating the flow values through the flow graph, applying the transfer functions, handling control-flow such as conditionals and loops, and testing for convergence. Once complete, it stores the final, stable flow values for each program point.

While other configurable program analyzers exist, ours is tailored specifically for library-level optimization. First, we would like library experts, not compiler experts, to be able to define their own analyses. Therefore, the specification of new analysis problems is designed to be simple and intuitive. Second, we do not intend to support every possible analysis problem. The annotation language provides a small set of flow value types and operators, which can be combined to solve many useful problems. The lattices implied by these types have predefined meet functions, allowing us to hide the underlying lattice theory from the annotator.

For library-level optimization, the most useful analyses seem to fall into three rough categories: (1) analyze the objects used by the program and classify them into library-specific categories, (2) track relationships among those objects, and (3) represent the

overall state of the computation. The PLAPACK distribution analysis is an instance of the first kind. To support these different kinds of analysis, we propose a simple type system for defining flow values.

Flow Value Types. Our flow value type system consists of primitive types and simple container types. A flow value is formed by combining a container type with one or two of the primitive types. Each flow value type includes a predefined meet function, which defines how instances of the type are combined at control-flow merge points. The other operations are used to define the transfer functions for each library routine.

number The *number* type supports basic arithmetic computations and comparisons. The meet function is a simple comparison: if the two numbers are not equal, then the result is lattice bottom.

object The *object* type represents any memory location, including global and stack variables, and heap allocated objects. The only operation supported tests whether two expressions refer to the same object.

statement The *statement* type refers to points in the program. We can use this type to record where computations take place. Operations include tests for equality, dominance and dependences.

category *Categories* are user-defined enumerated types that support hierarchies. For example, we can define a `Vehicle` categorization like this:

```
{ Land { Car, Truck { Pickup, Semi}}, Wa-
ter { Boat, Submarine}}
```

The meet function chooses the most specific category that includes the two given values. For example, the meet of `Pickup` and `Semi` yields `Truck`, while the meet of `Pickup` and `Submarine` yields lattice bottom. The resulting lattice forms a simple tree structure, as shown in Figure 5.

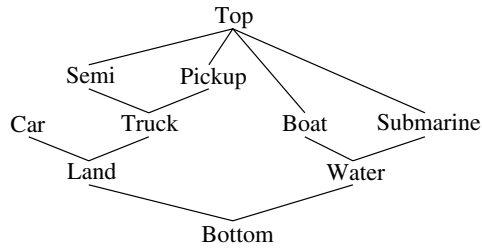


Fig. 5. The lattice induced by the example vehicle categories.

Operations on the categories include testing for equality and for category membership. For example, we may want to know whether something is a `Truck` without caring about its specific subtype.

set-of<T> Set is the simplest container: it holds any number of instances of the primitive type T. We can add and remove elements from the set, and test for membership. We support two possible meet functions for sets: set union for "optimistic" analyses, and set intersection for "pessimistic" analyses. It is up to the annotator to decide which is more appropriate for the specific analysis problem.

equivalence-of<T> This container maintains an equivalence relation over the elements that it contains. Operations on this container include adding pairs of elements to indicate that they are equivalent, and removing individual elements. The basic query operator tests whether two elements are equivalent by applying the transitive closure over the pairs that have been added. Like the set container, there are two possible meet functions, one optimistic and one pessimistic, that correspond to the union and intersection of the two relations.

ordering-of<T> The ordering container maintains a partial order over the elements it contains. We add elements in pairs, smaller element and larger element, to indicate ordering constraints. We can also remove elements. Like the equivalence container, the ordering container allows ordering queries on the elements it contains. In addition, it ensures that the relation remains antisymmetric by removing cycles completely.

map-of<K,V> The map container maintains a mapping between elements of the two given types. The first type is the "key" and may only have one instance of a particular value in the map at a time. It is associated with a "value."

We can model many interesting analysis problems using these simple types. The annotations define an analysis using the `property` keyword, followed by a name and then a flow value type expression. Figure 6 shows some example property definitions. The first one describes the flow value for the PLAPACK matrix distribution analysis. The equivalence `Aligned` could be used to determine when matrices are suitably aligned on the processors. The partial order `SubmatrixOf` could maintain the relative sizes of matrix views. The last example could be used for MPI to keep track of the asynchronous messages that are potentially "in flight" at any given point in the program.

```
property Distribution :
             map-of < object , { General { RowPanel, ColPanel, Local },
                                 Vector,
                                 Empty } >
property Aligned : pessimistic  equivalence-of < object >
property SubMatrixOf :  ordering-of < object >
property MessagesInFlight : optimistic  set-of < object >
```

Fig. 6. Examples of the `property` annotation for defining flow values.

Transfer Functions. For each analysis problem, transfer functions summarize the effects of library routines on the flow values. Transfer functions are specified as a case analysis, where each case consists of a condition, which tests the incoming flow values,

and a consequence, which sets the outgoing flow value. Both the conditions and the consequences are written in terms of the functions available on the flow value type.

```
procedure PLA_Obj_horz_split_2( obj, height, upper, lower)
{
  on_entry { obj    --> __view_1, DATA  of __view_1 -->__data }
  access { __view_1, height }
  modify { }
  analyze Distribution {
   (__view_1 == General)  => __view_2 = RowPanel, __view_3 = General;
   (__view_1 == RowPanel) => __view_2 = RowPanel, __view_3 = RowPanel;
   (__view_1 == ColPanel) => __view_2 = Local, __view_3 = ColPanel;
   (__view_1 == Local)    => __view_2 = Local, __view_3 = Empty;
  }
  on_exit  { upper  --> new __view_2, DATA  of  __view_2 --> __data,
             lower  --> new __view_3, DATA  of  __view_3 --> __data }
}
```

Fig. 7. Annotations for matrix distribution analysis.

Figure 7 shows the annotations for the PLAPACK routine `PLA_Obj_horz_split_2`, including those that define the matrix distribution transfer function. The `analyze` keyword indicates the property to which the transfer function applies. We integrate the transfer function with the dependence annotations because we need to refer to the underlying structures. Distribution is a property of the *views* (see Section 2), not the surface variables. Notice the last case: if we deduce that a particular *view* is `Empty`, we can remove any code that computes on that *view*.

4.2 Pattern-Based Transformations

The library-level optimizations themselves are best expressed using pattern-based transformations. Once the dataflow analyzer has collected whole-program information, many optimizations consist of identifying and modifying localized code fragments. Patterns provide an intuitive and configurable way to describe these code fragments. In PLAPACK, for example, we use the results of the matrix distribution analysis to replace individual library calls with specialized versions where possible.

Pattern-based transformations need to identify sequences of library calls, to check the call site against the dataflow analysis results, and to make modifications to the code. Thus, the annotations for pattern-based transformations consist of three parts: a *pattern*, which describes the target code fragment, *preconditions*, which must be satisfied, and an *action*, which specifies modifications to the code. The pattern is simply a code fragment that acts as a template, with special meta-variables that behave as "wildcards." The preconditions perform additional tests on the matching application code, such as checking data dependences and control-flow context, and looking up dataflow analysis results. The actions can specify several different kinds of code transformations, including moving, removing, or substituting the matching code.

Patterns. The pattern consists of a C code fragment with meta-variables that bind to different components in the matching application code. Our design is influenced by the issues raised in Section 3. Typical code pattern matchers work with expressions and rely on the tree structure of expressions to identify computations. However, the use of pointers and pointer-based data structures in the library interface presents a number of complications, and forces us to take a different approach.

The parameter passing conventions used by libraries have several consequences for pattern matching. First, the absence of a functional interface means that a pattern cannot be represented as an expression tree; instead, patterns consist of a sequence of statements with data dependences among them. Second, the use of the address operator to emulate pass-by-reference semantics obscures those data dependences. Finally, the pattern instance in the application code may contain intervening, but computationally irrelevant statements. Figure 8 depicts some of the possible complications by showing what would happen if the standard math library did not have a functional interface.

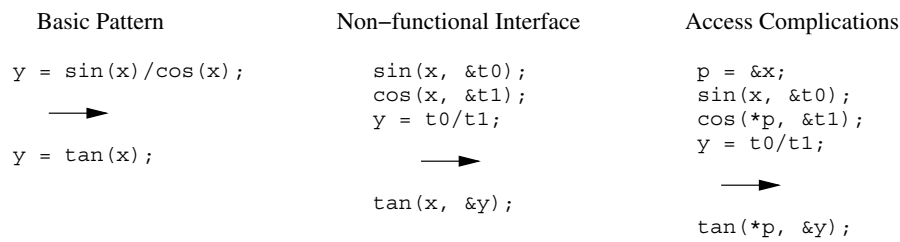


Fig. 8. The use of pointers for parameter passing complicates pattern matching.

To address these problems, we offer two meta-variable types, one that matches objects (both variables and heap-allocated memory), and one that matches constants. The object meta-variable ignores the different ways that objects are accessed. For example, in the third code fragment in Figure 8, the same meta variable would match both `x` and `*p`. The constant meta-variable can match a literal constant in the code, a constant expression, or the value of a variable if its value can be determined at compile time.

For a pattern to match, the application code must contain the specified sequence of statements, respecting any data dependences implied by the meta-variable names. The matching sequence may contain intervening statements, as long as those statements have no dependences with that sequence. We would like to weaken this restriction in the future, but doing so raises some difficult issues for pattern substitution.

Preconditions. The preconditions provide a way to test the results of the pointer analysis and user-defined dataflow analyses, since these can't be conveniently represented in the syntactic patterns. These dataflow requirements can be complicated for libraries, because important properties and dependences often exist between internal components of the data structures, rather than between the surface variables. For example, as shown in Figure 3, two different PLAPACK views may refer to the same underlying matrix data. An optimization may require that a sequence of PLAPACK calls all update the

same matrix. In this case the annotations need a way to access the pointer analysis information and make sure that the condition is satisfied. To do this, we allow the preconditions to refer to the `on_entry` and `on_exit` annotations for the library routines in the pattern. To access the dataflow analysis results, the preconditions can express queries using the same flow value operators that the transfer functions use. For example, the preconditions can express constraints such as, "the view of matrix A is empty."

Actions. When a pattern matches and the preconditions are satisfied, the compiler can perform the specified optimization. We have found that the most common optimizations for libraries consist of replacing a library call or sequence of library calls with more specialized code. The replacement code is specified as a code template, possibly containing meta variables, much like the patterns. Here, the compiler expands the embedded meta-variables, replacing them with the actual code bound to them. We also support queries on the meta-variables, such as the C datatype of the binding. This allows us to declare new variables that have the same type as existing variables.

In addition to pattern replacement, we offer four other actions: (1) remove the matching code, (2) move the code elsewhere in the application, (3) insert new code, or (4) trigger one of the enabling transformations such as inlining or loop unrolling.

When moving or inserting new code, the annotations support a variety of useful *positional indicators* that describe where to make the changes relative to the site of the matching code. For example, the earliest possible point and the latest possible point are defined by the dependences between the matching code and its surrounding context. Using these indicators, we can perform the MPI scheduling described in Section 2: move the `MPI_Isend` to the earliest point and the `MPI_Wait` to the latest point. Other positional indicators might include enclosing loop headers or footers, and the locations of reaching definitions or uses. Figure 9 demonstrates some of the annotations that use pattern-based transformations to optimize the examples presented in this paper.

5 Related Work

Our research extends to libraries previous work in optimization [17], partial evaluation [3, 7], abstract interpretation [8, 15], and pattern matching. This section relates our work to other efforts that provide configurable compilation technology.

The Genesis optimizer generator produces a compiler optimization pass from a declarative specification of the optimization [21]. Like Broadway, the specification uses patterns, conditions and actions. However, Genesis targets classical loop optimizations for parallelization, so it provides no way to define new program analyses. Conversely, the PAG system is a completely configurable program analyzer [16] that uses an ML-like language to specify the flow value lattices and transfer functions. While powerful, the specification is low-level and requires an intimate knowledge of the underlying mathematics. It does not include support for actual optimizations.

Some compilers provide special support for specific libraries. For example, *semantic expansion* has been used to optimize complex number and array libraries, essentially extending the language to include these libraries [1]. Similarly, some C compilers recognize calls to `malloc()` when performing pointer analysis. Our goal is to provide configurable compiler support that can apply to many libraries, not just a favored few.

```
pattern { ${obj:y} = sin(${obj:x})/cos(${obj:x})   }
{
  replace { $y = tan($x) }
}

pattern {
    MPI_Isend( ${obj:buffer}, ${expr:dest}, ${obj:req_ptr})
  }
{
  move @earliest;
}

pattern {
    PLA_Obj_horz_split_2( ${obj:A}, ${expr:size},
                          ${obj:upper_ptr}, ${obj:lower_ptr})
  }
{
  on_entry { A --> __view_1, DATA of __view_1 --> __data }
  when (Distribution of __view_1 == Empty) remove;
  when (Distribution of __view_1 == Local)
    replace {
      PLA_obj_view_all($A, $upper_ptr);
    }
}
```

Fig. 9. Example annotations for pattern-based transformations.

Meta-programming systems such as meta-object protocols [5], programmable syntax macros [20], and the Magik compiler [10], can be used to create customized library implementations, as well as to extend language semantics and syntax. While these techniques can be quite powerful, they require users to manipulate AST's and other compiler internals directly and with little dataflow information.

6 Conclusions

This paper has outlined the various challenges and possibilities for performing library-level optimizations. In particular, we have argued that such optimizations require precise pointer analysis, domain-specific information, and a configurable compilation scheme. We have also presented an annotation language that supports such a compilation scheme.

A large portion of our Broadway compiler has been implemented, including a flow- and context-sensitive pointer analysis, a configurable abstract interpretation pass, and the basic annotation language [13] without pattern matching. Experiments with this basic configuration have shown that significant performance improvements are possible for applications that use the PLAPACK library. One common routine, `PLA_Trsm()`, was customized to improve its performance by a factor of three, yielding speedups of 26% for a Cholesky factorization application and 9.5% for a Lyapunov program [13].

While we believe there is much promise for library-level optimizations, several open issues remain. We are in the process of defining the details of our annotation

language extensions for pattern matching, and we are implementing its associated pattern matcher. Finally, we need to evaluate the limits of our scheme—and of our use of abstract interpretation and pattern matching in particular—with respect to both optimization capabilities and ease of use.

References

[1] P.V. Artigas, M. Gupta, S.P. Midkiff, and J.E. Moreira. High performance numerical computing in Java: language and compiler issues. In *Workshop on Languages and Compilers for Parallel Computing*, 1999.

[2] G. Baker, J. Gunnels, G. Morrow, B. Riviere, and R. van de Geijn. PLAPACK: high performance through high level abstractions. In *Proceedings of the International Conference on Parallel Processing*, 1998.

[3] A. Berlin and D. Weise. Compiling scientific programs using partial evaluation. *IEEE Computer*, 23(12):23–37, December 1990.

[4] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. *ACM SIGPLAN Notices*, 25(6):296–310, June 1990.

[5] S. Chiba. A metaobject protocol for C++. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, pages 285–299, October 1995.

[6] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *ACM Symposium on Principles of Programming Languages*, pages 232–245, 1993.

[7] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *ACM Symposium on Principles of Programming Languages*, pages 493–501, 1993.

[8] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

[9] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *ACM Conference on Programming Language Design and Implementation*, pages 242–256, June 20–24, 1994.

[10] Dawson R. Engler. Incorporating application semantics and control into compilation. In *Proceedings of the Conference on Domain-Specific Languages (DSL-97)*, pages 103–118, Berkeley, October15–17 1997. USENIX Association.

[11] Message Passing Interface Forum. MPI: A message passing interface standard. *International Journal of Supercomputing Applications*, 8(3/4), 1994.

[12] Rakesh Ghiya and Laurie J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. *International Journal of Parallel Programming*, 24(6):547–578, December 1996.

[13] Samuel Z. Guyer and Calvin Lin. An annotation language for optimizing software libraries. In *Second Conference on Domain Specific Languages*, pages 39–52, October 1999.

[14] Samuel Z. Guyer and Calvin Lin. Broadway: A software architecture for scientific computing. In *IFIPS Working Group 2.5: Software Architectures for Scientific Computing Applications*, (to appear) October 2000.

[15] Neil D. Jones and Flemming Nielson. Abstract interpretation: a semantics-based tool for program analysis. In *Handbook of Logic in Computer Science*. Oxford University Press, 1994. 527–629.

[16] Florian Martin. PAG – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.

[17] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kauffman, San Francico, CA, 1997.

[18] M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In *4th International Static Analysis Symposium, Lecture Notes in Computer Science, Vol. 1302*, 1997.

[19] Robert van de Geijn. *Using PLAPACK – Parallel Linear Algebra Package*. The MIT Press, 1997.

[20] Daniel Weise and Roger Crew. Programmable syntax macros. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 156–165, June 1993.

[21] Deborah Whitfield and Mary Lou Soffa. Automatic generation of global optimizers. *ACM SIGPLAN Notices*, 26(6):120–129, June 1991.

[22] Robert P. Wilson. *Efficient, Context-sensitive Pointer Analysis for C Programs*. PhD thesis, Stanford University, Department of Electrical Engineering, 1997.

Compiler Techniques for Flat Neighborhood Networks

H.G. Dietz and T.I. Mattox

College of Engineering, Electrical Engineering Department
University of Kentucky
Lexington, KY 40506-0046
http://aggregate.org/
hankd@engr.uky.edu

Abstract. A Flat Neighborhood Network (FNN) is a new interconnection network architecture that can provide very low latency and high bisection bandwidth at a minimal cost for large clusters. However, unlike more traditional designs, FNNs generally are not symmetric. Thus, although an FNN by definition offers a certain base level of performance for random communication patterns, both the network design and communication (routing) schedules can be optimized to make specific communication patterns achieve significantly more than the basic performance. The primary mechanism for design of both the network and communication schedules is a set of genetic search algorithms (GAs) that derive good designs from specifications of particular communication patterns. This paper centers on the use of these GAs to compile the network wiring pattern, basic routing tables, and code for specific communication patterns that will use an optimized schedule rather than simply applying the basic routing.

1 Introduction

In order to use compiler techniques to design and schedule use of FNNs, it is first necessary to understand precisely what a FNN is and why such an architecture is beneficial. Toward that, it is useful to briefly discuss how the concept of a FNN arose. Throughout this paper, we will use KLAT2 (Kentucky Linux Athlon Testbed 2), the first FNN cluster, as an example. Though not huge, KLAT2 is large enough to effectively demonstrate the utility of FNNs: it unites 66 Athlon PCs using a FNN consisting of 264 NICs (Network Interface Cards) and 10 switches.

There are two reasons that the processors of a parallel computer need to be connected: (1) to send data between them and (2) to agree on global properties of the computation. As we discussed in [1], the second functionality is not well-served using message-passing hardware. Here, we focus on the first concern. Further, we will restrict our discussion to clusters of PCs, since few people will have the opportunity to design their own traditional supercomputer's network.

S.P. Midkiff et al. (Eds.): LCPC 2000, LNCS 2017, pp. 244–258, 2001.

Fig. 1a. Direct connections

Fig. 1b. Switchless connections

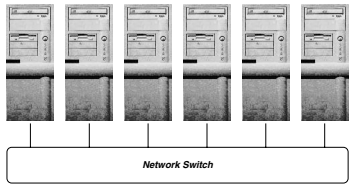


Fig. 1c. Ideal switch

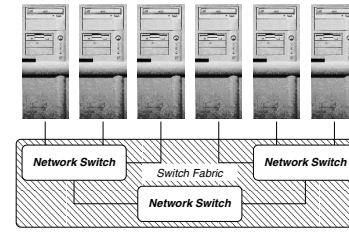


Fig. 1d. Switch fabric

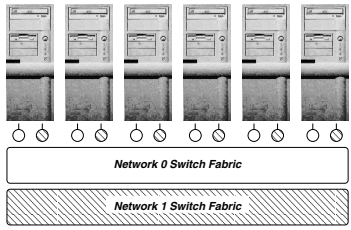


Fig. 1e. Channel bonding

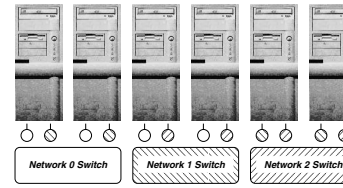


Fig. 1f. FNN

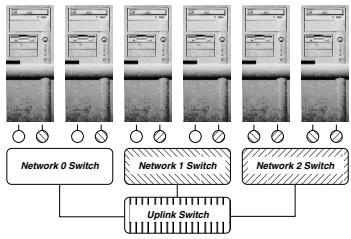


Fig. 1g. FNN with uplink switch

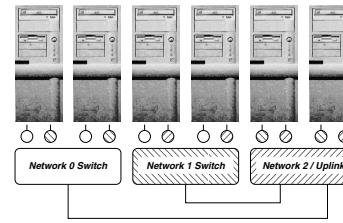


Fig. 1h. FNN with folded uplink

Fig. 1. Network topologies used in connecting cluster nodes

In the broadest terms, we need to distinguish only six different classes of network topologies (and two minor variations on the last). These are shown in Fig. 1.

The ideal network configuration would be one in which each processor is directly connected to every other node, as shown in Fig. 1a. Unfortunately,

for an *N*-node system this would require *N*-1 connections for each node. Using standard motherboards and NICs, there are only bus slots for a maximum of 4-6 NICs. Using relatively expensive 4-interface cards, the upper bound could be as high as 24 connections; but even that would not be usable for a cluster with more than 25 nodes.

Accepting a limit on the number of connections per node, direct connections between all nodes are not possible. However, it is possible to use each node as a switch in the network, routing through nodes for some communications. In general, the interconnection pattern used is equivalent to some type of hypermesh; in Fig. 1b, a degree 2 version (a ring) is pictured. Because NICs are generally cheaper than switches, this structure minimizes cost, but it also yields very large routing delays – very high latency.

To minimize latency without resorting to direct connections, the ideal network would connect a high-bandwidth NIC in each node to a single wire-speed switch, as shown in Fig. 1c. For example, using any of the various Gb/s network technologies (Gigabit Ethernet [2], Myricom's Myrinet [3], Giganet's CLAN [4], Dolphin's SCI [5]), it is now possible to build such a network. Unfortunately, the cost of a single Gb/s NIC exceeds the cost of a typical node, the switch is even more expensive per port, and wide switches are not available at all. Most Gb/s switches are reasonably cheap for 4 ports, expensive for 8 or 16 ports, and only a few are available with as many as 64 ports. Thus, this topology works only for small clusters.

The closest scalable approximation to the single switch solution substitutes a hierarchical switching fabric for the single switch, as shown in Fig. 1d. Some Gb/s technologies allow more flexibility than others in selecting the fabric's internal topology; for example, Gb/s Ethernet only supports a simple tree whereas Giganet CLAN can use higher-performance topologies such as fat trees – at the expense of additional switches. However, any switching fabric will have a higher latency than a single switch. Further, the bisection bandwidth of the entire system is limited to the lesser of the bisection bandwidth of the switch(es) at the top of the tree or the total bandwidth of the links to the top switch(es). This is problematic for Gb/s technologies because the QuplinksU that interconnect switches within the fabric are generally the same speed as the connections used for the NICs; thus, half of the ports on each switch must be used for uplinks to achieve the maximum bisection bandwidth.

Fortunately, 100Mb/s Ethernet switches do not share this last problem: wire-speed 100Mb/s switches often have Gb/s uplinks. Thus, it is possible to build significantly wider switch fabrics that preserve bisection bandwidth at a relatively low cost. The problem is that a single 100Mb/s NIC per node does not provide enough bandwidth for many applications. Fig. 1e shows the standard Linux-supported solution: use multiple NICs per node, connect them to identical fabrics, and treat the set of NICs in each node as a single, parallel, NIC. The software support for this, commonly known as Qchannel bonding,U was the primary technical contribution of the original Beowulf project. Unfortunately, the switch fabric latency is still high and building very large clusters this way yields

the same bisection bandwidth and cost problems discussed for Gb/s systems built as shown in Fig. 1d. Further, because channel-bonded NICs are treated as a single wide channel, the ability to send to different places by simultaneously using different NICs is not fully utilized.

The Flat Neighborhood Network (FNN), shown in Fig. 1f, solves all these problems. Because switches are connected only to NICs, not to other switches, single switch latency and relatively high bisection bandwidths are achieved. Cost also is significantly lower. However, FNNs do cause two problems. The first is that some pairs of nodes only have single-NIC bandwidth between them with the minimum latency, although extra bandwidth can be obtained with a higher latency by routing through nodes to QhopU neighborhoods. The second problem is that routing becomes a very complex issue.

For example, the first two machines in Fig. 1f have two neighborhoods (subnets) in common, so communication between them can be done much as it would be for channel bonding. However, that bonding of the first machine's NICs would not work when sending a message to the third machine because those nodes share only one neighborhood. Even without the equivalent of channel bonding, routing is complicated by the fact that the apparent address (NIC) for the third machine is different depending on which node is sending to it; the first machine would talk to the first NIC in the third node, but the last machine would talk to the second NIC. Further, although this very small example has some symmetry which could be used to simplify the specification of routing rules, that is not generally true of FNNs.

At this point, it is useful to abstract the fully general definition of a FNN: a network using a topology in which all important (usually, but not necessarily, all) point-to-point communication paths are implemented with only a single switch latency. In practice, it is convenient to augment the FNN with an additional switch that connects to the uplinks from the FNN's switches, since that switch can provide more efficient multicast support and I/O with external systems (e.g., workstations or other clusters). This second-level switch also can be a convenient location for Qhot spareU nodes to be connected. The FNN with this additional uplink switch is shown in Fig. 1g.

In the special case that one of the FNN switches has sufficient ports available, it also is possible to QfoldU the uplink switch into one of the FNN switches. This folded uplink FNN configuration is shown in Fig. 1h. Although the example's 4-port switches would not be wide enough to be connected as shown in this figure, if the switches are wide enough, it always is possible to design the network so that sufficient ports are reserved on one of the FNN switches.

Thus, FNNs scale well, easily provide multicast and external I/O, and offer high performance at low cost. A more detailed evaluation of the performance of FNNs (especially in comparison to fat trees and channel bonding) is given in [9]. Independent of and concurrent with our work, a group at the Australian National University created a cluster (*Bunyip* [10]) whose network happens to have the FNN properties, and their work confirms the performance benefits.

The problem with FNNs is that they require clever routing. Further, their performance can be improved by tuning the placement of the paths with extra bandwidth so that they correspond to the communication patterns that are most important for typical applications. In other words, FNNs require compiler technology for analysis and scheduling (routing) of communication patterns in order to achieve their full performance. Making full use of this technology for both the design and use of FNNs yields significant benefits.

Bunyip [10] uses a hand-designed symmetric FNN. However, the complexity of the FNN design problem explodes when a system is being designed with a set of optimization criteria. Optimization criteria range from information about relative importance of various communication patterns to node physical placement cost functions (intended to reduce physical wiring complexity). Further, many of these criteria interact in ponderous ways that only can be evaluated by partial simulation of potential designs. It is for these reasons that our FNN design tools are based on genetic search algorithms (GAs).

2 The FNN Compiler

The first step in creating a FNN system is the design of the physical network. Logically, the design of the network is a function of two separate sets of constraints: the constraints imposed by physical hardware and those derived from analysis of the communications that the resulting FNN is to perform. Thus, the compiler's task is to parse specifications of these constraints, construct and execute a GA that can optimize the design according to these constraints, and finally to encode the resulting design in a form that facilitates its physical construction and use.

The current version of our network compiler uses:

- A specification of how many PCs, the maximum number of NICs per PC (all PCs do not have to have the same number of NICs!), and a list of available switches specified by their width (number of ports available per switch). Additional dummy NICs and/or switches are automatically created within the program to allow uneven use of real NICs/switch ports. For example, KLAT2's current network uses only 8 of 31 ports on one of its switches; the other switch ports appear to be occupied by dummy NICs that were created by the program.
- A designer-supplied evaluation function that returns a quality value derived by analysis of specific communication patterns and other performance measures. This function also marks problem spots in the proposed network configuration so that they can be preferentially changed in the GA process.

In the near future, we expect to distribute a version of the compiler which has been enhanced to additionally include:

- A list of switch and NIC hardware costs, so that the selection of switches and NIC counts also can be automatically optimized.
- A clean language interface for this specification.

Currently, we modify the GA-based compiler itself by including C functions that redefine the search parameters.

2.1 The GA Structure

The GA is not a generic GA, but is highly specialized to the problem of designing the network. The primary data structure is a table of bitmasks for each PC; each PC's bitmask has a 1 only in positions corresponding to each neighborhood (switch) to which that PC has a NIC connected. This data structure does not allow a PC to have multiple NICs connected to the same neighborhood, however, such a configuration would add nothing to the FNN connectivity. Enforcing this constraint and the maximum number of NICs greatly narrows the search space.

Written in C, the GA's bitmask data structure facilitates use of SIMD-within-a-register parallelism [6] when executed on a single processor. It also can be executed in parallel using a cluster. KLAT2's current network design was actually created using our first Athlon cluster, Odie – four 600MHz Athlon PCs.

To more quickly converge on a good solution, the GA is applied in two distinct phases. Large network design problems with complex evaluation functions are first converted into smaller problems to be solved for a simplified evaluation function. This rephrased problem often can be solved very quickly and then scaled up, yielding a set of initial configurations that will make the full search converge faster.

The simplified cost weighting only values basic FNN connectivity, making each PC directly reachable from every other. The problem is made smaller by dividing both the PC count and the switch port counts by the same number while keeping the NICs per PC unchanged. For example, a design problem using 24-port switches and 48 PCs is first scaled to 2-port switches and 4 PCs; if no solution is found within the alloted time, then 3-port switches and 6 PCs are tried, then 4-port switches and 8 PCs, etc. A number of generations after finding a solution to one of the simplified network design problems, the population of network designs is scaled back to the original problem size, and the GA resumes using the designer-specified evaluation function.

If no solution is found for any of the scaled-down problems, the GA is directly applied to the full-size problem.

2.2 The Genetic Algorithm Itself

The initial population for the GA is constructed for the scaled-down problem using a very straightforward process in which each PC's NICs are connected to the lowest-numbered switch that still has ports available and is not connected to the same PC via another NIC. Additional dummy switches are created if the process runs out of switch ports; similarly, dummy NICs are assigned to virtual PCs to absorb any unused real switch ports. The resulting scaled-down initial FNN design satisfies all the constraints except PC-to-PC connectivity. Because the full-size GA search typically begins with a population created from

a scaled-down population, it also satisfies all the basic design constraints except connectivity.

By making all the GA transformations preserve these properties, the evaluation process checks only connectivity, not switch port usage, NIC usage, etc.

The GA's generation loop begins by evaluating all new members of a population of potential FNN designs. Determining which switches are shared by two PCs is a simple matter of bitwise AND of the two bitmasks; counting the ones in that result measures the available bandwidth. Which higher-level evaluation function is used depends on whether the problem has been scaled-down. The complete population is then sorted in order of decreasing fitness, so that the top `KEEP` entries will be used to build the next generation's population. In order to ensure some genetic variety, the last `FUDGE` FNN designs that would be kept intact are randomly exchanged with others that would not have been kept. If a new FNN design is the best fit, it is reported.

Aside from the GA using different evaluation functions for the full size and scaled-down problems, there are also different stopping conditions applied at this point in the GA. Since we cannot know what the precise optimum design's value will be for full-size search, it terminates only when the maximum number of generations has elapsed. In contrast, the scaled-down search will terminate in fewer generations if a FNN design with the basic connectivity is found earlier in the search.

Crossover is then used to synthesize `CROSS` new FNN designs by combining aspects of pairs of parent FNN designs that were marked to be kept. The procedure used begins by randomly selecting two different parent FNN designs, one of which is copied as the starting design for the child. This child then has a random number of substitutions made, one at a time, by randomly picking a PC and making its set of NIC connections match those for that PC in the other parent. This forced match process works by exchanging NIC connections with other PCs (which may be real or dummy PCs) in the child that had the desired NIC connections. Thus, the resulting child has properties taken from both parents, yet always is a complete specification of the NIC to switch mapping. In other words, crossover is based on exchange of closed sets of connections, so the new configuration always satisfies the designer-specified constraints on the number of NICs/PC and the number of ports for each switch.

Mutation is used to create the remainder of the new population from the kept and crossover designs. Two different types of crossover operation are used, both applied a random number of times to create each mutated FNN design:

1. The first mutation technique swaps individual NIC-to-switch connections between PCs selected at random.
2. The second mutation technique simply swaps the connections of one PC with those of another PC, essentially exchanging PC numbers.

Thus, the mutation operators are also closed and preserve the basic NIC and switch port design constraints. The generation process is then repeated with a population consisting of the kept designs from the previous generation, crossover products, and mutated designs.

2.3 The FNN Compiler's Output

The output of the FNN compiler is simply a table. Each line begins with a switch number followed by a :, which is then followed by the list of PC numbers connected to that switch.

This list is given in sorted order, but for ideal switches, it makes no difference which PCs are connected to which ports, provided that the ports are on the correct switch. It also makes very little difference which NICs within a PC are connected to which switch. However, to construct routing tables, it is necessary to know which NICs are connected to each switch, so we find it convenient to also order the NICs such that, within each PC, the lowest-numbered NIC is connected to the lowest-numbered switch, etc.

We use this simple text table as the input to all our other tools. Thus, the table could be edited, or even created, by hand.

3 The FNN Translators

Once the FNN compiler has created the network design, there are a variety of forms that this design must be translated into in order to create a working implementation. For this purpose, we have created a series of translators.

3.1 Physical Wiring

One of the worst features of FNNs is that they are physically difficult to wire. This is because, by design, they are irregular and often have very poor physical locality between switches and NICs. Despite this, wiring KLAT2's PCs with 4 NICs each took less than a minute per cable, including the time to neatly route the cables between the PC and the switches.

The trick that allowed us to wire the system so quickly is nothing more than color-coding of the switches and NICs. As described above, all the ports on a switch can be considered interchangeable; it doesn't matter which switch port a NIC is plugged into. Category 5 cable, the standard for Fast Ethernet, is available in dozens of colors at no extra cost. Thus, the problem is simply how to label the PCs with the appropriate colors for the NICs it contains.

For this purpose, we created a simple program that translates the FNN switch connection representation into an HTML file. This file, which can be loaded into any WWW browser and printed, contains a set of per-PC color-coded labels that have a color patch for each NIC in the PC showing which color cable, and hence which switch, should be connected. KLAT2's wiring, and the labels that were used to guide the physical process, are shown in Fig. 2.

For KLAT2, it happens that half of our cables were transparent colors; the transparent colors are distinguished from the solid colors by use of a double triangle. Of course, a monochrome copy of this paper makes it difficult to identify specific colors, but the color-coding of the wires is obvious when the color-coded labels are placed next to the NICs on the back of each PC case, as you can see them in the photo in Fig. 2.

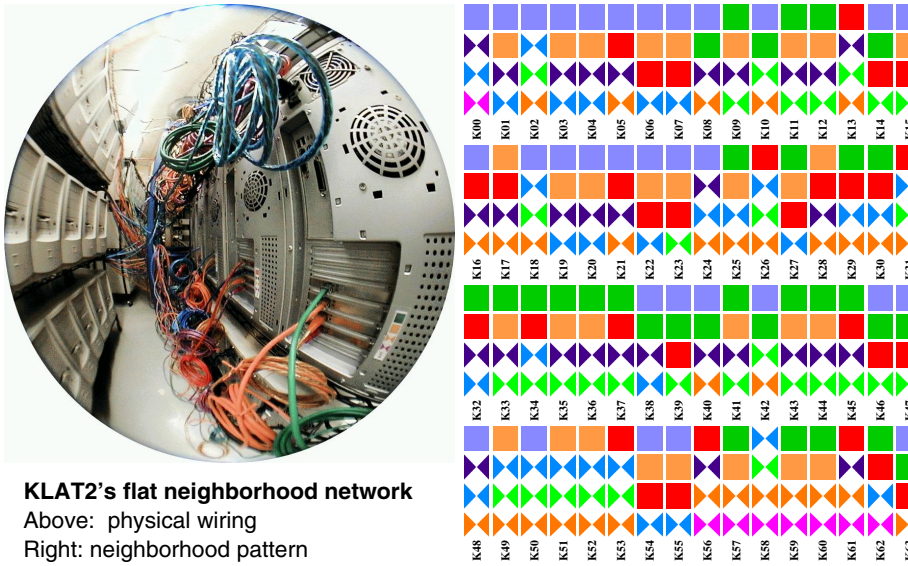


Fig. 2. FNN wiring of KLAT2's 64 PCs with 4 NICs each

3.2 Basic Routing Tables

In the days of the Connection Machines (CMs), Thinking Machines employees often could be heard repeating the mantra, "*all* the wires *all* the time." The same focus applies to FNN designs: there is tremendous bandwidth available, but only when all the NICs are kept busy. There are two ways to keep all the wires busy. One way is to have single messages divided into pieces sent by all the NICs within a PC, as is done using channel bonding. The other way is to have transmission of several messages to different destinations overlap, with one message per NIC. Because FNNs generally do not have sufficient connectivity to keep all the wires busy using the first approach, the basic FNN routing centers on efficient use of the second.

Although IP routing is normally an automatic procedure, the usual means by which it is automated do not work well using a FNN. Sending out broadcast requests to find addresses is an exceedingly bad way to use a FNN, especially if an uplink switch is used, because that will make all NICs appear to be connected rather than just the ones that share subnets. Worse still, some software systems, such as LAM MPI [7, 8], try to avoid the broadcasts by determining the locations of PCs once and then passing these addresses to all PCs. That approach fails because each PC actually has several addresses (one per NIC) and the proper one to use depends on which PC is communicating with it. For example, in Fig. 1f, the first PC would talk to the third PC via its address on subnet 1, but the last PC would talk to it via the address on subnet 3. Thus, we need to construct a unique routing table for each PC.

To construct these routing tables, we must essentially select one path between each pair of PCs. According to the user-specified communication patterns, some PC-to-PC paths are more important than others. Thus, the assignments are made in order of decreasing path importance. However, the number of alternative paths between PCs also varies, so among paths of equal importance, we assign the paths with the fewest alternatives first.

For the PC pairs that have only a single neighborhood in common, the selection of the path is trivial. Once that has been done, the translator then examines PC pairs with two neighborhoods in common, and tries to select the the path whose NICs have thus far appeared in the fewest assigned paths. The process then continues to assign paths for pairs with three, then four, etc., neighborhoods in common. The complete process is then repeated for the next most important pairs, and so forth, until every pair has been assigned a path.

KLAT2's current network was designed partially optimizing row and column communication in an 8x8 logical configuration of the 64 processors (the two hot spares are on the uplink switch). Although the translator software actually builds a shell script that, when executed, builds the complete set of host routing tables (actually, pre-loading of each ARP cache), that output is too large to include in this paper. A shorter version is simply a table that indicates which subnets are used for each pairwise communication, as shown in Fig. 3.

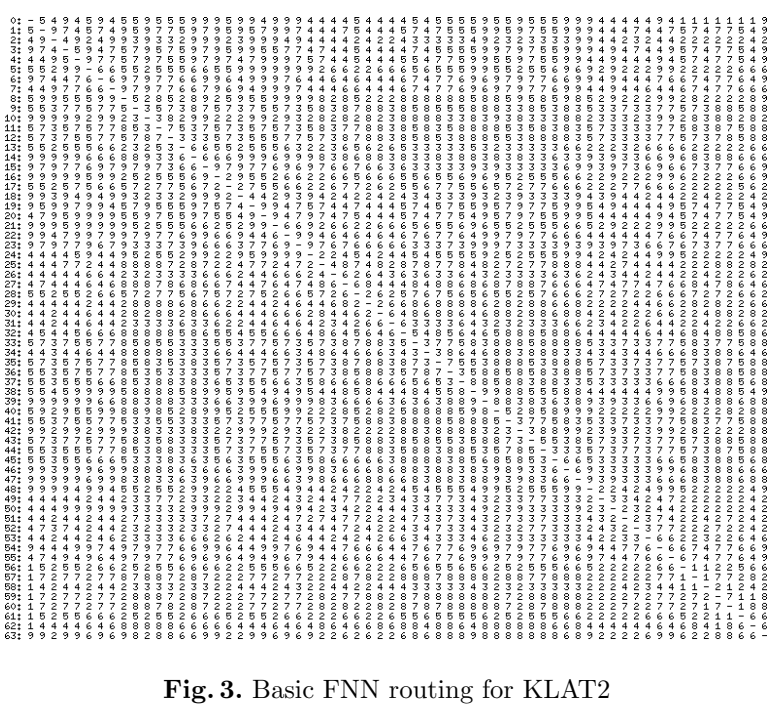


Fig. 3. Basic FNN routing for KLAT2

3.3 Advanced Routing Tables

As discussed above, in a typical FNN, many pairs of PCs will share multiple neighborhoods. Thus, additional bandwidth can be achieved for a single message communication by breaking the message into chunks that can be sent via different paths and sending the data over all available paths simultaneously – the FNN equivalent of channel bonding. What makes FNN advanced routing difficult is that, unlike conventional channel bonding, the FNN mechanism must be able to correctly manage the fact that NICs are bonded only for a specific message destination rather than for all messages.

For example, in Fig. 2, PC **k00** is connected to the blue, transparent purple, transparent blue, and transparent magenta neighborhoods. The second PC, **k01**, shares three of those neighborhoods, replacing the transparent magenta with orange. The third PC, **k02**, has only two neighborhoods in common with **k00**: blue and transparent blue. Thus, when **k00** sends to **k01**, three of its NICs can be used to create one wider data path, but when sending from **k00** to **k02**, only two NICs can be used together. If **k00** needs to send a message to **k63**, there is only one neighborhood in common and only one NIC can be used.

Although sending message chunks through different paths is not trivial, the good news is that the selection of paths can be done locally (within each PC) without loss of optimality for any permutation communication. By definition, any communication pattern that is a permutation has only one PC sending to any particular PC. Because there is no other sender targeting the same PC, and all paths are implemented directly through wire-speed switches, there is no possibility of encountering interference from another PC's communication. Further, nearly all Fast Ethernet NICs are able to send data at the same time that they are receiving data, so there is no interference within the NIC from other messages being sent out. Of course, there may be some memory access interference within the PCs, but that is relatively unimportant.

A simple translator can encode the FNN topology so that a runtime procedure can determine which NICs to specify as the sources and destinations. This is done by translating the switch neighborhood definitions into a table of NIC tuples. Each tuple specifies the NIC numbers in the destination PC that correspond to each of the NICs in the source PC. For example, the routing from **k00** to **k01** would be represented by a tuple of 1-3-4-0 meaning that **k00**'s first NIC is routed to **k01**'s first NIC, **k00**'s second NIC is routed to the third NIC of **K01**, the third NIC of **k00** is routed to the fourth NIC of **k01**, and the final value of 0 means that the fourth NIC of **k00** is not used.

To improve caching and simplify lookup, each of the NIC tuples is encoded as a single integer and a set of macros to extract the individual NIC numbers from that integer. Extraction of a field is a shift followed by a bitwise AND. With this encoding, the complete advanced routing table for a node in KLAT2 is just 128 bytes long.

Using standard Ethernet hardware, the routing by NIC numbers would require the ARP cache in each machine to translate these addresses into MAC hardware addresses. This is easily done for small clusters, but can become less

efficient if very large ARP caches are required. Thus, it may be more practical to lookup MAC addresses directly rather than NIC numbers. The result is a modest increase in table size. For KLAT2, the MAC-lookup table would require 1.5K bytes.

3.4 Problems and Troubleshooting

Unfortunately, the unusual properties of FNNs make it somewhat difficult to debug the system. Although one might expect wiring errors to be common, the color coding essentially eliminates this problem. Empirically, we have developed the following list of FNN problems and troubleshooting techniques:

- The numbering of NICs depends on the PCI bus probe sequence, which might not be in an obvious order as the PCI bus slots are physically positioned on the motherboard. For example, the slots in the FIC SD11 motherboards are probed in the physical order 1-2-4-3. Fortunately, the probe order is consistent for a particular motherboard, so it is simply a matter of determining this order using one machine before physically wiring the FNN.
- If the FNN has an uplink switch, any unintended broadcast traffic, especially ARPs, can cripple network performance. Looking at the Ethernet status lights, it is very easy to recognize broadcasts; unfortunately, a switch failure also can result in unwanted broadcast traffic. Using a network analyzer and selectively pulling uplinks makes it fairly easy to identify the source(s) of the broadcasts. Typically, if it is a software problem, it will be an external machine that sent an ARP into the cluster. This problem can be fixed by appropriately adjusting ARP caches or by firewalling – which we strongly recommend for clusters.
- Application-level software that assumes each machine has a single IP/MAC address independent of the originating PC will cause many routes to go through the FNN uplink switch, whereas normal cluster-internal communications do not use the uplink switch. All application code should use host name lookup (e.g., in the local ARP cache) on each node.

Given that the system is functioning correctly with respect to the above problems, physical wiring problems (typically, a bad cable or NIC) are trivially detected by failure of a ping.

4 Performance

Although the asymmetry of FNNs defies closed-form analysis, it is possible to make a few analytic statements about performance. Using KLAT2, we also have preliminary empirical evidence that the benefits predicted for FNNs actually are delivered.

4.1 Latency and Pairwise Bandwidth

Clearly, the minimum latency between any pair of PCs is just one switch delay and the minimum bandwidth available on any path is never less than that provided by one NIC (i.e., 100Mb/s unidirectional, 200Mb/s bidirectional for Fast Ethernet). The bandwidth available between a pair of PCs depends on the precise wiring pattern, however, it is possible to compute a tight upper bound on the average bandwidth as follows.

PCs communicate in pairs. Because no PC can have two NICs connected to the same switch, the number of ways in which a pair of connections through an *S*-port switch can be selected is *S*(S-1)/2*. Only switch ports that are connected to NICs count. Similarly, if there are *P* PCs, the number of pairs of PCs is *P*(P-1)/2*. If we sum the number of connections possible through all switches and divide that sum by the number of PC pairs, we have a tight upper bound on the average number of links between a PC pair. Since both the numerator and denominator of this fraction are divided by 2, the formula can be simplified by multiplying all terms by 2.

For example, KLAT2's network design described in this paper uses 4 NICs, 31 ports on each of 8 switches and 8 ports on the ninth, and has 64 PCs (the two "hot spare" PCs are placed on the uplink switch). Thus, we get about 1.859 bidirectional links/pair. In fact, the FNN design shown for KLAT2 achieves precisely this average pairwise bandwidth. Using 100Mb/s Ethernet, that translates to 371.8Mb/s bidirectional bandwidth per pair.

An interesting side effect of this formula is that, if some switch ports will be unused, the maximum average pairwise bandwidth will be achieved when all but one of the switches has all its ports used. Thus, the GA naturally tends to result in FNN designs that facilitate the folded uplink configuration.

4.2 Bisection Bandwidth

Bisection bandwidth is far more difficult to compute because the bisection is derived by dividing the machine in half in the worst way possible and measuring the maximum bandwidth between the halves. A reasonable upper bound on the bisection bandwidth is clearly the total number of NICs times the number of PCs times the unidirectional bandwidth per NIC; for KLAT2, this is 4*64*100, or 25.6Gb/s.

Generally, bisection bandwidth benchmarks measure performance using a permutation communication pattern, but which pairwise communications are used is not specified and it can make a large difference which PCs in each half are paired. If we select pairwise communications between the two halves using a random permutation, the expected bisection bandwidth can be computed using the average bandwidth available per PC, computed as described above. For KLAT2, this would yield 371.8Mb/s*32, or 11.9Gb/s.

Of course, the above computations ignore the additional bandwidth available by hopping subnets using either routing through PCs or the uplink switch. Although a folded uplink switch adds slightly more bisection bandwidth than an

unfolded one, it is easy to see that a non-folded uplink switch essentially adds bisection bandwidth equal to the number of non-uplink switches used times the bidirectional uplink bandwidth. For KLAT2's 9 switches with Fast Ethernet uplinks, an unfolded uplink switch adds 1.8Gb/s to the 11.9Gb/s total, yielding 13.7Gb/s. However, the routing techniques described in this paper normally ignore the communication paths that would route through the uplink switch.

Further complicating the measurement of FNN bisection bandwidth is the fact that peak bisection bandwidth for a FNN is not necessarily achievable for *any* permutation communication pattern. The ability of multiple NICs to function simultaneously within each PC makes it far easier to achieve high bisection bandwidth using a pattern in which many PCs *simultaneously* send messages to various destinations through several of their NICs. We know of no standard bisection bandwidth benchmark that would take advantage of this property, yet the performance increase is easily observed in running real codes.

4.3 Empirical Performance

The FNN concept is very new and we have not yet had time to fully evaluate its performance nor to clean-up and release the public-domain compiler and runtime software that we have been developing to support it. Thus, we have not yet run detailed network performance benchmarks. However, KLAT2's FNN has enabled it to achieve very high performance on several applications.

At this writing, a full CFD (Computational Fluid Dynamics) code [11], such as normally would be run on a shared-memory machine, is running on KLAT2 well enough that it is a finalist for a Gordon Bell Price/Performance award. KLAT2 also achieves over 64 GFLOPS on the standard LINPACK benchmark (using ScaLAPACK with our 32-bit floating-point *3DNow!* SWAR extensions).

Why is performance so good? The first reason is the bandwidth. As described above, KLAT2's FNN has about 25Gb/s bisection bandwidth – an ideal 100Mb/s switch the full width of the cluster would provide no more than 6.4Gb/s bisection bandwidth, and such a switch would cost far more than the FNN. Although Gb/s hardware can provide higher pairwise bandwidth, using a tree switch fabric yields less than 10Gb/s bisection bandwidth at an order of magnitude higher cost than KLAT2's FNN.

Additional FNN performance boosts come from the low latency that results from having only a single switch delay between source and destination PCs and from the semi-independent use of multiple NICs. Having four NICs in each PC allows for parallel overlap in communications that the normal Linux IP mechanisms would not provide with channel bonding or with a single NIC. Further, because each hardware interface is buffered, the FNN communications benefit from greater buffered overlap.

5 Conclusion

In this paper, we have introduced a variety of compiler-flavored techniques for the design and use of a new type of scalable network, the Flat Neighborhood Network (FNN).

The FNN topology and routing concepts make it exceptionally cheap to implement – the network hardware for KLAT2's 64 (plus 2 spare) nodes cost about 8,000 dollars. It is not only much cheaper than Gb/s alternatives that it outperforms, but also is cheaper than a conventional 100Mb/s implementation would have been using a single NIC per PC and a cluster-width switch.

The low cost and high performance are not an accident, but are features designed using a genetic search algorithm (GA) to create a network optimized for the specific communications that are expected to be important for the parallel programs the system will run. Additional compiler tools also were developed to manage the relatively exotic wiring complexity and routing issues. With these tools, it is easy and cost-effective to customize the system network design at a level never before possible.

KLAT2, the first FNN machine, is described in detail at:
`http://aggregate.org/KLAT2/`

References

[1] Dietz, H., Chung, T., Mattox, T.: A Parallel Processing Support Library Based On Synchronized Aggregate Communication. In: Languages and Compilers for Parallel Computing, Springer-Verlag, New York (1996) 254-268
[2] The Gigabit Ethernet Alliance, `http://www.gigabit-ethernet.org/`
[3] Myricom, Inc., `http://www.myri.com/`
[4] Giganet CLAN, `http://www.giganet.com/products/indexlinux.htm`
[5] Dolphin SCI (Scalable Coherent Interconnect), `http://www.dolphinics.com/`
[6] Fisher, R., Dietz, H.: The Scc Compiler: SWARing at MMX and 3DNow!. In: Carter, L., Ferrante, J. (eds.): Languages and Compilers for Parallel Computing, Springer-Verlag, New York (2000) 399-414
[7] The LAM MPI Implementation, `http://www.mpi.nd.edu/lam/`
[8] Message Passing Interface Forum: MPI: A Message-Passing Interface Standard. Rice University, Technical Report CRPC-TR94439, April 1994.
[9] Dietz, H., Mattox, T.: KLAT2's Flat Neighborhood Network. In: proceedings of the Fourth Annual Linux Showcase (ALS2000, Extreme Linux track) Atlanta, GA, October 12, 2000
[10] Aberdeen, D., Baxter, J., Edwards, R.: 92cents/MFlops/s, Ultra-Large-Scale Neural-Network Training on a PIII Cluster. In: IEEE/ACM proceedings of SC2000, Dallas, TX, November 4-10, 2000
[11] Hauser, Th., Mattox, T., LeBeau, R., Dietz, H., Huang, G.: High-Cost CFD on a Low-Cost Cluster. In: IEEE/ACM proceedings of SC2000, Dallas, TX, November 4-10, 2000

Exploiting Ownership Sets in HPF*

Pramod G. Joisha and Prithviraj Banerjee

Center for Parallel and Distributed Computing, Electrical and Computer Engineering Department, Technological Institute, 2145 Sheridan Road, Northwestern University, IL 60208–3118.
Phone: (847) 467–4610, Fax: (847) 491–4455
{pjoisha, banerjee}@ece.nwu.edu

Abstract. Ownership sets are fundamental to the partitioning of program computations across processors by the owner-computes rule. These sets arise due to the mapping of data arrays onto processors. In this paper,[a] we focus on how ownership sets can be efficiently determined in the context of the HPF language, and show how the structure of these sets can be symbolically characterized in the presence of arbitrary data alignment and data distribution directives. Our starting point is a system of equalities and inequalities due to Ancourt et al. that captures the array mapping problem in HPF. We arrive at a refined system that enables us to efficiently solve for the ownership set using the Fourier-Motzkin Elimination technique, and which requires the course vector as the only auxiliary vector. We develop[b] important and general properties pertaining to HPF alignments and distributions, and show how they can be used to eliminate redundant communication due to array replication. We also show how the generation of communication code can be avoided when pairs of array references are ultimately mapped onto the same processors. Experimental data demonstrating the improved code performance that the latter optimization enables is presented and discussed.

1 Introduction

In an automated code generation scenario, the compiler decides the processors on which to execute the various compute operations occurring in a program. In languages such as High Performance Fortran (HPF) [10], array mappings guide the computation partitioning process. They are specified by the programmer in terms of annotations called *directives*. The actual mapping process typically involves two steps: arrays are first *aligned* with a template and templates are then *distributed* onto virtual processor meshes. As a consequence of the alignment operation—performed via the `ALIGN` directive—each array element gets

* This research was partially supported by the National Science Foundation under Grant NSF CCR–9526325, and in part by DARPA under Contract F30602–98–2–0144.

[a] A longer version of this paper has been submitted to the *IEEE Transactions on Parallel and Distributed Processing*.

[b] The proofs for all lemmas and theorems are available in [9].

S.P. Midkiff et al. (Eds.): LCPC 2000, LNCS 2017, pp. 259–273, 2001.

assigned to at least a single template cell. Every template cell is then associated with exactly one processor through the `DISTRIBUTE` directive. In this way, array elements are eventually mapped onto processors.

In allocating program computations to processors, the compiler uses the mapping information associated with the data. A possible scheme known as the *owner-computes rule* is to allow only the owner of the left-hand side reference in an assignment statement to execute the statement. By the owner-computes rule, expressions that use elements located on the same processor can be evaluated locally on that processor, without the need for inter-processor communication. When the need to transfer remote data elements arises, the compiler produces the relevant communication code. Hence, the owner-computes rule leads to the notion of an *ownership set* which is the set of all data elements mapped onto a processor by virtue of the alignment and distribution directives in the program.

Since the assignment of computations to processors is determined by the allocation of data to processors, one of the aims of the HPF compilation problem is to find a suitable way for capturing the alignment and distribution information. Given such a means, the next issue that must be addressed is how the owner-computes rule can be realized using the representation. Does the proposed framework provide insights into the nature of the array mapping problem? Does the representation reveal general properties that can be leveraged to generate efficient code? In this paper, we investigate these questions in the context of a recent representation proposed by Ancourt et al [1].

1.1 Related Work

The problem of array alignment and distribution has been extensively studied and numerous structures have been suggested and examined that describe the mapping of arrays to processors [8, 6, 2, 13]. Early representations focused on `BLOCK` distributions alone and were incapable of conveniently describing the general `CYCLIC`(B) distribution. This deficiency was addressed in subsequent work by using techniques ranging from finite state machines, virtual processor meshes to set-theoretic methods [5, 7, 11]. However, these schemes primarily concentrated on enumerating local memory access sequences and handling array expressions. A generalized view of the HPF mapping problem was subsequently presented by Ancourt et al. [1] who showed how a system of equalities and inequalities could be used to mathematically express the *regular* alignment and distribution of arrays to processors. These systems were then used to formulate ownership sets and compute sets for loops qualified by the `INDEPENDENT` directive, and parametric solutions for the latter were provided based on the Hermite Normal Form [1].

1.2 Contributions

The system of equalities and inequalities in the Ancourt et al. framework uncover interesting properties that relate to the HPF mapping problem. We discuss these properties and show how some of them can be exploited to devise an efficient

run-time test that avoids redundant communication due to array replication. Our approach to solving for the ownership set is based on the Fourier-Motzkin Elimination (FME) technique, and in that respect, we deviate from [1]. We show how the originally proposed formulation for the ownership set can be refined to a form that involves the course vector as the only auxiliary vector and which also enables the efficient enumeration of the constituent elements of this set. We present a sufficient condition called the *mapping test* which eliminates the need for generating communication code for certain right-hand side array references in an assignment statement. The mapping test often results in marked performance improvements and this fact is substantiated by experimental data. The techniques mentioned in this paper have been incorporated into a new version of the PARADIGM compiler [4], using *Mathematica*[c] as the symbolic manipulation engine.

2 Preliminaries

Consider the synthesized declaration and HPF directives shown in Figure 1. Since the first dimension of `A` and the single subscript-triplet expression conform, this fragment is equivalent to that shown in Figure 2. The dummy variables i, j, k and l satisfy the constraints $-1 \leq i \leq 20$, $3 \leq j \leq 40$, $0 \leq k \leq 20$ and $0 \leq l \leq 99$ respectively.

```
...
REAL A(-1:20, 3:40, 0:20)
...

!HPF$ TEMPLATE T(0:99, 0:99, 0:99)
!HPF$ PROCESSORS P(1:9, 1:9)
!HPF$ ALIGN A(:, *, k) WITH T(2*k+1, 2:44:2, *)
!HPF$ DISTRIBUTE T(*, CYCLIC(4), BLOCK(13)) ONTO P
```

Fig. 1. Original Fragment

```
...
REAL A(-1:20, 3:40, 0:20)
...

!HPF$ TEMPLATE T(0:99, 0:99, 0:99)
!HPF$ PROCESSORS P(1:9, 1:9)
!HPF$ ALIGN A(i, j, k) WITH T(2*k+1, (i+1)*2+2, l)
!HPF$ DISTRIBUTE T(*, CYCLIC(4), BLOCK(13)) ONTO P
```

Fig. 2. Equivalent Fragment

The implications of the above alignment directives can be compactly expressed through the following collection of equalities and inequalities [1]:

$$\hat{\mathcal{R}}\boldsymbol{t} = \hat{\mathcal{A}}\boldsymbol{a} + \boldsymbol{s_0} - \hat{\mathcal{R}}\boldsymbol{l_T}, \tag{1}$$

$$\boldsymbol{a_l} \leq \boldsymbol{a} \leq \boldsymbol{a_u}, \tag{2}$$

$$\boldsymbol{0} \leq \boldsymbol{t} \leq \boldsymbol{u_T} - \boldsymbol{l_T}. \tag{3}$$

For the given example, the various matrices and vectors are

$$\hat{\mathcal{R}} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}, \hat{\mathcal{A}} = \begin{pmatrix} 0 & 0 & 2 \\ 2 & 0 & 0 \end{pmatrix}, \boldsymbol{s_0} = \begin{pmatrix} 1 \\ 4 \end{pmatrix},$$

[c] *Mathematica* is a registered trademark of Wolfram Research, Inc.

and,

$$\boldsymbol{a_l} = \begin{pmatrix} -1 \\ 3 \\ 0 \end{pmatrix}, \boldsymbol{a_u} = \begin{pmatrix} 20 \\ 40 \\ 20 \end{pmatrix}, \boldsymbol{l_T} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \boldsymbol{u_T} = \begin{pmatrix} 99 \\ 99 \\ 99 \end{pmatrix}.$$

In a similar vein, information relating to the distribution directives can be captured by the following system [1]:

$$\hat{\pi}\boldsymbol{t} = \hat{C}\hat{P}\boldsymbol{c} + \hat{C}\boldsymbol{p} + \boldsymbol{l}, \tag{4}$$

$$\hat{\lambda}\boldsymbol{c} = \boldsymbol{0}, \tag{5}$$

$$\boldsymbol{0} \leq \boldsymbol{p} < \hat{P}\boldsymbol{1}, \tag{6}$$

$$\boldsymbol{0} \leq \boldsymbol{l} < \hat{C}\boldsymbol{1}, \tag{7}$$

where $\boldsymbol{t}$ satisfies (3). The respective matrices for the running example become

$$\hat{\pi} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \hat{C} = \begin{pmatrix} 4 & 0 \\ 0 & 13 \end{pmatrix}, \hat{P} = \begin{pmatrix} 9 & 0 \\ 0 & 9 \end{pmatrix}, \hat{\lambda} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}.$$

While equation (1) is a consequence of the `ALIGN` directive, equations (4) and (5) are a result of the `DISTRIBUTE` directive. While (4) dictates the mapping of template cells onto processors, (5) indicates whether a particular processor dimension has a `BLOCK` or a `CYCLIC` distribution associated with it. Constraints on the array bounds vector $\boldsymbol{a}$ and the template cell vector $\boldsymbol{t}$ are given by (2) and (3) respectively. Finally, (6) and (7) describe the constraints that the processor identity vector $\boldsymbol{p}$ and the offsets vector $\boldsymbol{l}$ must satisfy. We shall use x, y and z to represent the number of dimensions of the *alignee* [10], template and processor mesh respectively. Using this notation, the column vectors $\boldsymbol{a}$ and $\boldsymbol{t}$ consist of x and y elements respectively, while the column vectors $\boldsymbol{p}$, $\boldsymbol{c}$ and $\boldsymbol{l}$ have z elements each.

We can now formulate the *ownership set* of a processor $\boldsymbol{p}$, which is defined with respect to an array `X`. It denotes those elements of `X` that are finally mapped onto the processor $\boldsymbol{p}$. In set-theoretic notation, this set becomes [1]

$$\begin{aligned} &\Delta_p(X) = \{\boldsymbol{a} | \exists \boldsymbol{t}, \boldsymbol{c}, \boldsymbol{l} \text{ such that} \\ &\hat{\mathcal{R}}\boldsymbol{t} = \hat{\mathcal{A}}\boldsymbol{a} + \boldsymbol{s_0} - \hat{\mathcal{R}}\boldsymbol{l_T} \\ &\wedge\ \hat{\pi}\boldsymbol{t} = \hat{C}\hat{P}\boldsymbol{c} + \hat{C}\boldsymbol{p} + \boldsymbol{l} \\ &\wedge\ \hat{\lambda}\boldsymbol{c} = \boldsymbol{0} \\ &\wedge\ \boldsymbol{a_l} \leq \boldsymbol{a} \leq \boldsymbol{a_u} \\ &\wedge\ \boldsymbol{0} \leq \boldsymbol{t} \leq \boldsymbol{u_T} - \boldsymbol{l_T} \\ &\wedge\ \boldsymbol{0} \leq \boldsymbol{l} < \hat{C}\boldsymbol{1}\}, \end{aligned} \tag{8}$$

where $\boldsymbol{0} \leq \boldsymbol{p} < \hat{P}\boldsymbol{1}$.

3 Ownership Sets Revisited

The new version of the PARADIGM compiler uses the system of equalities and inequalities described in § 2 to solve the HPF mapping problem [9]. The

PARADIGM approach to solving for the ownership, compute and communication sets is based on integer FME solutions [3]. Though the FME technique has a worst-case exponential time complexity [12], we have observed the method to be reasonably efficient in practice.

The elements of $\boldsymbol{c}$, $\boldsymbol{l}$ and $\boldsymbol{t}$ manifest as auxiliary loop variables when compute sets derived from ownership sets are used to partition loops [9]. It is therefore desirable to have a lesser number of unknowns from the standpoint of code generation. Reducing the number of unknowns also improves the overall timing of the FME solver. To meet these goals, the formulation for the ownership set given in (8) was refined using two transformations: $\boldsymbol{t}^* = \hat{\mathcal{R}}^T(\hat{\mathcal{A}}\boldsymbol{a} + \boldsymbol{s_0} - \hat{\mathcal{R}}\boldsymbol{l_T}) + (\hat{I} - \hat{\mathcal{R}}^T\hat{\mathcal{R}})\hat{\pi}^T(\hat{C}\hat{P}\boldsymbol{c} + \hat{C}\boldsymbol{p})$ and $\boldsymbol{c}^* = \hat{\pi}\hat{\mathcal{R}}^T\hat{\mathcal{R}}\hat{\pi}^T\boldsymbol{c}$. Details are available in [9].

Lemma 3.1: The following is an equivalent definition for the ownership set:

$$
\begin{aligned}
&\Delta_p(X) = \{\boldsymbol{a} | \exists \boldsymbol{c} \text{ such that} \\
&\hat{\mathcal{R}}^T\hat{\mathcal{R}}\hat{\pi}^T(\hat{C}\hat{P}\boldsymbol{c} + \hat{C}\boldsymbol{p}) \leq \hat{\pi}^T\hat{\pi}\hat{\mathcal{R}}^T(\hat{\mathcal{A}}\boldsymbol{a} + \boldsymbol{s_0} - \hat{\mathcal{R}}\boldsymbol{l_T}) \\
&\wedge\ \hat{\pi}^T\hat{\pi}\hat{\mathcal{R}}^T(\hat{\mathcal{A}}\boldsymbol{a} + \boldsymbol{s_0} - \hat{\mathcal{R}}\boldsymbol{l_T}) \leq \hat{\mathcal{R}}^T\hat{\mathcal{R}}\hat{\pi}^T(\hat{C}\hat{P}\boldsymbol{c} + \hat{C}\boldsymbol{p} + \hat{C}\mathbf{1} - \mathbf{1}) \\
&\wedge\ \mathbf{0} \leq \hat{C}\hat{P}\boldsymbol{c} + \hat{C}\boldsymbol{p} \leq \hat{\pi}(\boldsymbol{u_T} - \boldsymbol{l_T}) \\
&\wedge\ (\hat{I} - \hat{\pi}\hat{\mathcal{R}}^T\hat{\mathcal{R}}\hat{\pi}^T)\boldsymbol{c} = \mathbf{0} \\
&\wedge\ \hat{\lambda}\boldsymbol{c} = \mathbf{0} \\
&\wedge\ \boldsymbol{a}_l \leq \boldsymbol{a} \leq \boldsymbol{a}_u \\
&\wedge\ \mathbf{0} \leq \hat{\mathcal{A}}\boldsymbol{a} + \boldsymbol{s_0} - \hat{\mathcal{R}}\boldsymbol{l_T} \leq \hat{\mathcal{R}}(\boldsymbol{u_T} - \boldsymbol{l_T})\}
\end{aligned}
\tag{9}
$$

where, for every $\boldsymbol{a} \in \Delta_p(X)$, there exists exactly one $\boldsymbol{c}$ for which the system in (9) holds.

An important consequence of Lemma 3.1 is that if the FME technique is now applied to the system in (9), then, whatever be the order of elimination of the unknown variables (corresponding to the elements of $\boldsymbol{c}$ and $\boldsymbol{a}$), the associated loop nest scans every member of the ownership set (i.e., $\boldsymbol{a}$) exactly once. To see why this is so, let $\boldsymbol{\epsilon}$ represent one such elimination order. Suppose $\theta = \|\boldsymbol{\epsilon}\| = x+z$ where x and z denote the number of dimensions of the alignee and processor mesh respectively. The integer bound expressions returned by the FME solver can be used to construct a loop nest that scans $\Delta_p(X)$. The outermost loop in this nest matches ϵ_1, while the innermost loop matches ϵ_θ. Consider an iteration point $\boldsymbol{\rho}$ of such a loop nest and let $\boldsymbol{\varrho}$ be any other iteration point of the same loop nest. Thus, $\boldsymbol{\rho}$ and $\boldsymbol{\varrho}$ also represent solutions to the system in (9). Since every iteration point of a loop nest is distinct, let $\boldsymbol{\rho}$ and $\boldsymbol{\varrho}$ differ in the ith position. If $\epsilon_i \equiv a_n$, then the $\boldsymbol{a}$ that corresponds to $\boldsymbol{\rho}$ obviously differs from the $\boldsymbol{a}$ that corresponds to $\boldsymbol{\varrho}$. If instead $\epsilon_i \equiv c_j$, then the $\boldsymbol{c}$ that corresponds to $\boldsymbol{\rho}$ differs from the $\boldsymbol{c}$ that corresponds to $\boldsymbol{\varrho}$. But from Lemma 3.1, two different course vectors cannot satisfy the system for the same $\boldsymbol{a}$. Thus, the corresponding values for $\boldsymbol{a}$ in $\boldsymbol{\rho}$ and $\boldsymbol{\varrho}$ must also be different. That is, the $\boldsymbol{a}$ associated with the iteration point $\boldsymbol{\rho}$ must be different from the $\boldsymbol{a}$ associated with any other iteration point $\boldsymbol{\varrho}$ of the same loop nest. In other words, every member of the ownership set gets enumerated exactly once.

Note that if the FME technique were applied to the system in (8) and if a loop nest were generated from the resulting solution system to scan $\Delta_p(X)$, it is not guaranteed that every member of $\Delta_p(X)$ will get enumerated exactly once. This inability to ensure the "uniqueness" of each enumerated member has serious repercussions if the ownership set is used to generate other sets. For instance, we use the compute sets defined in [1] to handle loops qualified by the `INDEPENDENT` directive. In [1], these sets are defined using a formulation of the ownership set similar to that in (8). If the FME technique were applied directly to the compute set formulation given in [1], certain iterations in this set could get executed more than once for certain alignment and distribution combinations. However, if the formulation in (9) is used, this problem can be avoided. The FME approach that we adopt to solve for these sets is quite different from the approach in [1] where a parametric solution based on the Hermite Normal Form was exploited for the same purpose.

Though the system in (9) has the same number of inequalities as the formulation in (8), the number of unknowns to be eliminated is lesser by $y + z$ variables. This resulted in improved solution times for the FME solver. For instance, when $\Delta_p(A)$ was computed for the array `A` in the example in § 2, a significant reduction in the time required to solve the system was noticed; while solving the system in (8) took 0.58 seconds, solving the system in (9) took only 0.25 seconds [9].

4 An Equivalence Relation

It is interesting to enquire into the nature of ownership sets across processors. That is, for an arbitrary alignment and distribution combination, can these sets partially overlap? Or, are they equal or disjoint? The answers to these questions can be used to devise an efficient run-time test that avoids redundant communication due to array replication (see § 5).

Lemma 4.1: If $\hat{\pi}\hat{\mathcal{R}}^T\hat{\mathcal{R}}\hat{\pi}^T(\boldsymbol{p} - \boldsymbol{q}) = \mathbf{0}$, and $\Delta_p(X) \neq \emptyset$, $\Delta_q(X) \neq \emptyset$, then $\Delta_p(X) = \Delta_q(X)$.

To comprehend the meaning of Lemma 4.1, an understanding of the expression $\hat{\pi}\hat{\mathcal{R}}^T\hat{\mathcal{R}}\hat{\pi}^T$ is necessary. The product $\hat{\pi}\hat{\mathcal{R}}^T\hat{\mathcal{R}}\hat{\pi}^T$ is a square diagonal matrix of size $z \times z$. It is easy to see that the principal diagonal elements of this matrix are either 0 or 1. It is also easy to see that the jth principal diagonal element is 0 if and only if the template dimension distributed on the jth processor dimension is a replicated dimension. What is a "replicated dimension"? We refer to those dimensions of a template that do contain a `*` in the alignment specification as its *replicated dimensions*.[d] The remaining non-`*` dimensions are called its *aligned dimensions*. The definitions of an aligned and replicated dimension arise on account of a particular `ALIGN` directive and are always spoken of in connection with

[d] Specifically, replicated dimensions are those dimensions of a template that contain either a `*` or an *unmatched* dummy variable in the alignment specification (see [10]).

that directive. For example, in Figure 1, the third dimension of T is a replicated dimension while its first and second dimensions are the aligned dimensions in that ALIGN directive.

Suppose that the array X is aligned to a template T that is then distributed onto the processor mesh P. Lemma 4.1 states that if the coordinates $\boldsymbol{p}$ and $\boldsymbol{q}$ of two processors in P match in at least those dimensions onto which the aligned dimensions of T are distributed, and if $\boldsymbol{p}$ and $\boldsymbol{q}$ own at least one element of the alignee array X, then their ownership sets with respect to X must be the same.

Lemma 4.2: If $\Delta_p(X) \cap \Delta_q(X) \neq \emptyset$, then $\hat{\pi}\hat{\mathcal{R}}^T\hat{\mathcal{R}}\hat{\pi}^T(\boldsymbol{p} - \boldsymbol{q}) = \boldsymbol{0}$.

The reverse is also true; if the ownership sets with respect to X of two processors in P overlap, then their coordinates along those dimensions of P onto which the aligned dimensions of T are distributed must match. This is what Lemma 4.2 states. The above two lemmas can be used to prove the following theorem.

Theorem I

For a given array X, the ownership sets across processors must be either equal or disjoint. That is,

$$\Delta_p(X) = \Delta_q(X),$$
or
$$\Delta_p(X) \cap \Delta_q(X) = \emptyset.$$

Proof:

Suppose $\Delta_p(X)$ and $\Delta_q(X)$ are not disjoint. Then $\Delta_p(X) \cap \Delta_q(X) \neq \emptyset$. Hence, from Lemma 4.2, we get

$$\hat{\pi}\hat{\mathcal{R}}^T\hat{\mathcal{R}}\hat{\pi}^T(\boldsymbol{p} - \boldsymbol{q}) = \boldsymbol{0}. \tag{I.1}$$

Since we have assumed that $\Delta_p(X) \cap \Delta_q(X) \neq \emptyset$, then $\Delta_p(X) \neq \emptyset$ and $\Delta_q(X) \neq \emptyset$. By Lemma 4.1, this fact and (I.1) therefore imply

$$\Delta_p(X) = \Delta_q(X).$$

Thus, either $\Delta_p(X) \cap \Delta_q(X) = \emptyset$ or $\Delta_p(X) = \Delta_q(X)$ must be true.

Let us define a binary relation $\sim$ on a mapped array such that given two array elements $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$, $\boldsymbol{\beta} \sim \boldsymbol{\gamma}$ if and only if $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$ are mapped onto the same processors. The rules of HPF ensure that for any legal ALIGN and DISTRIBUTE combination, every element of the mapped array will reside on at least one processor $\boldsymbol{p}$ [9]. Hence, $\sim$ must be reflexive. Also, if $\boldsymbol{\beta} \sim \boldsymbol{\gamma}$, $\boldsymbol{\gamma} \sim \boldsymbol{\beta}$ is obviously true. Therefore, $\sim$ is symmetric. Finally, if $\boldsymbol{\beta} \sim \boldsymbol{\gamma}$ and $\boldsymbol{\gamma} \sim \boldsymbol{\delta}$ are true, then from Theorem I, $\boldsymbol{\beta} \sim \boldsymbol{\delta}$. That is, $\sim$ is transitive. Hence, the ALIGN and DISTRIBUTE directives for a mapped array define an equivalence relation on that array.

5 The Replication Test

Let $\nabla_p(S', Y(\hat{S}_y\boldsymbol{\alpha} + \boldsymbol{a_{0_y}}))$ indicate those elements of a right-hand side array reference $Y(\hat{S}_y\boldsymbol{\alpha} + \boldsymbol{a_{0_y}})$ contained in an assignment statement S' that $\boldsymbol{p}$ views on account of its compute work in S'. Thus $\boldsymbol{p}$ would have to potentially fetch these elements from a remote processor $\boldsymbol{q}$ and the set of elements to be received would be $\Delta_q(Y) \cap \nabla_p(S', Y(\hat{S}_y\boldsymbol{\alpha} + \boldsymbol{a_{0_y}}))$. Likewise, $\boldsymbol{p}$ would have to potentially send the elements in $\Delta_p(Y) \cap \nabla_q(S', Y(\hat{S}_y\boldsymbol{\alpha} + \boldsymbol{a_{0_y}}))$ to a remote processor $\boldsymbol{q}$ because $\boldsymbol{q}$ may in turn view the elements owned by $\boldsymbol{p}$. Code fragments in Figure 3 illustrate how such data exchange operations can be realized.

```
for each 0 ≤ q < P̂1
    if p ≠ q then
       send  Δ_p(Y) ∩ ∇_q(S', Y(Ŝ_y α + a_{0_y}))  to q
    endif
endfor
                          for each 0 ≤ q < P̂1
                              if p ≠ q then
                                 receive  Δ_q(Y) ∩ ∇_p(S', Y(Ŝ_y α + a_{0_y}))  from q
                              endif
                          endfor
```

Fig. 3. Send and Receive Actions at Processor $\boldsymbol{p}$

In § 3, we saw that if $\Delta_p(Y) \neq \emptyset$, and $\Delta_q(Y) \neq \emptyset$, then $\Delta_p(Y)$ and $\Delta_q(Y)$ are equal if and only if $\hat{\pi}\hat{\mathcal{R}}^T\hat{\mathcal{R}}\hat{\pi}^T(\boldsymbol{p}-\boldsymbol{q}) = \boldsymbol{0}$. This property can be used to avoid redundant communication due to array replication; the modified code fragments in Figure 4 show this optimization.

```
for each 0 ≤ q < P̂1
    if (Δ_q(Y) = ∅) ∨ (π̂R̂^T R̂ π̂^T(q − p) ≠ 0) then
       send  Δ_p(Y) ∩ ∇_q(S', Y(Ŝ_y α + a_{0_y}))  to q
    endif
endfor
                          for each 0 ≤ q < P̂1
                              if (Δ_p(Y) = ∅) ∨ (π̂R̂^T R̂ π̂^T(p − q) ≠ 0) then
                                 receive  Δ_q(Y) ∩ ∇_p(S', Y(Ŝ_y α + a_{0_y}))  from q
                              endif
                          endfor
```

Fig. 4. Send and Receive Actions with the Replication Test Optimization

Once the integer FME solution for the system of equalities and inequalities which describe the ownership set is obtained, computing whether $\Delta_p(Y)$ is the empty set for a given $\boldsymbol{p}$ only incurs an additional polynomial-time overhead [9]. The key idea that enables this decision is that in the FME solution system for (9), a particular p_j will occur in the bound expressions for c_j (p_j and c_j are elements in $\boldsymbol{p}$ and $\boldsymbol{c}$ respectively). That is, there will be an inequality pair of the form

$$f_j(p_j) \leq c_j \leq g_j(p_j) \tag{10}$$

in the solution system. In addition, there *could be one more* inequality pair in the FME solution system that contains p_j in its bound expressions. This inequality pair will have the form $F_j(p_j, c_j) \leq a_{n_j} \leq G_j(p_j, c_j)$. Besides the above two inequality pairs, there can be no other inequality pair in the FME solution system that also contains p_j. Hence, if (9) has a solution $\boldsymbol{a}$ for a given $\boldsymbol{p}$, each of the z disjoint inequality groups

$$\begin{gathered} f_j(p_j) \leq c_j \leq g_j(p_j), \\ F_j(p_j, c_j) \leq a_{n_j} \leq G_j(p_j, c_j) \end{gathered}$$

must *independently* admit a solution. The task of checking whether each of these groups has a solution for a particular p_j is clearly of quadratic complexity. Hence, the complexity of ascertaining whether $\Delta_p(Y)$ is non-empty for a given $\boldsymbol{p}$ is polynomial. Since evaluating the condition $\hat{\pi}\hat{\mathcal{R}}^T\hat{\mathcal{R}}\hat{\pi}^T(\boldsymbol{p}-\boldsymbol{q}) \neq \mathbf{0}$ is of $O(z)$ time complexity (as is the test $\boldsymbol{p} \neq \boldsymbol{q}$), the overall run-time complexity to evaluate the Boolean predicate, given the integer FME solution system for the ownership set (which is known at compile-time), becomes polynomial.

Observe that in the absence of replication, $\hat{\mathcal{R}}^T\hat{\mathcal{R}}$ is the identity matrix; in this situation, $\hat{\pi}\hat{\mathcal{R}}^T\hat{\mathcal{R}}\hat{\pi}^T(\boldsymbol{p}-\boldsymbol{q}) \neq \mathbf{0}$ if and only if $\boldsymbol{p} \neq \boldsymbol{q}$. Hence, in the absence of replication, the test degenerates to the usual $\boldsymbol{p} \neq \boldsymbol{q}$ condition; we therefore refer to the Boolean predicate $\hat{\pi}\hat{\mathcal{R}}^T\hat{\mathcal{R}}\hat{\pi}^T(\boldsymbol{p}-\boldsymbol{q}) \neq \mathbf{0}$ as the *replication test*.

6 The Mapping Test

Consider the assignment statement S' contained in a loop nest characterized by the loop iteration vector $\boldsymbol{\alpha}$, and in which the subscript expressions are linear:

$$X(\hat{S}_x\boldsymbol{\alpha} + \boldsymbol{a}_{\mathbf{0_x}}) = \cdots + Y(\hat{S}_y\boldsymbol{\alpha} + \boldsymbol{a}_{\mathbf{0_y}}) + \cdots$$

Consider $\Delta_p(Y) \cap \nabla_q(S', Y(\hat{S}_y\boldsymbol{\alpha} + \boldsymbol{a}_{\mathbf{0_y}}))$ and $\Delta_q(Y) \cap \nabla_p(S', Y(\hat{S}_y\boldsymbol{\alpha} + \boldsymbol{a}_{\mathbf{0_y}}))$ shown in Figures 3 and 4 respectively. These communication sets would be generated for the above assignment statement and take into account the relative alignments and distributions of the left-hand side and right-hand side array references. In the event of these communication sets being null, no communication occurs at run-time. However, the overhead of checking at run-time whether a particular processor must necessarily dispatch a section of its array to some other processor that views it exists, irrespective of whether data is actually communicated or not. This could result in the expensive run-time cost of communication

checks, which could have been avoided all together, if the compiler had the capacity to ascertain that the elements of the array references $X(\hat{S}_x\boldsymbol{\alpha} + \boldsymbol{a}_{\mathbf{0}_x})$ and $Y(\hat{S}_y\boldsymbol{\alpha} + \boldsymbol{a}_{\mathbf{0}_y})$ are in fact ultimately mapped onto the same processor. This is precisely what the mapping test attempts to detect.

The mapping test (elaborated in Lemma 6.1) is a sufficient condition which when true guarantees that the processor owning the left-hand side array reference of an assignment statement also *identically* owns a particular right-hand side array reference of the same statement. By "identically owns," we mean that for *any* value of the loop iteration vector $\boldsymbol{\alpha}$ (not necessarily those limited by the bounds of the enclosing loop nest), the right-hand side array element $Y(\hat{S}_y\boldsymbol{\alpha} + \boldsymbol{a}_{\mathbf{0}_y})$ resides on the same processor that owns the left-hand side array element $X(\hat{S}_x\boldsymbol{\alpha} + \boldsymbol{a}_{\mathbf{0}_x})$.

Lemma 6.1: Define two vectors $\boldsymbol{\zeta}$ and $\boldsymbol{\xi}$ for the assignment statement S':

$$\begin{aligned}\boldsymbol{\zeta} &= \hat{\pi}_T\hat{\mathcal{R}}_x^T(\hat{\mathcal{A}}_x(\hat{S}_x\boldsymbol{\alpha} + \boldsymbol{a}_{\mathbf{0}_x}) + \boldsymbol{s}_{\mathbf{0}_x} - \hat{\mathcal{R}}_x\boldsymbol{l}_{\boldsymbol{T}}),\\ \boldsymbol{\xi} &= \hat{\pi}_S\hat{\mathcal{R}}_y^T(\hat{\mathcal{A}}_y(\hat{S}_y\boldsymbol{\alpha} + \boldsymbol{a}_{\mathbf{0}_y}) + \boldsymbol{s}_{\mathbf{0}_y} - \hat{\mathcal{R}}_y\boldsymbol{l}_{\boldsymbol{S}}).\end{aligned}$$

Here we assume that `T` and `S` are the templates to which `X` and `Y` are respectively aligned, and that both `T` and `S` are distributed onto a processor mesh `P`. If $\hat{\mu} = \hat{\pi}_T\hat{\mathcal{R}}_x^T\hat{\mathcal{R}}_x\hat{\pi}_T^T$ and $\hat{\nu} = \hat{\pi}_S\hat{\mathcal{R}}_y^T\hat{\mathcal{R}}_y\hat{\pi}_S^T$, then

$$\begin{aligned}&\Delta_p(Y) \neq \emptyset\ \forall\ \mathbf{0} \leq \boldsymbol{p} < \hat{P}\mathbf{1}\\ &\wedge\ \hat{\nu} \leq \hat{\mu}\\ &\wedge\ \hat{\nu}(\lfloor\hat{C}_T^{-1}\boldsymbol{\zeta}\rfloor \bmod \hat{P}\mathbf{1}) = \lfloor\hat{C}_S^{-1}\boldsymbol{\xi}\rfloor \bmod \hat{P}\mathbf{1}\end{aligned} \tag{11}$$

is a sufficient condition for the array reference $Y(\hat{S}_y\boldsymbol{\alpha} + \boldsymbol{a}_{\mathbf{0}_y})$ to be identically mapped onto the same processor which owns $X(\hat{S}_x\boldsymbol{\alpha} + \boldsymbol{a}_{\mathbf{0}_x})$.

The sufficient condition stated in Lemma 6.1 can be established at compile-time. The predicate (11) forms the actual mapping test. Verifying whether $\hat{\nu} \leq \hat{\mu}$ and $\hat{\nu}(\lfloor\hat{C}_T^{-1}\boldsymbol{\zeta}\rfloor \bmod \hat{P}\mathbf{1}) = \lfloor\hat{C}_S^{-1}\boldsymbol{\xi}\rfloor \bmod \hat{P}\mathbf{1}$ is an $O(z)$ time complexity operation. Establishing whether $\Delta_p(Y) \neq \emptyset\ \forall\ \mathbf{0} \leq \boldsymbol{p} < \hat{P}\mathbf{1}$ is a polynomial time operation given the symbolic representation of the ownership set (see § 5 and [9]). Thus, the overall time complexity for verifying the requirements of Lemma 6.1 is polynomial, once the FME solution system for the ownership set is known.

The impact of the mapping test on run times can often be dramatic. To illustrate the savings, the run times for the ADI benchmark, for arrays of sizes $4\times1024\times2$ on a 1×4 mesh of processors, with and without this optimization were 0.51 and 64.79 seconds respectively [9]! The large value of 64.79 seconds arose due to three assignment statements which were the sinks of loop-independent flow dependencies and which were enclosed within a triply nested loop spanning an iteration space of $2048\times2\times1022$ points. Each of the three assignment statements included right-hand side array references that were finally distributed onto the same processor as the corresponding left-hand side array reference. Hence, in

all, eighteen communication checks (nine for `MPI_SEND` and another nine for `MPI_RECV`) per iteration were eliminated.

7 Mapping Test Measurements

Execution times and compilation times were measured for the PARADIGM (version 2.0) system with and without the mapping test optimization. For the sake of comparison, execution times and compilation times for the original sequential sources and the parallelized codes generated by `pghpf` (version 2.4) and `xlhpf` (version 1.03) were also recorded. `pghpf` and `xlhpf` are commercial HPF compilers from the Portland Group Inc., (PGI) and the International Business Machines (IBM) respectively. In the input codes to the `pghpf` compiler, `DO` loops were recast into `FORALL` equivalents where possible and were qualified with the `INDEPENDENT` directive when appropriate. The `FORALL` construct and the `INDEPENDENT` directive were not mixed in the inputs to `pghpf` and the tabulated execution times correspond to the best of the two cases. All of the PARADIGM measurements were done in the presence of the replication test.

7.1 System Specifications

The IBM compilers `xlf` and `mpxlf` were used to handle Fortran 77 and Fortran 77+MPI sources respectively. The HPF sources were compiled using `xlhpf` and `pghpf`. The `-O` option, which results in the generation of optimized code, was always used during compilations done with `xlf`, `xlhpf`, `mpxlf` and `pghpf`. Compilation times were obtained by considering the source-to-source transformation effected by PARADIGM, *as well as* the source-to-executable compilation done using `mpxlf` (version 5.01). The source-to-source compilation times for PARADIGM were measured on an HP Visualize C180 with a 180MHz HP PA-8000 CPU, running HP-UX 10.20 and having 128MB of RAM. Compilation times for `pghpf`, `xlhpf` as well as `mpxlf` were measured on an IBM E30 running AIX 4.3 and having a 133MHz PowerPC 604 processor and 96MB of main memory. In those tables that tabulate the execution times, the RS6000 column refers to the sequential execution times obtained on the IBM E30. The parallel codes were executed on a 16-node IBM SP2 multicomputer, running AIX 4.3 and in which each processor was a 62.5MHz POWER node having 128MB of RAM. Inter-processor communication on the IBM SP2 was across a high performance adapter switch.

7.2 Alignments and Distributions

Measurements for the mapping test were taken across three benchmarks—ADI, Euler Fluxes (from FLO52 in the Perfect Club Suite) and Matrix Multiplication. For all input samples, fixed templates and alignments were chosen; these are shown in Figure 5. Note that the most suitable alignments were chosen for

```
!HPF$ TEMPLATE T(4, 1024, 2)
!HPF$ ALIGN DU1(i) WITH T(*, i, *)
!HPF$ ALIGN DU2(i) WITH T(*, i, *)
!HPF$ ALIGN DU3(i) WITH T(*, i, *)
!HPF$ ALIGN AU1(i, j, k) WITH T(i, j, k)
!HPF$ ALIGN AU2(i, j, k) WITH T(i, j, k)
!HPF$ ALIGN AU3(i, j, k) WITH T(i, j, k)
```

```
!HPF$ TEMPLATE T(0:5001, 34, 4)
!HPF$ ALIGN FS(i, j, k) WITH T(i, j, k)
!HPF$ ALIGN DW(i, j, k) WITH T(i, j, k)
!HPF$ ALIGN W(i, j, k) WITH T(i, j, k)
!HPF$ ALIGN X(i, j, k) WITH T(i, j, k)
!HPF$ ALIGN P(i, j) WITH T(i, j, *)
```

```
!HPF$ TEMPLATE S(1024, 1024)
!HPF$ TEMPLATE T(1024, 1024)
!HPF$ ALIGN A(i, *) WITH S(i, *)
!HPF$ ALIGN B(*, j) WITH S(*, j)
!HPF$ ALIGN C(i, j) WITH T(i, j)
```

Fig. 5. ADI, Euler Fluxes and Matrix Multiplication

the benchmark input samples. For the Matrix Multiplication and ADI benchmarks, these alignments resulted in communication-free programs irrespective of the distributions. For the Euler Fluxes benchmark, the distributions resulted in varying amounts of communication.

The `PROCESSORS` and the `DISTRIBUTE` directives were changed in every benchmark's input sample. The various distributions were chosen *arbitrarily*, the idea being to demonstrate the ability of the mapping test to handle any given alignment and distribution combination.

Table 1. Execution Times in Seconds

Benchmark	Processor Array Size	Distribution	IBM AIX 4.3				
			RS6000	SP2			
			xlf v 5.01	pdm v 2.0		pghpf v 2.4	xlhpf v 1.03
				Optimized	Non-optimized		
ADI	1×2	(BLOCK, BLOCK, *)	7.07	1.41	83.57	16.78	3.36
	1×4	(BLOCK, BLOCK, *)	6.79	0.51	64.79	12.67	2.35
	1×8	(BLOCK, CYCLIC(120), *)	7.90	4.34	609.44	20.94	-[e]
Euler Fluxes	1×2	(BLOCK, BLOCK, *)	71.85	19.03	19.44	231.67	33.79
	2×2	(BLOCK, *, CYCLIC)	71.40	13.47	13.83	274.93	3113.51
	8×1	(*, BLOCK, CYCLIC)	71.49	5.96	6.39	91.83	8.17
Matrix Multiplication	2×1	(BLOCK, BLOCK)[f]	12.65	2.47	5.83	57.48	25.88
	2×2	(BLOCK, BLOCK)[g]	104.96	10.06	17.20	224.29	100.37
	4×2	(BLOCK, CYCLIC(120))[g]	104.74	5.71	205.00	123.59	-[e]

7.3 Analysis

As Table 1 reveals, the benefits of the mapping test were most pronounced for the ADI benchmark, followed by the Matrix Multiplication benchmark. In the case of the Euler Fluxes benchmark, the mapping test eliminated six communication checks per iteration for the first input sample, and eight communication checks per iteration for the second and third input samples. In the absence of the mapping test, ten communication checks per iteration were generated. On account of loop-carried flow dependencies, the associated communication code was hoisted immediately within the outermost loop. However, since the number of iterations of the outermost loop was a mere 100, the optimized compiled codes did not exhibit any significant improvement in run times. For all of the ADI benchmark input samples, the iteration space comprised of $2048 \times 2 \times 1022$ points,

and the communication codes generated in the absence of the mapping test were hoisted immediately within the innermost loop. For the three Matrix Multiplication benchmark samples, the number of iteration points were $512 \times 512 \times 512$, $1024 \times 1024 \times 1024$ and $1024 \times 1024 \times 1024$ respectively and the single communication check that was generated in the absence of the mapping test was hoisted within the second innermost loop.

Table 2. Compilation Times in Seconds

Benchmark	Processor Array Size	Distribution	pdm v 2.0		mpxlf v 5.01		pghpf v 2.4	xlhpf v 1.03
			Optimized	Non-optimized	Optimized	Non-optimized		
ADI	1×2	(BLOCK, BLOCK, *)	6.00	9.00	2.00	7.00	7.00	5.00
	1×4	(BLOCK, BLOCK, *)	7.00	8.00	2.00	7.00	8.00	5.00
	1×8	(BLOCK, CYCLIC(120), *)	7.00	11.00	2.00	17.00	8.00	-[e]
Euler Fluxes	1×2	(BLOCK, BLOCK, *)	11.00	12.00	6.00	8.00	7.00	6.00
	2×2	(BLOCK, *, CYCLIC)	10.00	14.00	4.00	11.00	9.00	12.00
	8×1	(*, BLOCK, CYCLIC)	11.00	12.00	6.00	22.00	9.00	5.00
Matrix Multiplication	2×1	(BLOCK, BLOCK)[f]	4.00	5.00	2.00	2.00	5.00	1.00
	2×2	(BLOCK, BLOCK)[g]	4.00	5.00	2.00	3.00	6.00	1.00
	4×2	(BLOCK, CYCLIC(120))[g]	5.00	5.00	2.00	3.00	6.00	-[e]

Given a sequential input source written using Fortran 77 and having HPF directives, PARADIGM produces an SPMD output consisting of Fortran 77 statements and procedure calls to the MPI library. The compilation of this SPMD code into the final executable is then performed using `mpxlf`. Since the mapping test eliminates the generation of communication code where possible, it also exerts an influence on the *overall* compilation times. That is, the application of the mapping test often results in the generation of a smaller intermediate SPMD code, and this improves on the back-end source-to-executable compilation time. In our setup, this was done using `mpxlf`. Note that applying the mapping test *does not necessarily* mean an increased time for the source-to-source compilation phase performed by PARADIGM. This is because though compilation in the presence of the mapping test involves the additional effort of identifying the candidate array reference pairs that are identically mapped, it however saves on the communication code generation part which would otherwise have to be done for the same array reference pairs. Hence, compilation times for the source-to-source compilation phase may in fact be more in the absence of the mapping test and this was found to be true for nearly all of the benchmark samples tested. However, as Table 2 also reveals, the back-end compilation times were nearly always more in the absence of the mapping test, and this was because of the larger intermediate SPMD code sizes handled.

[e] `xlhpf` does not permit a `CYCLIC` blocking factor greater than 1.
[f] Arrays were of type `REAL`; array sizes were 512×512.
[g] Arrays were of type `REAL`; array sizes were 1024×1024.

8 Summary

The preceding sections have shown certain basic and interesting properties that ownership sets exhibit, even in the presence of arbitrary alignments and distributions. Our approach to solving for the ownership set and other sets derived from it is based on integer FME solutions to the systems characterizing these sets. We have also shown how the system of equalities and inequalities originally proposed in [1] can be refined to a form requiring the course vector as the only auxiliary vector. This refinement is beneficial to the FME approach. The fundamental property of ownership set equivalence is derived and we demonstrate how it can be used to eliminate redundant communication due to array replication. We also briefly describe how to efficiently make decisions regarding the "emptiness" of an ownership set. Finally, we derive a sufficient condition which when true ensures that a right-hand side array reference of an assignment statement is available on the same processor that owns the left-hand side array reference, thus making it possible to avoid generating communication code for the pair.

The mapping test is a very useful optimization. Its positive effect was observable in the case of other benchmarks such as Jacobi, TOMCATV and 2-D Explicit Hydrodynamics (from the Livermore Kernel 18), and was significant in most situations. This was on account of the fact that typically, suitably chosen `ALIGN` and `DISTRIBUTE` directives perfectly align and distribute at least one pair of left-hand side and right-hand side array references in at least one assignment statement of the program, and such alignments and distributions are often valid whatever be the values that the loop iteration vector ranges through.

Thus, by efficiently exploiting the ownership set, efficient SPMD code can be generated efficiently at compile-time.

References

[1] C. Ancourt, F. Coelho, F. Irigoin, and R. Keryell. "A Linear Algebra Framework for Static HPF Code Distribution". Technical Report A–278–CRI, Centre de Recherche en Informatique, École Nationale Supérieure des Mines de Paris, 35, rue Saint-Honoroé, F–77305 Fontainebleau cedex, France, November 1995.

[2] V. Balasundaram. "A Mechanism for keeping Useful Internal Information in Parallel Programming Tools—The Data Access Descriptor". *Journal of Parallel and Distributed Computing*, 9(2):154–170, June 1990.

[3] U. Banerjee. **Loop Transformations for Restructuring Compilers: The Foundations**. Kluwer Academic Publishers, Norwell, MA 02061, USA, January 1993. ISBN 0–7923–9318–X.

[4] P. Banerjee, J. A. Chandy, M. Gupta, E. W. Hodges IV, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Y.-H. Su. "The PARADIGM Compiler for Distributed-Memory Multicomputers". *IEEE Computer*, 28(10):37–47, October 1995.

[5] S. Chatterjee, J. R. Gilbert, F. J. E. Long, R. S. Schreiber, and S.-H. Teng. "Generating Local Addresses and Communication Sets for Data-Parallel Programs". In the *4th ACM SIGPLAN Symposium on Principles*

and Practices of Parallel Programming, pages 149–158, San Diego, CA, USA, May 1993.

[6] D. Callahan and K. Kennedy. "Analysis of Interprocedural Side Effects in a Parallel Programming Environment". In the *1st ACM International Conference on Supercomputing*, pages 138–171, Athens, Greece, May 1987.

[7] S. K. S. Gupta, S. D. Kaushik, S. Mufti, S. Sharma, C.-H. Huang, and P. Sadayappan. "On Compiling Array Expressions for Efficient Execution on Distributed-Memory Machines". In *22nd International Conference on Parallel Processing*; Editors: A. Choudhary and P. B. Berra, volume II of *Software*, pages 301–305, St. Charles, IL, USA, August 1993. The Pennsylvania State University, CRC Press, Inc., Boca Raton, FL, USA.

[8] M. Gupta. "Automatic Data Partitioning on Distributed Memory Multicomputers". Ph.D. dissertation, University of Illinois at Urbana-Champaign, Department of Electrical and Computer Engineering, September 1992.

[9] P. G. Joisha and P. Banerjee. "Analyzing Ownership Sets in HPF". Technical Report CPDC–TR–9906–012, Center for Parallel and Distributed Computing, Department of Electrical and Computer Engineering, Northwestern University, 2145 Sheridan Road, Evanston, IL, USA, January 1999.

[10] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele Jr., and M. E. Zosel. **The High Performance Fortran Handbook**. Scientific and Engineering Computation Series. The MIT Press, Cambridge, MA 02142, USA, 1994. ISBN 0–262–61094–9.

[11] S. P. Midkiff. "Local Iteration Set Computation for Block-Cyclic Distributions". In *24th International Conference on Parallel Processing*; Editor: C. D. Polychronopoulos, volume II of *Software*, pages 77–84, Oconomowoc, WI, USA, August 1995. The Pennsylvania State University, CRC Press, Inc., Boca Raton, FL, USA.

[12] A. Schrijver. **Theory of Linear and Integer Programming**. Wiley-Interscience Series in Discrete Mathematics. John Wiley & Sons, Inc., New York City, NY 10158, USA, 1986. ISBN 0–471–90854–1.

[13] E. Y.-H. Su, D. J. Palermo, and P. Banerjee. "Processor Tagged Descriptors: A Data Structure for Compiling for Distributed-Memory Multicomputers". In the *1994 International Conference on Parallel Architectures and Compilation Techniques*, pages 123–132, Montréal, Canada, August 1994.

A Performance Advisor Tool for Shared-Memory Parallel Programming*

Seon Wook Kim, Insung Park, and Rudolf Eigenmann

School of Electrical and Computer Engineering
Purdue University, West Lafayette, IN 47907-1285

Abstract. Optimizing a parallel program is often difficult. This is true, in particular, for inexperienced programmers who lack the knowledge and intuition of advanced parallel programmers. We have developed a framework that addresses this problem by automating the analysis of static program information and performance data, and offering active suggestions to programmers. Our tool enables experts to transfer programming experience to new users. It complements today's parallelizing compilers in that it helps to tune the performance of a compiler-optimized parallel program. To show its applicability, we present two case studies that utilize this system. By simply following the suggestions of our system, we were able to reduce the execution time of benchmark programs by as much as 39%.

1 Introduction

Parallelization, performance data analysis and program tuning are very difficult tasks for inexperienced parallel programmers. Tools such as parallelizing compilers and visualization systems help facilitate this process. Today's state-of-the-art parallelization and visualization tools provide efficient automatic utilities and ample choices for viewing and monitoring the behavior of parallel applications.

Nevertheless, tasks such as identifying performance problems and finding the right solutions have remained cumbersome to many programmers. Meaningful interpretation of a large amount of performance data is challenging and takes significant time and effort. Once performance bottlenecks are found through analysis, programmers need to study code regions and devise remedies to address the problems. Programmers generally rely on their knowhow and intuition to accomplish these tasks. Experienced programmers have developed a sense of "what to look for" in the given data in the presence of performance problems. Tuning programs requires dealing with numerous individual instances of code segments. Categorizing these variants and finding the right remedies also demand sufficient experience from programmers. For inexperienced programmers there are few choices other than empirically acquiring knowledge through trials and

* This work was supported in part by NSF grants #9703180-CCR, #9872516-EIA, and #9975275-EIA. This work is not necessarily representative of the positions or policies of the U. S. Government

S.P. Midkiff et al. (Eds.): LCPC 2000, LNCS 2017, pp. 274–288, 2001.

errors. Even learning parallel programming skills from experienced peers takes time and effort. No tools exists that help transfer knowledge from experienced to inexperienced programmers.

We believe that tools can be of considerable use in addressing these problems. We have developed a framework for an automatic performance advisor, called MERLIN, that allows performance evaluation experience to be shared with others. It analyzes program and performance characterization data and presents users with interpretations and suggestions for performance improvement. MERLIN is based on a database utility and an expression evaluator implemented previously [1]. With MERLIN, experienced programmers can guide inexperienced programmers in handling individual analysis and tuning scenarios. The behavior of MERLIN is controlled by a knowledge-based database called a *performance map*. Using this framework, we have implemented several performance maps reflecting our experiences with parallel programs.

The contribution of this paper is to provide a mechanism and a tool that can assist programmers in parallel program performance tuning. Related work is presented in Section 2. Section 3 describes MERLIN in detail. In Section 4 and 5, we present two case studies that utilize the tool. First, we present a simple technique to improve performance using an application of MERLIN for the automatic analysis of timing and static program analysis data. Next, we apply this system to the more advanced performance analysis of data gathered from hardware counters. Section 6 concludes the paper.

2 Related Work

Tools provide support for many steps in a parallelization and performance tuning scenario. Among the supporting tools are those that perform automatic parallelization, performance visualization, instrumentation, and debugging. Many of the current tools are summarized in [2, 3]. Performance visualization has been the subject of many previous efforts [4, 5, 6, 7, 8, 9], providing a wide variety of perspectives on many aspects of the program behavior. The natural next step in supporting the performance evaluation process is to automatically analyze the data and actively advise programmers. Providing such support has been attempted by only few researchers.

The terms "performance guidance" or "performance advisor" are used in many different contexts. Here, we use them to refer to taking a more active role in helping programmers overcome the obstacles in optimizing programs through an automated guidance system. In this section, we discuss several tools that support this functionality.

The SUIF Explorer's Parallelization Guru bases its analysis on two metrics: parallelism coverage and parallelism granularity [10]. These metrics are computed and updated when programmers make changes to a program and run it. It sorts profile data in a decreasing order to bring programmers' attention to most time-consuming sections of the program. It is also capable of analyz-

ing data dependence information and highlighting the sections that need to be examined by the programmers.

The Paradyn Performance Consultant [6] discovers performance problems by searching through the space defined by its own search model (named W^3 space). The search process is fully automatic, but manual refinements to direct the search are possible as well. The result is presented to the users through a graphical display.

PPA [11] proposes a different approach in tuning message passing programs. Unlike the Parallelization Guru and the Performance Consultant, which base their analysis on runtime data and traces, PPA analyzes program source and uses a deductive framework to derive the algorithm concept from the program structure. Compared to other programming tools, the suggestions provided by PPA are more detailed and assertive. The solution, for example, may provide an alternative algorithm for the code section under inspection.

The Parallelization Guru and the Performance Consultant basically tell the user where the problem is, whereas the expert system in PPA takes the role of a programming tool a step further toward an active guidance system. However, the knowledge base for PPA's expert system relies on an understanding of the underlying algorithm based on pattern matching, and the tool works only for a narrow set of algorithms.

Our approach is different from the others, in that it is based on a flexible system controlled by a performance map, which any experienced programmer can write. An experienced user states relationships between common performance problems, characterization data signatures that may indicate sources of the problem, and possible solutions related to these signatures. The performance map may contain complex calculations and evaluations and therefore can act flexibly as either or both of a performance advisor and an analyzer. In order to select appropriate data items and reason about them, a pattern matching module and an expression evaluation utility are provided by the system. Experienced programmers can use this system to help inexperienced programmers in many different aspects of parallel programming. In this way, the tool facilitates an efficient transfer of knowledge to less experienced programmers. For example, if a programmer encounters a loop that does not perform well, they may activate a performance advisor to see the expert's suggestions on such phenomena. Our system does not stop at pointing to problematic code segments. It presents users with possible causes and solutions.

3 Merlin: Performance Advisor

Merlin is a graphical user interface utility that allows users to perform automated analysis of program characterization and performance data. This data can include dynamic information such as loop timing statistics and hardware performance statistics. It can also include compiler-generated data such as control flow graphs and listings of statically applied techniques using the Polaris parallelizing compiler [12]. It can be invoked from the Ursa Minor performance evaluation

tool as shown in Figure 1 [1]. The activation of MERLIN is as simple as clicking a mouse on a problematic program section from this tool.

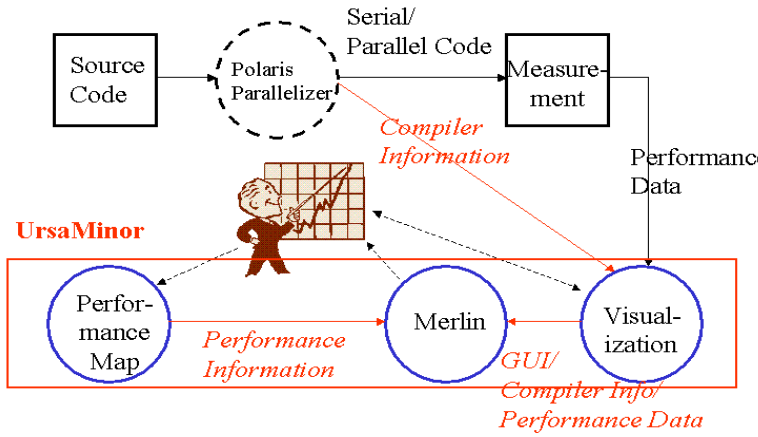


Fig. 1. Interface between related tools. The MERLIN performance advisor plays a key role in the URSA MINOR tool. It uses static and dynamic information collected by the URSA MINOR tool, and suggests possible solutions to performance problems.

Through compilation, simulation, and execution, a user gathers various types of data about a target program. Upon activation, MERLIN performs various analysis techniques on the data at hand and presents its conclusions to the user. Figure 2 shows an instance of the MERLIN interface. It consists of an analysis text area, an advice text area, and buttons. The analysis text area displays the diagnostics MERLIN has performed on the selected program unit. The advice text area provides MERLIN's solution to the detected problems with examples. Diagnosis and the corresponding advice are paired by a number (such as `Analysis 1-2`, `Solution 1-2`).

MERLIN navigates through a database that contains knowledge on diagnosis and solutions for cases where certain performance goals are not achieved. Experienced programmers write performance maps based on their knowledge, and inexperienced programmers can view their suggestions by activating MERLIN. Figure 3 shows the structure of a typical map used by this framework. It consists of three "domains." The elements in the *Problem Domain* corresponds to general performance problems from the viewpoint of programmers. They can represent a poor speedup, a large number of stalls, etc., depending upon the performance data types targeted by the performance map writer. The *Diagnostics Domain* depicts possible causes of these problems, such as floating point dependencies, data cache overflows, etc. Finally, the *Solution Domain* contains possible remedies. These elements are linked by *Conditions*. Conditions are logical expressions representing an analysis of the data. If a condition evaluates to

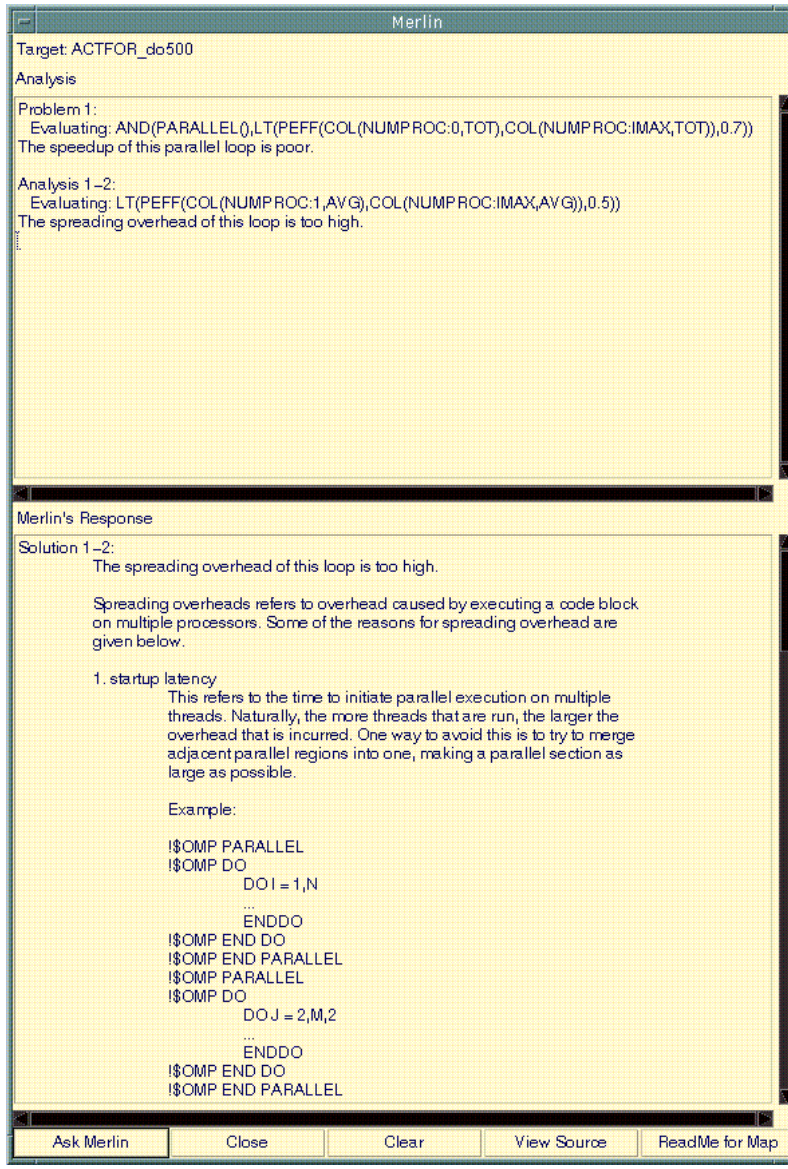


Fig. 2. The user interface of MERLIN in use. MERLIN suggests possible solutions to the detected problems. This example shows the problems addressed in loop `ACTFOR DO500` of program `BDNA`. The button labeled `Ask Merlin` activates the analysis. The `View Source` button opens the source viewer for the selected code section. The `ReadMe for Map` button pulls up the ReadMe text provided by the performance map writer.

true, the corresponding link is taken, and the element in the next domain pointed to by the link is explored. MERLIN invokes an expression evaluator utility for the evaluation of these expressions. When it reaches the Solutions domain, the suggestions given by the map writer are displayed and MERLIN moves on to the next element in the Problem domain.

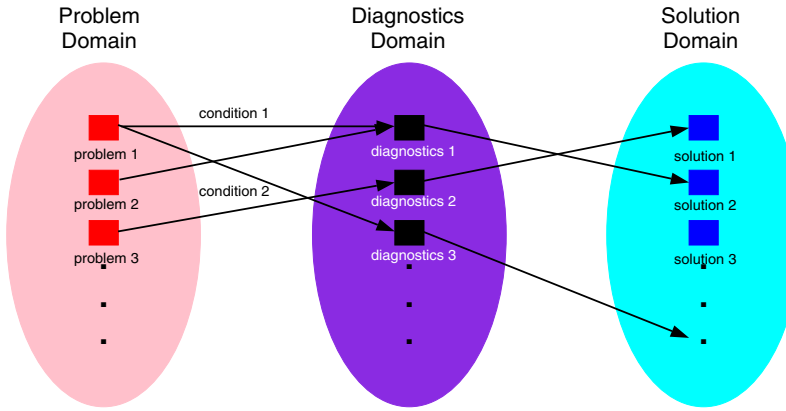


Fig. 3. The internal structure of a typical MERLIN performance map. The Problem Domain corresponds to general performance problems. The Diagnostics Domain depicts possible causes of the problems, and the Solution Domain contains suggested remedies. Conditions are logical expressions representing an analysis of the data.

For example, in the default map of the URSA MINOR tool, one element in the Problem Domain is entitled "poor speedup." The condition for this element is "the loop is parallel and the parallel efficiency is less than 0.7." The link for this condition leads to an element in the Diagnostics Domain labeled "poor speedup symptoms" with conditions that evaluate the parallelization and spreading overheads. When these values are too high, the corresponding links from these conditions points to suggestions for program tuning steps, such as serialization, fusion, interchange, and padding. The data items needed to compute this expression are fetched from URSA MINOR's internal database using the pattern matching utility. If needed data are missing, (e.g., because the user has not yet generated hardware counter profiles,) MERLIN displays a message and continues with the next element. The performance map is written in URSA MINOR's generic input text format [1]. It is structured text of data descriptions that can be easily edited, so map writers can use any text editor. MERLIN reads this file and stores it internally. When a user chooses a loop for automatic analysis, MERLIN begins by analyzing the conditions in the Problem domain.

MERLIN differs from conventional spreadsheet macros in that it is capable of comprehending static analysis data generated by a parallelizing compiler. MERLIN can take into account numeric performance data as well as program

characterization data, such as parallel loops detected by the compiler, the existence of I/O statements, or the presence of function calls within a code block. This allows a comprehensive analysis based on both performance and static data available for the code section under consideration.

A MERLIN map enables efficient cause-effect analyses of performance and static data. It fetches the data specified by the map from the URSA MINOR tool, performs the listed operations and follows the links if the conditions are true. There are no restrictions on the number of elements and conditions within each domain, and each link is followed independently. Hence, multiple perspectives can be easily incorporated into one map. For instance, stalls may be caused by poor locality, but it could also mean a floating point dependence in the pipeline CPU. In this way, MERLIN considers all possible causes for performance problems separately and presents an inclusive set of solutions to its users. At the same time, the remedies suggested by MERLIN assist users in "learning by example." MERLIN enables users to gain expertise in an efficient manner by listing performance data analysis steps and many example solutions given by experienced programmers.

MERLIN is able to work with any performance map as long as it is in the proper format. Therefore, the intended focus of performance evaluation may shift depending on the interest of the user group. For instance, the default map that comes with MERLIN focuses on a performance model based on parallelization and spreading overhead. Should a map that focuses on architecture be developed and used instead, the response of MERLIN will reflect that intention. Furthermore, the URSA MINOR environment does not limit its usage to parallel programming. Coupled with MERLIN, it can be used to address many topics in optimization processes of various engineering practices.

MERLIN is accessed through the URSA MINOR performance evaluation tool [1]. The main goal of URSA MINOR is to optimize program performance through the interactive integration of performance evaluation with static program analysis information. It collects and combines information from various sources, and its graphical interface provides selective views and combinations of the gathered data. URSA MINOR consists of a database utility, a visualization system for both performance data and program structure, a source searching and viewing tool, and a file management module. URSA MINOR also provides users with powerful utilities for manipulating and restructuring the input data to serve as the basis for the users' deductive reasoning. URSA MINOR can present to the user and reason about many different types of data (e.g., compilation results, timing profiles, hardware counter information), making it widely applicable to different kinds of program optimization scenarios. The ability to invoke MERLIN greatly enhances the functionality of URSA MINOR.

4 Case Study 1: Simple Techniques to Improve Performance

In this section, we present a performance map based solely on execution timings and static compiler information. Such a map requires program characterization data that an inexperienced user can easily obtain. The map is designed to advise programmers in improving the performance of programs achieved by a parallelizing compiler such as Polaris [12]. Parallelizing compilers significantly simplify the task of parallel optimization, but they lack knowledge of the dynamic program behavior and have limited analysis capabilities. These limitations may lead to marginal performance gains. Therefore, good performance from a parallel application is often achieved by a substantial amount of manual tuning. In this case study, we assume that programmers have used a parallelizing compiler as the first step in optimizing the performance of the target program. We also assume that the compiler's program analysis results are available. The performance map presented in this section aims at increasing this initial performance.

4.1 Performance Map Description

Based on our experiences with parallel programs, we have chosen four programming techniques that are (1) easy to apply and (2) may yield considerable performance gain. These techniques are serialization, loop interchange, and loop fusion. They are applicable to loops, which are often the focus of the shared memory programming model. All of these techniques are present in modern compilers. However, compilers may not have enough knowledge to apply them most profitably [13], and some code sections may need small modifications before the techniques become applicable automatically.

We have devised criteria for the application of these techniques, which are shown in Table 1. If the speedup of a parallel loop is less than 1, we assume that the loop is too small for parallelization or that it incurred excessive transformations. Serializing it prevents performance degradation. Loop interchange may be used to improve locality by increasing the number of stride-1 accesses in a loop nest. Loop interchange is commonly applied by optimizers; however, our case study shows many examples of opportunities missed by the backend compiler. Loop fusion can likewise be used to increase both granularity and locality. The criteria shown in Table 1 represent simple heuristics and do not attempt to be an exact analysis of the benefits of each technique. We simply chose a speedup threshold of 2.5 to apply loop fusion.

4.2 Experiment

We have applied the techniques shown in Table 1 based on the described criteria. We performed our measurements on a Sun Enterprise 4000 with six 248 MHz UltraSPARC V9 processors, each with a 16KB L1 data cache and 1MB unified L2 cache. Each code variant was compiled by the Sun v5.0 Fortran 77 compiler with the flags `-xtarget=ultra2 -xcache=16/32/1:1024/64/1 -O5`.

Table 1. Optimization technique application criteria.

Techniques	Criteria
Serialization	speedup < 1
Loop Interchange	# of stride-1 accesses $<$ # of non stride-1 accesses
Loop Fusion	speedup < 2.5

The OpenMP code is generated by the Polaris OpenMP backend. The results of five programs are shown. They are `SWIM` and `HYDRO2D` from the SPEC95 benchmark suite, `SWIM` from SPEC2000, and `ARC2D` and `MDG` from the Perfect Benchmarks. We have incrementally applied these techniques starting with serialization. Figure 4 shows the speedup achieved by the techniques. The improvement in execution time ranges from -1.8% for fusion in `ARC2D` to 38.7% for loop interchange in `SWIM'2000`. For `HYDRO2D`, application of the MERLIN suggestions did not noticeably improve performance.

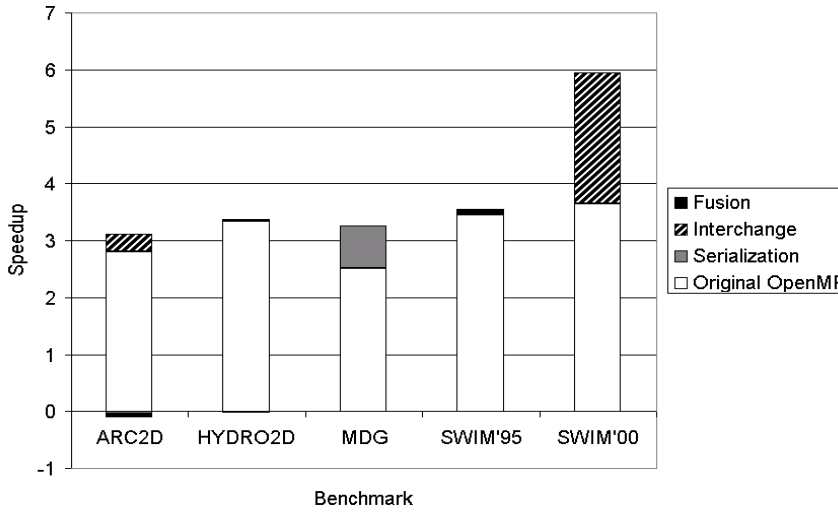


Fig. 4. Speedup achieved by applying the performance map of Table 1. Detailed numbers can be seen in Table 2. The speedup is with respect to one-processor run with serial code on a Sun Enterprise 4000 system. Each graph shows the cumulative speedup when applying each technique. The original OpenMP program was generated by the Polaris parallelizer.

Among the codes with large improvement, `SWIM` from SPEC2000 benefits most from loop interchange. It was applied under the suggestion of MERLIN to the most time-consuming loop, `SHALOW DO3500`. Likewise, the main technique

that improved the performance in ARC2D was loop interchange. MDG consists of two large loops and numerous small loops. Serializing these small loops was the sole reason for the performance gain. Table 2 shows a detailed breakdown of how often techniques were applied and their corresponding benefit.

Table 2. A detailed breakdown of the performance improvement due to each technique.

Benchmark	Technique	Number of Modifications	% Improvement
ARC2D	Serialization	3	-1.55
	Interchange	14	9.77
	Fusion	10	-1.79
HYDRO2D	Serialization	18	-0.65
	Interchange	0	0.00
	Fusion	2	0.97
MDG	Serialization	11	22.97
	Interchange	0	0.00
	Fusion	0	0.00
SWIM'95	Serialization	1	0.92
	Interchange	0	0.00
	Fusion	3	2.03
SWIM'00	Serialization	0	0.00
	Interchange	1	38.69
	Fusion	1	0.03

Using this map, considerable speedups are achieved with relatively small effort. Inexperienced programmers can simply run Merlin to see the suggestions made by the map. The map can be updated flexibly without modifying Merlin. Thus if new techniques show potential or the criteria need revision, programmers can easily incorporate changes.

5 Case Study 2: Hardware Counter Data Analysis

In our second case study, we discuss a more advanced performance map that uses the *speedup component model* introduced in [14]. The model fully accounts for the gap between the measured speedup and the ideal speedup in each parallel program section. This model assumes execution on a shared-memory multiprocessor and requires that each parallel section be fully characterized using hardware performance monitors to gather detailed processor statistics. Hardware monitors are now available on most commodity processors.

With hardware counter and timer data loaded into Ursa Minor, users can simply click on a loop from the Ursa Minor table view and activate Merlin. Merlin then lists the numbers corresponding to the various overhead components responsible for the speedup loss in each code section. The displayed values

for the components show overhead categories in a form that allows users to easily see why a parallel region does not exhibit the ideal speedup of p on p processors. MERLIN then identifies the dominant components in the loops under inspection and suggests techniques that may reduce these overheads. An overview of the speedup component model and its implementation as a MERLIN map are given below.

5.1 Performance Map Description

The objective of our performance map is to be able to fully account for the performance losses incurred by each parallel program section on a shared-memory multiprocessor system. We categorize overhead factors into four main components. Table 3 shows the categories and their contributing factors.

Table 3. Overhead categories of the speedup component model.

Overhead Category	Contributing Factors	Description	Measured with
Memory stalls	IC miss	Stall due to I-Cache miss.	HW Cntr
	Write stall	The store buffer cannot hold additional stores.	HW Cntr
	Read stall	An instruction in the execute stage depends on an earlier load that is not yet completed.	HW Cntr
	RAW load stall	A read needs to wait for a previously issued write to the same address.	HW Cntr
Processor stalls	Mispred. Stall	Stall caused by branch misprediction and recovery.	HW Cntr
	Float Dep. stall	An instruction needs to wait for the result of a floating point operation.	HW Cntr
Code overhead	Parallelization	Added code necessary for generating parallel code.	computed
	Code generation	More conservative compiler optimizations for parallel code.	computed
Thread management	Fork&join	Latencies due to creating and terminating parallel sections.	timers
	Load imbalance	Wait time at join points due to uneven workload distribution.	

Memory stalls reflect latencies incurred due to cache misses, memory access times and network congestion. MERLIN will calculate the cycles lost due to these overheads. If the percentage of time lost is large, locality-enhancing software techniques will be suggested. These techniques include optimizations such as

loop interchange, loop tiling, and loop unrolling. We found, in [13], that loop interchange and loop unrolling are among the most important techniques.

Processor stalls account for delays incurred processor-internally. These include branch mispredictions and floating point dependence stalls. Although it is difficult to address these stalls directly at the source level, loop unrolling and loop fusion, if properly applied, can remove branches and give more freedom to the backend compiler to schedule instructions. Therefore, if processor stalls are a dominant factor in a loop's performance, MERLIN will suggest that these two techniques be considered.

Code overhead corresponds to the time taken by instructions not found in the original serial code. A positive code overhead means that the total number of cycles, excluding stalls, that are consumed across all processors executing the parallel code is larger than the number used by a single processor executing the equivalent serial section. These added instructions may have been introduced when parallelizing the program (e.g., by substituting an induction variable) or through a more conservative parallel code generating compiler. If code overhead causes performance to degrade below the performance of the original code, MERLIN will suggest serializing the code section.

Thread management accounts for latencies incurred at the fork and join points of each parallel section. It includes the times for creating or notifying waiting threads, for passing parameters to them, and for executing barrier operations. It also includes the idle times spent waiting at barriers, which are due to unbalanced thread workloads. We measure these latencies directly through timers before and after each fork and each join point. Thread management latencies can be reduced through highly-optimized runtime libraries and through improved balancing schemes of threads with uneven workloads. MERLIN will suggest improved load balancing if this component is large.

Ursa Minor combined with this MERLIN map displays (1) the measured performance of the parallel code relative to the serial version, (2) the execution overheads of the serial code in terms of stall cycles reported by the hardware monitor, and (3) the speedup component model for the parallel code. We will discuss details of the analysis where necessary to explain effects. However, for the full analysis with detailed overhead factors and a larger set of programs we refer the reader to [14].

5.2 Experiment

For our experiment we translated the original source into OpenMP parallel form using the Polaris parallelizing compiler [12]. The source program is the Perfect Benchmark `ARC2D`, which is parallelized to a high degree by Polaris. We have used the same machine as in Section 4. For hardware performance measurements, we used the available hardware counter (`TICK` register) [15].

`ARC2D` consists of many small loops, each of which has a few milli-seconds average execution time. Figure 5 shows the overheads in the loop `STEPFX DO230` of the original code, and the speedup component graphs generated before and after applying a loop interchange transformation.

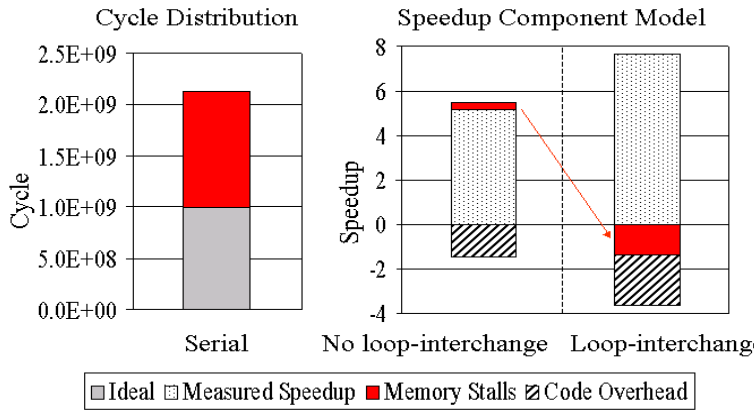


Fig. 5. Performance analysis of the loop `STEPFX DO230` in program `ARC2D`. The graph on the left shows the overhead components in the original, serial code. The graphs on the right show the speedup component model for the parallel code variants on 4 processors before and after loop interchanging is applied. Each component of this model represents the change in the respective overhead category relative to the serial program. The sum of all components is equal to the ideal speedup (=4 in our case). MERLIN is able to generate the information shown in these graphs.

MERLIN calculates the speedup component model using the data collected by a hardware counter, and displays the speedup component graph. MERLIN applies the following map using the speedup component model: *If the memory stall appears both in the performance graph of the serial code and in the speedup component model for the Polaris-parallelized code, then apply loop interchange.* From this suggested recipes the user tries loop interchanging, which results in significant, now superlinear speedup. Figure 5 "loop-interchange" on the right shows that the memory stall component has become negative, which means that there are fewer stalls than in the original, serial program. The negative component explains why there is superlinear speedup. The speedup component model further shows that the code overhead component has drastically decreased from the original parallelized program. The code is even more efficient than in the serial program, further contributing to the superlinear speedup.

In this example, the use of the performance map for the speedup component model has significantly reduced the time spent by a user analyzing the performance of the parallel program. It has helped explain both the sources of overheads and the sources of superlinear speedup behavior.

6 Conclusions

We have presented a framework and a tool, MERLIN, that addresses an important open issue in parallel programming: Guiding inexperienced programmers in the

process of tuning parallel program performance. It is a utility with a graphical user interface that allows programmers to examine suggestions made by expert programmers on various performance issues. While equipped with a powerful expression evaluator and pattern matching utility, MERLIN's analysis and advice are entirely guided by a performance map. Any experienced programmer can write a performance map to help new programmers in performance tuning. MERLIN is accessed through a performance evaluation tool, so performance visualization and data gathering are done in conjunction with this performance advisor.

We have presented two case studies that utilize MERLIN. In the first study, we have introduced a simplified performance map that can still effectively guide users in improving the performance of real applications. In the second study, MERLIN is used to compute various overhead components for investigating performance problems. The results show that the relatively small effort to run MERLIN can lead to significant speedup gains and insight into the performance behavior of a program.

With the increasing number of new users of parallel machines, the lack of experience and transfer of programming knowledge from advanced to novice users is becoming an important issue. A novel aspect of our system is that it alleviates these problems through automated analysis and interactive guidance. With advances in performance modeling and evaluation techniques as exemplified in this paper, parallel computing can be made amenable to an increasingly large community of users.

References

[1] Insung Park, Michael J. Voss, Brian Armstrong, and Rudolf Eigenmann. Parallel programming and performance evaluation with the URSA tool family. *International Journal of Parallel Programming*, 26(5):541–561, November 1998.

[2] J. Brown, A. Geist, C. Pancake, and D. Rover. Software tools for developing parallel applications. 1. code development and debugging. In *Proc. of Eighth SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.

[3] J. Brown, A. Geist, C. Pancake, and D. Rover. Software tools for developing parallel applications. 2. interactive control and performance tuning. In *Proc. of Eighth SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.

[4] Michael T. Heath. Performance visualization with ParaGraph. In *Proc. of the Second Workshop on Environments and Tools for Parallel Scientific Computing*, pages 221–230, May 1994.

[5] Daniel A. Reed. Experimental performance analysis of parallel systems: Techniques and open problems. In *Proc. of the 7th Int' Conf on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 25–51, 1994.

[6] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, November 1995.

[7] J. Yan, S. Sarukkai, and P. Mehra. Performance measurement, visualization and modeling of parallel and distributed programs using the AIMS toolkit. *Software-Practice and Experience*, 25(4):429–461, April 1995.

[8] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach. VAMPIR: visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, January 1996.

[9] B. Topol, J. T. Stasko, and V. Sunderam. PVaniM: A tool for visualization in network computing environments. *Concurrency Practice and Experience*, 10(14):1197–1222, December 1998.

[10] W. Liao, A. Diwan, R. P. Bosch Jr., A. Ghuloum, and M. S. Lam. SUIF explorer: An interactive and interprocedural parallelizer. In *Proc. of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 37–48, August 1999.

[11] K. C. Li and K. Zhang. Tuning parallel program through automatic program analysis. In *Proc. of Second International Symposium on Parallel Architectures, Algorithms, and Networks*, pages 330–333, June 1996.

[12] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with Polaris. *IEEE Computer*, pages 78–82, December 1996.

[13] Seon Wook Kim, Michael Voss, and Rudolf Eigenmann. Performance analysis of parallel compiler backends on shared-memory multiprocessors. *Proceedings of Compilers for Parallel Computers (CPC2000)*, pages 305–320, January 2000.

[14] Seon Wook Kim and Rudolf Eigenmann. Detailed, quantitative analysis of shared-memory parallel programs. Technical Report ECE-HPCLab-00204, HPCLAB, School of ECE, Purdue University, 2000.

[15] David L. Weaver and Tom Germond. *The SPARC Architecture Manual, Version 9.* SPARC International, Inc., PTR Prentice Hall, Englewood Cliffs, NJ 07632, 1994.

A Comparative Analysis of Dependence Testing Mechanisms*

Jay Hoeflinger[1] and Yunheung Paek[2]

[1] Center for Simulation of Advanced Rockets, University of Illinois at Urbana-Champaign
[2] Department of Electrical Engineering & Computer Science, Korea Advanced Institute of Science & Technology

Abstract. The internal mechanism used for a dependence test constrains its accuracy and determines its speed. So does the form in which it represents array subscript expressions. The internal mechanism and representational form used for our Access Region Test (ART) is different from that used in any other dependence test, and therefore its constraints and characteristics are likewise different. In this paper, we briefly describe our descriptor for representing memory accesses, the Linear Memory Access Descriptor (LMAD) and the ART. We then describe the LMAD intersection algorithm in some detail. Finally, we compare and contrast the mechanisms of the LMAD intersection algorithm with the internal mechanisms for a number of prior dependence tests.

1 Introduction

Dependence analysis has traditionally been cast as an equation-solving activity. The *dependence problem* was posed originally as a problem in *constrained Diophantine equations*. That is, the subscript expressions for two array references within nested loops were equated, constraints derived from the program text were added, and an attempt was made to solve the resulting system for integer values of the loop indices. Solving this problem is equivalent to *integer programming*, which is known to be NP-complete [4, 7], so many heuristics have been proposed for the problem over the last 25 years. We refer to this regime as *point-to-point dependence analysis*.

More recently, principally due to the introduction of the Omega Test [19], which made Fourier Elimination practical, dependence testing has been cast in terms of the solution of a linear system [1, 5, 20]. A linear system can represent multiple memory locations as easily as it can a single memory location, so people began using linear systems to summarize the memory accesses in whole loops, then intersected the summaries to determine whether there was a dependence. We refer to this as *summary-based dependence analysis*.

Linear systems also became the basis for interprocedural dependence testing [9, 5, 20], through the translation of the linear systems from one procedural

* This work was supported in part by the US Department of Energy through the University of California under Subcontract number B341494.

S.P. Midkiff et al. (Eds.): LCPC 2000, LNCS 2017, pp. 289–303, 2001.

context to another. When array reshaping occurs across the procedure boundary, this translation can lead to added complexity within the linear system.

The memory locations accessed by a subscripted array reference have been represented to a dependence test by several forms. Triplet notation (*begin : end : stride*) and various derivatives of this form have been used for representing memory accesses [2, 10, 21], as have sets of linear constraints. As indicated in [12, 18], these forms are limited in the types of memory accesses that they can represent exactly. Sets of linear constraints[6, 20, 9] are a richer representational mechanism, able to represent more memory reference patterns exactly than can triplet notation. Still, linear constraints cannot represent the access patterns of non-linear expressions. The Range Test is unique among dependence tests, in that it does not use a special representational form for its subscript expressions. It uses triplet notation for value ranges, but relies on the original subscript expressions themselves within its testing mechanism, so it loses no accuracy due to representational limitations.

These techniques have been relatively successful within their domain of applicability, but are limited by the restrictions that the representation and the equation-solving activity itself imposes. Almost all techniques are restricted to subscript expressions that are linear with respect to the loop indices. Most tests require constant coefficients and loop bounds. However, as noted in [15], common situations produce non-linear subscript expressions, such as programmer-linearized arrays, the closed form for an induction variable, and aliasing between arrays with different numbers of dimensions. The Range Test[3] is the only test that can do actual dependence testing with arbitrary non-linear subscript expressions.

We have developed a new representational form, the Linear Memory Access Descriptor (LMAD) [17, 18], which can precisely represent nearly all the memory access patterns found in real Fortran programs. We have also developed an interprocedural, summary-based dependence testing framework [12], the Access Region Test (ART), that uses this representation. At the core of the dependence testing framework is an intersection algorithm that determines the memory locations accessed in common between two LMADs. This is the equivalent of the equation-solving mechanisms at the core of traditional dependence tests.

The LMAD intersection algorithm has restrictions similar to those of other dependence mechanisms. It cannot intersect LMADs produced by non-affine subscript expressions, or those produced within triangular loops. Its strengths come from the fact that it is an exact test, it can produce distance vectors, and that it can be used precisely, interprocedurally. Simplification operators have been defined for the LMAD, sometimes allowing non-affine LMADs to be simplified to affine ones, compensating somewhat for the intersection algorithm's limitations.

In previous papers, we focused on an in-depth description of the representational form of the LMAD and the dependence testing framework based on the form. In this paper, we describe the intersection algorithm used in our dependence testing, and discuss how it compares with the internal mechanisms of

other dependence testing techniques. In Section 2, we will cover brief summaries of the representational form and the dependence analysis framework, and describe the intersection algorithm in the framework. Then, in Sections 3 and 4, we will consider other dependence analysis techniques and compare their core mechanisms with ours. Finally, we will summarize our discussion in Section 5.

2 A Dependence Analysis Framework Using Access Descriptors

In this section, we present an overview of the Access Region Test (ART). The details of the ART can found in our previous reports [11, 12].

2.1 Representing Array Accesses within Loops

```
// A declared with m dimensions
for I1 = 0 to U1 {
   for I2 = 0 to U2 {
      ...
         for Id = 0 to Ud {
            ··· A(s1(I), s2(I), ···, sm(I)) ···
         }
      ...
   }
}
```

Fig. 1. General form of a reference to array A in a d-nested loop. The notation $\boldsymbol{I}$ represents the vector of loop indices: $(I_1, I_2, \cdots, I_d)$.

Without loss of generality, in all that follows, we will assume that all loop indices are normalized.

The *memory space* of a program is the set of memory locations which make up all the memory usable by a program. When an m-dimensional array is allocated in memory, it is linearized and usually laid out in either row-major or column-major order, depending on the language being used. In order to map the array space to the memory space of the program, the subscripting function must be mapped to a single integer that is the offset from the beginning of the array for the access. We define this *subscript mapping* F_a for an array reference with a *subscripting function* s, as in Figure 1 by

$$F_a(s_1, s_2, \cdots, s_m) = \sum_{k=1}^{m} s_k \cdot \lambda_k.$$

When the language allocates an array in column-major order, $\lambda_1 = 1$ and $\lambda_k = \lambda_{k-1} \cdot n_{k-1}$ for $k \neq 1$. If the language allocates the array in row-major order,

$\lambda_m = 1$ and $\lambda_k = \lambda_{k+1} \cdot n_{k+1}$ for $k \neq m$. After applying F_a to the subscripting expressions s_k in Figure 1, we would have the *linearized form*:

$$F_a(\mathbf{s}(\boldsymbol{I})) = s_1(\boldsymbol{I})\lambda_1 + s_2(\boldsymbol{I})\lambda_2 + \cdots + s_m(\boldsymbol{I})\lambda_m. \tag{1}$$

As the nested loop in Figure 1 executes, the subscript mapping function F_a generates a sequence of offsets from the base address of `A`. We call this sequence the *subscripting offset sequence*:

$$F_a(s)|_{I_d, I_{d-1}, \cdots, I_1} = S_1, S_2, \cdots, S_n{}^{\star\star}$$

If the linearized form of the subscript expression s can be written in a sum-of-products form with respect to the individual loop indices,

$$F_a(\mathbf{s}(\boldsymbol{I})) = f_0 + f_1(I_1) + f_2(I_2) + \cdots + f_m(I_m). \tag{2}$$

then, we can isolate the effect of each loop index on the subscripting offset sequence. In this case, there is no restriction on the form of the functions f_k. They can be subscripted-subscripts, or any non-affine function.

We define the isolated effect of any loop in a loop nest on a memory reference pattern to be a *dimension* of the access. A dimension k can be characterized by its *stride*, δ_k, and the number of iterations in the loop, $U_k + 1$. An additional expression, the *span*, is carried with each dimension since it is used in some operations. The *stride* and *span* associated with a given loop index I_k are defined

$$\delta_k = f_k[I_k \leftarrow I_k + 1] - f_k \qquad \text{(stride)} \tag{3}$$

$$\sigma_k = f_k[I_k \leftarrow U_k] - f_k[I_k \leftarrow 0] \qquad \text{(span)} \tag{4}$$

where the notation $f[i \leftarrow k]$ means to replace every occurrence of i in expression f by k.

An LMAD is a representation of the subscripting offset sequence. *It can be built for any array reference whose subscript expressions can be put in the form of Equation 2, so all algorithms in this paper will assume this condition has been met for all LMADs.* It contains all the information necessary to generate the subscripting offset sequence. Each loop index in the program is *normalized* internally for purposes of the representation, and called the *dimension index*.

The LMAD contains:

- a starting value, called the *base offset*, represented as τ, and
- for each dimension k:
  - a dimension index I_k, taking on all integer values between 0 and U_k,
  - a stride expression, δ_k
  - a span expression, σ_k.

The span is useful for doing certain operations and simplifications on the LMAD (for instance detecting *internal overlap*, as will be described in Section 2.3), however it is only accurate when the dimension is monotonic and is not required for accuracy of the representation. The general form for an LMAD is written as

$$\mathcal{A}^{\delta_1, \delta_2, \cdots, \delta_d}_{\sigma_1, \sigma_2, \cdots, \sigma_d} + \tau.$$

```
      real A(0:N-1,0:N-1)
      do I = 0, 9
         do J = 0, 2
T:          A(J+5,2*I) = B(J,C(I))
```

Fig. 2. Example of references to array `A` in statement T within a 2-nested loop.

Examples of the LMAD form for `A` and `B` in statement T of Figure 2 are as follows:

$$\mathrm{LMAD}_T(\mathrm{A}) = \mathcal{A}\ {}^{1,2N}_{2,18N} + 5 \tag{5}$$

$$\mathrm{LMAD}_T(\mathrm{B}) = \mathcal{B}\ {}^{1,C(I+1)-C(I)}_{2,C(9)-C(0)} + 5 \tag{6}$$

Note that an LMAD dimension whose span is zero adds no data elements to an access pattern in the LMAD (we call this the *zero-span property* ⋆⋆⋆). The memory accesses represented by an LMAD are not changed when a dimension with a span of 0 is added. Also, note that insertion of a zero-span dimension can not change any width of an LMAD, and likewise, will not change the perfectly-nested property. Therefore, it is always possible to add a dimension with any desired stride to an LMAD, by making the associated span 0. So, given any two LMADs, with any dimensions, it is always possible to make them have dimensions with matching strides by adding zero-span dimensions to each.

The following LMAD characteristics will be used later in the paper:

Definition 1. *Given an LMAD $\mathcal{D}$, we call the sum of the spans of the first k dimensions the* **k-dimensional width** *of the LMAD, defined by*

$$\mathrm{width}_k(\mathcal{D}) = \sum_{j=1}^{k} \sigma_j .$$

Definition 2. *Given an LMAD $\mathcal{D}$, we say it is* **perfectly nested** *if, for all k,*

$$\delta_k > \mathrm{width}_{k-1}(\mathcal{D})$$

2.2 Building Access Descriptors

An LMAD representing the memory accesses in a nested loop, such as the one in Figure 1, is built starting from the inner-most loop of a loop nest out to the outer-most loop. The LMAD is first built to represent the scalar access made by the statement itself. As an example of that, notice that statement `T` by itself, in Figure 2, refers to a single memory location within `A`, namely `A(J+5,2*I)`, using the current values of `I` and `J`.

⋆⋆ This notation means that the loop indices take on values within F_a in normal loop-nest order.

⋆⋆⋆ For a proof of this, see [11, 17]

Then, the process moves outward in the loop nest, *expanding* the LMAD successively by the loop index of each surrounding loop. The expansion operation is detailed in Figure 3. The dimensions of the previous descriptor are copied intact to the new descriptor. A single new dimension and a new base offset expression are formed, using the original descriptor's base offset expression as input. For example, the statement T in Figure 2 initially produces the following LMAD for array `A`:

$$\text{LMAD}_T(\text{A}) = A_0^0 + I * 2 * N + J + 5$$

Expanding $\text{LMAD}_T(\text{A})$ for the `J` loop produces a stride expression of $[J+1] - [J] = 1$ and a new span expression of $[2] - [0] = 2$. The new offset expression becomes $I * 2 * N + 0 + 5 = I * 2 * N + 5$ making the expanded descriptor $\text{LMAD}_T = \mathcal{A}_2^1 + (I * 2 * N) + 5$, which can be further expanded by the outer loop index `I` to produce the LMAD shown in Equation 5. Function `Expand` is accurate for any LMADs, regardless of their complexity.

Input: Original LMAD $\mathcal{H}^{\delta_1,\cdots,\delta_{k-1}}_{\sigma_1,\cdots,\sigma_{k-1}} + \tau$,
loop index i_k with loop lower bound b_k, upper bound e_k, and stride s_k
Output: LMAD expanded by loop index i_k, and new dimension-index i'_k

Function Expand:
Create dimension-index i'_k with $0 \le i'_k \le u_k$, $u_k = \lfloor (e_k - b_k)/s_k \rfloor$
$\tau \leftarrow \tau[i_k \leftarrow i'_k \cdot s_k + b_k]$
$\delta_k \leftarrow \tau[i'_k \leftarrow i'_k + 1] - \tau$;
if (u_k is unknown) {
  $\sigma_k \leftarrow \infty$;
} **else** {
  $\sigma_k \leftarrow \tau[i'_k \leftarrow u_k] - \tau[i'_k \leftarrow 0]$;
}
$\tau_{new} \leftarrow \tau[i'_k \leftarrow 0]$;
Insert new dimension in LMAD
return $\mathcal{H}^{\delta_1,\cdots,\delta_{k-1},\delta_k}_{\sigma_1,\cdots,\sigma_{k-1},\sigma_k} + \tau_{new}$;
end Function Expand

Fig. 3. A function for expanding an LMAD by a loop index.

2.3 Intersecting LMADs

The ART is used within the general framework of *memory classification analysis*. Within memory classification analysis, the entire program is traversed in execution order, using abstract interpretation, with access summaries being computed for each level of each loop nest and stored in LMADs. Whenever loops are encountered, the LMADs for the accesses are intersected to determine whether the loops are parallel. For further details of memory classification analysis, see [11, 12].

The intersection algorithm that is at the heart of the Access Region Test is shown in Figure 4. The precision of the algorithm suffers whenever the *less-than* relationship cannot be established between the various expressions involved in it. For example, for two strides to be sorted, *less-than* must be established between them. Establishing *less-than* depends on both the richness of the symbolic information derivable from the program and the symbolic manipulation capabilities of the compiler.

So, provided that all necessary symbolic less-than relationships can be established, the intersection algorithm is an exact algorithm *for descriptors meeting its input requirements*. By "exact", we mean that whenever there is an intersection, it will return exactly the elements involved, and whenever there is no intersection, it will report that there is no intersection. The exactness of this algorithm makes possible an *exact dependence test*.

The intersection algorithm operates on two LMADs which have the same number of sorted dimensions (d), and the same strides. As mentioned above, given any two LMADs, it is always possible to transform them into matching descriptors because of the zero-span property. In addition each descriptor must have the perfect nesting attribute.

The dimensions on the two descriptors are sorted according to the stride, so that $\delta_{k+1} > \delta_k$, for $1 \leq k \leq d-1$. In addition to the dimensions 1 through d, the algorithm refers to dimension 0, which means a scalar access at the location represented by the base offset. Likewise, of the two descriptors, it is required that one be designated the *left* descriptor (the one with the lesser base offset), while the other is the *right* descriptor.

As shown in Figure 4, the algorithm treats the dimension with the largest stride first. For each dimension, the algorithm first checks whether the extents of the two dimensions overlap. If they do, the algorithm then makes two new descriptors from the original descriptors by stripping off the outermost dimension, and adjusting the base offsets of the two descriptors appropriately. It then recursively calls the intersection algorithm, targeting the next inner-most dimension this time.

The algorithm can compute the *dependence distance* associated with each dimension of the descriptor. If a given dimension of both descriptors comes from the same original loop index and there is a non-empty intersection, then the algorithm can compute the accurate dependence distance for that loop.

After the algorithm has checked each dimension and adjusted the base offsets at each step, it calls itself with a dimension of 0, which compares the adjusted base offsets of the two descriptors. If they are not equal, then the algorithm returns an empty intersection. If the base offsets are equal, then an intersection descriptor with that base offset is returned. As the algorithm returns from each recursive call, a dimension is added to the intersection descriptor.

The intersection of a given dimension can occur at either side (to the right or left) of the right descriptor. If the intersection happens at both sides, then two intersection descriptors will be generated for that dimension. The algorithm thus returns a list of descriptors representing the intersection.

Algorithm Intersect

Input: Two LMADs, with properly nested, sorted dimensions :
$\text{LMAD}_{\text{left}} = \mathcal{A}^{\delta_1,\delta_2,\cdots,\delta_d}_{\sigma_1,\sigma_2,\cdots,\sigma_d} + \tau$, $\text{LMAD}_{\text{right}} = \mathcal{A}^{\delta_1,\delta_2,\cdots,\delta_d}_{\sigma'_1,\sigma'_2,\cdots,\sigma'_d} + \tau'$
$[\tau \leq \tau' \text{ and } \delta_i \leq \delta_{i+1}]$,
The number of the dimension to work on: k $[0 \leq k \leq d]$
Direction to test: $<$ or $>$
Output: Intersection LMAD list and dependence distance
Algorithm:

intersect($\text{LMAD}_{\text{left}}$, $\text{LMAD}_{\text{right}}$, k, DIR) **returns** LMAD_*List*, DIST
  $\text{LMAD\_}List \leftarrow \emptyset$
  $D \leftarrow \tau' - \tau$
  `if` $(k == 0)$ `then` // scalar intersection
    `if` $(D == 0)$ `then`
      $\text{LMAD}_{\text{rlist1}} \leftarrow$ **LMAD_***scalar*(τ)
      add_to_list$(\text{LMAD\_}List, \text{LMAD}_{\text{rlist1}})$
    `endif`
    `return` LMAD_*List*,0
  `endif`
  `if` $(D \leq \sigma_k)$ `then` // periodic intersection on the left
    $R \leftarrow \texttt{mod}(D, \delta_k)$
    $m \leftarrow \lfloor \frac{D}{\delta_k} \rfloor$
$I1$: $\text{LMAD}_{\text{rlist1}} \leftarrow$ **intersect**(**remove_dim**$(\text{LMAD}_{\text{left}}, k, \tau + m\delta_k)$,
      remove_dim$(\text{LMAD}_{\text{right}}, k, \tau'), k-1, <)$
    `if` ($\text{LMAD}_{\text{rlist1}} \neq \emptyset$ AND loop indices match) `then`
      $\text{DIST}_k = (\text{DIR is } < ?\ m : -m)$
    $\text{LMAD}_{\text{rlist1}} \leftarrow$ **add_dim**$(\text{LMAD}_{\text{rlist1}}, \textbf{dim}(\delta_k, \textbf{min}(\sigma_k - m\delta_k, \sigma'_k)))$
    add_to_list$(\text{LMAD\_}List, \text{LMAD}_{\text{rlist1}})$
    `if` ($(k>1)$ **and** $(R + \textbf{width}_{k-1} \geq \delta_k)$) `then` //periodic intersection right
$I2$: $\text{LMAD}_{\text{rlist2}} \leftarrow$ **intersect**(**remove_dim**$(\text{LMAD}_{\text{right}}, k, \tau')$,
      remove_dim$(\text{LMAD}_{\text{left}}, k, \tau + (m+1)\delta_k)$,
      $k-1, >)$
      `if` ($\text{LMAD}_{\text{rlist2}} \neq \emptyset$ AND loop indices match) `then`
        $\text{DIST}_k = (\text{DIR is } < ?\ m+1 : -(m+1))$
      $\text{LMAD}_{\text{rlist2}} \leftarrow$ **add_dim**$(\text{LMAD}_{\text{rlist2}}, \textbf{dim}(\delta_k, \textbf{min}(\sigma_k - (m+1)\delta_k, \sigma'_k)))$
      add_to_list$(\text{LMAD\_List}, \text{LMAD}_{\text{rlist2}})$
    `endif`
  `else` // intersection at the end
$I3$: $\text{LMAD}_{\text{rlist1}} \leftarrow$ **intersect**(**remove_dim**$(\text{LMAD}_{\text{left}}, k, \tau + \sigma_k)$,
      remove_dim$(\text{LMAD}_{\text{right}}, k, \tau'), k-1, <)$
    `if` ($\text{LMAD}_{\text{rlist1}} \neq \emptyset$ AND loop indices match) `then`
      $\text{DIST}_k = (\text{DIR is } < ?\ \sigma_k/\delta_k : -\sigma_k/\delta_k)$
    add_to_list$(\text{LMAD\_}List, \text{LMAD}_{\text{rlist1}})$
  `endif`
  `return` LMAD_*List*, DIST
`end` **intersect**

Fig. 4. Algorithm Intersect for LMADs

Algorithm Intersect Support Routines

```
dim( stride, span ) returns LMAD_Dimension
    Dimension.stride ← stride ; Dimension.span ← span
    return Dimension
end dim

add_dim( LMAD_list, Dimension ) returns LMAD_list
    For each LMAD in LMAD_list
        // Add "Dimension" to the list of dimensions of LMAD
    return LMAD_list
end add_dim

remove_dim( LMAD_arg, k, Offset ) returns LMAD
    LMAD_new ← LMAD_arg
    // Remove dimension k from the list of dimensions of LMAD_new
    // Set the base offset of LMAD_new to "Offset"
    return LMAD_new
end remove_dim
```

Fig. 5. Algorithm Intersect support routines.

The complexity of the LMAD intersection algorithm itself is exponential in the number of dimensions, d, since 2^d LMADs may be produced when 2 LMADs are intersected.

The Access Region Test Memory classification analysis is used within the ART to classify the accesses, caused by a section of code, to a region of memory. A region of memory is represented by an LMAD. When an LMAD is expanded by a loop index (to simulate the effect of executing the loop), cross-iteration dependences are found by one of two means:

1. discovering overlap within a single LMAD (caused by the expansion)
2. a non-empty intersection of two expanded LMADs

The first case is found by checking whether the LMAD is *perfectly nested*, as defined above. If it loses the *perfectly nested* attribute due to expansion by a loop index, then the expansion has caused an internal overlap. That means that the subscripting offset sequence represented by the LMAD contains at least one offset that appears more than once.

The second case is found by using the LMAD intersection algorithm on each pair of LMADs for a given variable. A non-empty intersection indicates that the two LMADs represent at least one identical memory address. Since this is caused by expansion by a loop index, it represents a loop-carried cross-iteration dependence.

A full description of the ART may be found in [11, 12].

3 Comparing the Internal Mechanisms of Dependence Tests

This section will briefly describe several well-known dependence analysis techniques and the internal mechanisms used in them, in preparation for comparing their mechanisms with the LMAD intersection algorithm in Section 4. In our description of these tests, we will consider a loop nest of the form in Figure 1. The basic problem that these tests attempt to solve (the *Dependence Problem*) for two references to an array, $A(s_1(I), s_2(I), \cdots, s_m(I))$ and $A(s'_1(I), s'_2(I), \cdots, s'_m(I))$ is a simultaneous solution of a *System of Equalities* subject to a *System of Constraints*:

$$s_k(\boldsymbol{I}) \quad = \quad s'_k(\boldsymbol{I}) \mid 1 \leq k \leq m \tag{7}$$

$$L_k \leq I_k \leq U_k \mid 1 \leq k \leq m \tag{8}$$

We will describe the characteristics of these tests in terms of eight criteria:

1. the acceptable form of coefficients of loop indices (**coef form** in Table 1),
2. the acceptable form of loop bounds (**loop bnd form**),
3. to what extent the test uses the System of Constraints (**use constr?**),
4. under what conditions (if any) is the test exact (**exact?**),
5. whether the test can produce dependence distance vectors (**dist vec?**),
6. whether the test can solve all dimensions simultaneously? (**simult soln**),
7. the complexity of the mechanism (**complex**), and
8. the test's ability to do interprocedural dependence testing (**inter proc?**).

These characteristics are summarized for all tests in Table 1.

3.1 Basic Techniques

We consider three dependence analysis techniques - GCD, Extreme Value, and Fourier Elimination (FE) - to be basic in that they form a core group which covers most of the other types of techniques that have been used to determine dependence. Descriptions of these can be found in [22].

3.2 Extended Techniques

The basic dependence analysis techniques described above can be too naive or expensive in practice. To cope with the disadvantages, numerous advanced techniques extending the basic ones have been proposed, some of which will be described in this section.

Omega Test The Omega Test [19] uses clever heuristics to speed up the FE algorithm in commonly-occurring cases. The complexity of the Omega Test is typically polynomial for situations occurring in real programs. Algorithms exist for handling array reshaping at `call` sites, although they may result in adding complicated constraints to the system.

Generalized GCD-Based Tests The GCD test can be generalized by writing the System of Equalities in matrix form: $EI = e$, in which E is the matrix formed by the coefficients of the loop indices, I is a vector of loop indices, and e is a vector of the remaining terms, which do not involve loop indices. The system is solved by transforming matrix E to a column echelon form matrix F by a series of unimodular operations. The equivalent equation $Ft = e$ is then solved by back-substitution. If there are no solutions to this equation, then there will be no solutions to the original equation, and therefore no dependences between the original array references. When there are solutions to this equation, then the solutions can be generated by values of the t vector. Sometimes the t vector can be used to compute dependence distances.

A typical test belonging to this class of tests is the Power test [23] that uses the Generalized GCD to obtain a t vector. The t vector is then manipulated to determine whether any inconsistencies can be detected with it. If the results are inconclusive, the test finally resorts to using an FE-based test to determine dependence. Other tests of this type are the Single Variable per Constraint test, the Acyclic test, and the Simple Loop Residue test [16]. These tests can tolerate unknown variables by just adding them to the vector of unknowns.

λ Test The Extreme Value test can be generalized to test all dimensions of subscript expressions simultaneously, as done in λ-test [14]. In the λ-test, an attempt is made to apply the loop bounds to the solution of the System of Equalities, to determine whether the hyper-plane representing the solution intersects the bounded convex set representing the loop bounds. Linear constraints are formed from Equation 7 and simplified by removing redundancies. The Extreme Value technique is applied to these simplified constraints to find simultaneous real-valued solutions by comparing with loop bounds. The testing time is relatively fast because it approximates integer-valued solutions for linear constraints with real-valued solutions. The test extends the original Extreme Value test to handle coupled subscripts, but has no capability to handle non-linear subscripts.

I Test The I test [13] is a combination of the Extreme Value test and the GCD test. The I test operates on a single equation from the System of Equalities at a time, putting it in the form of $\sum_{i=1}^{d} e_i I_i = e_0 \mid e_i \in Z$.

It first creates an interval equation from that equation, giving the upper and lower bounds of the left-hand side. In the original form, the upper and lower bounds are both equal to the constant value on the right-hand side. It iterates, selecting a term from the left-hand side on each iteration, moving it into the upper and lower bounds on the right-hand side (using the Extreme Value test mechanism), then applying the GCD test on the remaining coefficients. This process continues until either it can be proven that the equation is infeasible, or there are no more terms which can be moved.

Delta Test One main drawback of Fourier-based tests like the Omega and Power tests is their execution cost. To alleviate this problem, the Delta test [8]

was developed for certain classes of array access patterns that can be commonly found in scientific codes. In some sense, the Delta test is a faster but more restricted version of the λ-test because it simplifies the generalized Extreme Value test by using pattern matching to identify certain easily-solved forms of the dependence problem. Subscript expressions are classified into ZIV (zero-index variable), SIV (single-index variable) and MIV (multiple-index variable) forms. ZIV and SIV forms are sub-categorized into various forms which are easily and exactly solved. The test propagates constraints discovered from solving SIV forms into the solution of the MIV forms. This propagation sometimes reduces the MIV forms into SIV problems, so the technique must iterate.

Range Test The Range test [3] is a generalized application of the Extreme Value test. It can be used with symbolic coefficients, and even with non-affine functions for f and g from Figure 1. It starts with an equation of the form $\sum_{i=1}^{d} b_i I_i + b_0 = \sum_{i=1}^{d} c_i I_i + c_0 \mid b_i, c_i \in Z.$ and attempts to determine whether the minimum value of the left-hand side (f^{min}) is larger than the maximum value of the right-hand side (g^{max}), or whether the maximum value of the left-hand side (f^{max}) is smaller than the minimum value of the right-hand side (g^{min}). It uses the fact that if $f_d^{max}(I_1, I_2, \cdots, I_d) < g_d^{min}(I_1, I_2, \cdots, I_d + 1)$ for all values of all indices, and for g_d^{min} monotonically non-decreasing, then there can be no loop-carried dependence from $A(f(\mathbf{I}))$ to $A(g(\mathbf{I}))$.

First, the Range test must prove that the functions involved are monotonic with respect to the loop indices. It does this by symbolically incrementing the loop variable and subtracting the original form from the incremented form of the expression. If the result can be proven to be always positive or always negative, then the expression is monotonic with respect to that loop variable. Once the subscript expressions are proven monotonic, the Range test determines whether, for a given nesting of the loops surrounding the array references being tested, the addresses of one reference are always larger or smaller than for the other reference. This is similar to the Extreme Value test, although the whole test is structured to make use of symbolic range information.

4 Comparing the ART with Other Dependence Techniques

Comparing the ratings of the LMAD intersection algorithm with the ratings of the other dependence testing mechanisms, it is apparent that the ART mechanism is most similar to the most powerful dependence testing techniques - those based on FE. Interprocedural accuracy may even be greater for the ART than for the FE-based techniques because of the precision with which LMADs can be translated precisely across procedure boundaries, without adding complexity to the representation. The FE-based techniques have a more general formulation, and therefore can avoid restrictions, such as those placed on the inputs of the LMAD intersection algorithm.

	coef form	loop bnd form	use constr?	exact?	dist vec?	simult soln	complex	inter proc?
GCD	ints	N.A.	no	no	no	no	lin	no
Ext Val	ints	ints	loop bnd	yes:coef ±1	no	no	lin	no
FE	ints	exprs	yes	yes	yes	yes	exp	yes
Omega	ints	exprs	yes	yes	yes	yes	exp usual. poly	yes
Gen GCD	ints	exprs	yes	yes	yes	yes	exp	yes
λ	ints	ints	loop bnd	yes:2D	yes	yes	poly	no
I	ints	ints	loop bnd	yes:coef ±1	no	no	lin	no
Δ	ints	exprs	yes	usually	yes	yes	lin/poly/exp	no
Range	exprs	exprs	yes	no	no	no	poly	no
ART	exprs	exprs	yes	usually	yes	yes	exp	yes

Table 1. The characteristics of dependence testing techniques compared.

4.1 Comparison with GCD and Extreme Value

The LMAD intersection algorithm essentially combines the type of testing done within the two most basic dependence testing algorithms, the GCD test and the Extreme Value Test, although with higher accuracy. The intersection algorithm actually finds values of the loop indices for which a dependence occurs, so it does more than just determine that a dependence exists. It is able to construct the distance vector for the dependence, which is more than either the GCD or Extreme Value Test can do.

The LMAD intersection algorithm will report an intersection whenever the Extreme Value test would report a dependence. If the minimum (τ') of one descriptor is greater than the maximum ($\tau + \text{width}_d$) of the other, then `intersect` would employ step I3 for each dimension until for the scalar case the distance D would be greater than 0, meaning no intersection. In all other cases, the Extreme Value test would report a dependence, while the LMAD intersection algorithm would sometimes report no intersection.

In steps I1, I2, and I3, the base offsets of the LMADs are maintained at the original base offset plus a linear combination of the strides involved: $\tau + \sum_i t_i\delta_i$ and $\tau' + \sum_j t_j\delta_j$, where $i \neq j$. The t_i are formed from the values for m and $m+1$, according to the rules for calculating DIST in the algorithm. Therefore, when the final comparison (for $k = 0$) is made, it is equivalent to testing whether

$$\tau' + \sum_i t_i\delta_i = \tau + \sum_j t_j\delta_j \quad \text{or} \quad \tau' - \tau = \sum_j t_j\delta_j - \sum_i t_i\delta_i.$$

This means that the difference between the base offsets is made up of a linear combination of the coefficients of the loop indices, which is precisely what the GCD test attempts to test for, albeit in a crude way. The GCD test merely tells us whether it is possible that the difference between the base offsets can be expressed as a linear combination of the coefficients, while our intersection algorithm constructs precisely what the linear combination is.

4.2 Values Unknown at Compile Time

Both the GCD test and the Extreme Value test require that the coefficients and loop bounds be integer constants because of the operations they carry out. There is no way to compute the GCD of symbolic expressions, and there is no way to compute the "positive part of a number" or the "negative part of a number" symbolically, as the Extreme Value test would require.

The LMAD intersect algorithm, on the other hand, has fewer symbolic limitations. The operations used in it are all amenable to symbolic computation, except for the *mod* function. The *mod* function is only used in a check to determine whether there could be an intersection on the right, at step $I2$. If D is not a symbolic multiple of δ_k (in which case we could use the value 0 for R), then we can just always call `intersect` at $I2$. The rest of the functions can be represented within symbolic expressions. The *floor* function is simple integer division. The *min* function can be carried in a symbolic expression, and in some cases simplified algebraically.

5 Conclusion

We have described the LMAD, the ART and the LMAD intersection algorithm. We discussed the properties of the intersection algorithm in some detail and compared it with the mechanisms of a set of well-known dependence tests.

The comparison made in Table 1 demonstrates that the ART has characteristics similar to the most powerful dependence tests - those based on Fourier Elimination, yet the intersection mechanism is somewhat simpler to describe. The ART may have an advantage in interprocedural analysis, while the FE-based techniques are more general in formulation.

References

[1] V. Balasundaram and K. Kennedy. A Technique for Summarizing Data Access and its Use in Parallelism Enhancing Transformations. *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, June 1989.

[2] W. Blume. *Symbolic Analysis Techniques for Effective Automatic Parallelization.* PhD thesis, Univ. of Illinois at Urbana-Champaign, Dept. of Computer Science, June 1995.

[3] William Blume and Rudolf Eigenmann. Non-linear and symbolic data dependence testing. *IEEE Transactions on Parallel and Distributed Systems*, 9(12), December 1998.

[4] S. Cook. The complexity of theorem-proving procedures. *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, 1971. New York, NY.

[5] B. Creusillet and F. Irigoin. Interprocedural Array Region Analyses. In *Lecture Notes in Computer Science*. Springer Verlag, New York, New York, August 1995.

[6] B. Creusillet and F. Irigoin. Exact vs. Approximate Array Region Analyses. In *Lecture Notes in Computer Science*. Springer Verlag, New York, New York, August 1996.

[7] G. Dantzig. *Linear Programming and Extensions.* Princeton University Press, 1963.

[8] G. Goff, K. Kennedy, and C. Tseng. Practical Dependence Testing. In *Proceedings of the ACM SIGPLAN 91 Conference on Programming Language Design and Implementation*, pages 15–29, June 1991.

[9] M. Hall, B. Murphy, S. Amarasinghe, S. Liao, and M. Lam. Detecting Coarse-grain Parallelism Using An Interprocedural Parallelizing Compiler. *Proceedings of Supercomputing '95*, December 1995.

[10] P. Havlak and K. Kennedy. An Implementation of Interprocedural Bounded Regular Section Analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[11] J. Hoeflinger. *Interprocedural Parallelization Using Memory Classification Analysis.* PhD thesis, Univ. of Illinois at Urbana-Champaign, Dept. of Computer Science, August, 1998.

[12] J. Hoeflinger, Y. Paek, and K. Yi. Unified Interprocedural Parallelism Detection. *accepted by the International Journal of Parallel Processing*, 2000.

[13] X. Kong, D. Klappholz, and K. Pssaris. The I Test: An Improved Dependence Test for Automatic Parallelization and Vectorization. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):342–349, 1991.

[14] Z. Li, P. Yew, and C. Zhu. An Efficient Data Dependence Analysis for Parallelizing Compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):26–34, January 1990.

[15] V. Maslov. Delinearization: An Efficient Way to Break Multiloop Dependence Equations. *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, June 1991.

[16] D. Maydan, J. Hennessy, and M. Lam. Eficient and Exact Data Dependence Analysis. *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, June 1991.

[17] Y. Paek. *Automatic Parallelization for Distributed Memory Machines Based on Access Region Analysis.* PhD thesis, Univ. of Illinois at Urbana-Champaign, Dept. of Computer Science, April 1997.

[18] Y. Paek, J. Hoeflinger, and D. Padua. Simplification of Array Access Patterns for Compiler Optimizations. *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, June 1998.

[19] W. Pugh. A Practical Algorithm for Exact Array Dependence Analysis. *Communications of the ACM*, 35(8), August 1992.

[20] P. Tang. Exact Side Effects for Interprocedural Dependence Analysis. In *1993 ACM International Conference on Supercomputing, Tokyo, Japan*, pages 137–146, July 1993.

[21] P. Tu and D. Padua. Gated SSA-Based Demand-Driven Symbolic Analysis for Parallelizing Compilers. *ACM International Conference on Supercomputing*, pages 414–423, July 1995.

[22] M. Wolfe. *High Performance Compilers for Parallel Computers.* Addison-Wesley, California, 1996.

[23] M. Wolfe and C. Tseng. The Power Test for Data Dependence. *IEEE Transactions on Parallel and Distributed Systems*, September 1992.

Safe Approximation of Data Dependencies in Pointer-Based Structures

D.K. Arvind and T.A. Lewis

Institute for Computing Systems Architecture,
Division of Informatics, The University of Edinburgh,
Mayfield Road, Edinburgh EH9 3JZ, Scotland.
dka|tl@dcs.ed.ac.uk

Abstract. This paper describes a new approach to the analysis of dependencies in complex, pointer-based data structures. Structural information is provided by the programmer in the form of two-variable finite state automata (2FSA). Our method extracts data dependencies. For restricted forms of recursion, the data dependencies can be exact; however in general, we produce approximate, yet safe (*i.e.* overestimates dependencies) information. The analysis method has been automated and results are presented in this paper.

1 Introduction

We present a novel approach to the analysis of dependencies in complex, pointer-based data structures which builds on our earlier work [AL98]. The input to our analysis is structural information of pointer-based data structures, and algorithms that work over those structures. We consider algorithms that update the data in the nodes of the structure, although we disallow structural changes that would result from pointer assignment. The structural specification is given by identifying a tree backbone and additional pointers that link nodes of the tree. These links are described precisely using two-variable finite state automata (2FSA).

We can then produce dependency information for the program. This details for each runtime statement the set of statements that either read or write into the same node of the structure. Some of this information may be approximate, but we can check that it is conservative in that the correct set of dependencies will be a subset of the information we produce. For even quite small sections of program the output may be dauntingly complex, but we explore techniques for reducing it to a tractable size and extracting useful information.

The paper is organised as follows: we outline the notion of a 2FSA description in Section 2, and describe the restricted language that we use in Section 2.1. In Section 3.1 we look at an example of a recursive rectangular mesh, and follow through with the description and analysis of a simple piece of program. We deal with a more complex example in Section 4. We describe related works in Section 5, with conclusions and plans for future work in Section 6.

S.P. Midkiff et al. (Eds.): LCPC 2000, LNCS 2017, pp. 304–323, 2001.

2 Structure Descriptions Using 2FSA

We first observe how dynamic data structures are handled in a language such as C, and then relate this to our approach. Consider the following example of a tree data structure:

```
struct Tree {
    int data;
    Tree * d1;
    Tree * d2;
    Tree * d3;
    Tree * d4;
    Tree * r;
};
```

The items `data`, `d1`, `d2`, `d3`, `d4` and `r` are the *fields* of the structure, and may contain items of data (such as `data`) or pointers to other parts of the structure. We assume here that `d1`, `d2`, `d3`, `d4` point to four disjoint subtrees, and the `r` pointer links nodes together across the structure.

We next explain how this structure is represented. We have a fixed list of symbols, the *alphabet* A, that corresponds to the fields in the structure. We define a subset, $G \subseteq A$, of *generators*. These pointers form a tree backbone for this structure, with each node in the structure being identified by a unique string of symbols, called the *pathname* (a member of the set G^*), which is the path from the root of the tree to that particular node. Therefore `d1`, `d2`, `d3` and `d4` are the generators, in the example.

Our description of the structure also contains a set of relations, $\rho_i \subseteq G^* \times G^*$, one for each non-generator or *link* field i. This relation links nodes that are joined by a particular pointer. A node may be joined to more than one target node via a particular link. This allows approximate information to be represented. It is useful to consider each relation as a function from pathnames to the power set of pathnames: $F_i : G^* \rightarrow \mathcal{P}(G^*)$; each pathname maps to a set of pathnames that it may link to. In our example `r` is such a link. A *word* is a string of fields; we append words to pathnames to produce new ones.

We represent each relation as a *two-variable finite state automaton* (2FSA). These are also known as *Left Synchronous Transducers* (see for example [Coh99]). We recall that a (deterministic) Finite State Automaton (FSA) reads a string of symbols, one at a time, and moves from one state to another. The automaton consists of a finite set of states S, and a transition function $F : S \times A \rightarrow S$, which gives the next state for each of the possible input symbols in A. The string is accepted, if it ends in one of the *accept* states when the string is exhausted.

A *two-variable* FSA attempts to accept a pair of strings and inspects a symbol from each of them, at each transition. It can be thought of as a one-variable FSA, but with the set of symbols extended to $A \times A$. There is one subtlety, in that we may wish to accept strings of unequal lengths, in which case the shorter one is padded with the additional '$-$' symbol. This results in the actual set of symbols

to be: $((A \cup \{-\}) \times (A \cup \{-\}))\backslash(-,-)$, since the double padding symbol is never needed.

We can utilise non-deterministic versions of these automata. As already described, deterministic 2FSAs allow only one transition from a fixed state with a fixed symbol; non-deterministic ones relax this condition by allowing many. Most manipulations work with deterministic 2FSAs, but on occasions it may be simpler to define a non-deterministic 2FSA with a particular property, and determinise it afterwards.

We also use 2FSAs to hold and manipulate the dependency information that we gather. There are other useful manipulations of these 2FSAs which we use in our analysis.

- *Logical Operations.* We can perform basic logical operations such as AND ($\wedge$), OR ($\vee$), and NOT on these 2FSAs.
- The *Exists and Forall* automata. The one variable FSA $\exists(F)$, accepts x if there exists a y such that $(x, y) \in F$. Related to this is the 2FSA $\forall(R)$ built from the one variable FSA R, that accepts (x, y) for all y, if R accepts x.
- *Composition.* Given 2FSAs for individual fields, we wish to combine the multiplier 2FSAs into one multiplier for each word of fields that appear in the code. For instance, we may wish to find those parts of the structure which are accessed by the word `a.b` given the appropriate 2FSAs for `a` and `b`. This *composition* 2FSA can be computed: given two FSAs, $\mathcal{R}$ and $\mathcal{S}$, their composition is denoted by $\mathcal{R}.\mathcal{S}$. This is defined as : $(x, y) \in \mathcal{R}.\mathcal{S}$, if there exists a z, such that $(x, z) \in \mathcal{R}$ and $(z, y) \in \mathcal{S}$. See [ECH+92] for the details of its construction.
- The *inverse* of 2FSA $\mathcal{F}$, denoted $\mathcal{F}^{-1}$, which is built by swapping the pair of letters in each transition of $\mathcal{F}$.
- The *closure* of 2FSA $\mathcal{F}$, denoted $\mathcal{F}*$, which we discuss later in Section 3.4.

With the exception of the *closure* operation, these manipulations are exact for any 2FSAs.

2.1 Program Model

We work with a fairly restricted programming language with C-like syntax that manipulates these structures. The program operates on one global data structure. Data elements are accessed via pathnames, which are used in the same manner as conventional pointers. The program consists of a number of possibly mutually recursive functions. These functions take any number of pathname parameters, and return `void`. It is assumed that the first (`main`) function is called with the root path name. Each function may make possibly recursive calls to other functions using the syntax '`Func(w->g)`' where `g` is any field name.

The basic statements of the program are *reads* and *writes* to parts of the structure. A typical read/write statement is '`w->a = w->b`' where `w` is a variable and `a` and `b` are words of directions, and denotes the copying of an item of data from `w->b` to `w->a` within the structure. Note that we do not allow structures

to be changed dynamically by pointer assignment. This makes our analysis only valid for sections of algorithms where the data in an existing structure is being updated, without structural changes taking place.

3 The Analysis

3.1 A Rectangular Mesh Structure

We work through the example of a rectangular mesh structure, as shown in Figure 1 to demonstrate how a 2FSA description is built up from a specification. These mesh structures are often used in finite-element analysis; for example, to analyse fluid flow within an area. The area is recursively divided into rectangles, and each rectangle is possibly split into four sub-rectangles. This allows for a greater resolution in some parts of the mesh. We can imagine each rectangle being represented as a node in a tree structure. Such a variable resolution mesh results in an unbalanced tree, as shown in Fig. 1. Each node may be split further into four subnodes. Each rectangle in the mesh has four adjacent rectangles that meet along an edge in that level of the tree. For example, the *rectangle* 4 on the bottom right has *rectangle* 3 to the left of it and *rectangle* 2 above. This is also true for the smallest *rectangle* 4, except that it has a *rectangle* 3 to its right. We call these four directions l, r, d and u. The tree backbone of the structure has four generators $d1, d2, d3$ and $d4$ and linking pointers l, r, d and u. We assume that these links join nodes at the same level of the tree, where these are available, or to the parent of that node if the mesh is at a lower resolution at that point. So moving in direction u from node $d3$ takes us to node $d1$, but going up from node $d3.d1$ takes us to node $d1$, since node $d1.d3$ does not exist.

We next distill this information into a set of equations that hold this linkage information. We will then convert these equations into 2FSA descriptions of the structure. Although they will be needed during analysis, the generator descriptions do not have to be described by the programmer as they are simple enough to be generated automatically.

Let us now consider the r direction. Going right from a $d1$ node takes us to the sibling node $d2$. Similarly,we reach the $d4$ node from every $d3$. To go right from any $d2$ node, we first go up to the parent, then to the right of that parent, and down to the $d1$ child. If that child node does not exist then we link to the parent. For the $d4$ node, we go to the $d3$ child to the right of the parent. This information can be represented for the r direction by the following set of equations (where x is any pathname) :

$$r(x.d1) = x.d2 \tag{1}$$
$$r(x.d2) = r(x).d1|r(x) \tag{2}$$
$$r(x.d3) = x.d4 \tag{3}$$
$$r(x.d4) = r(x).d3|r(x) \tag{4}$$

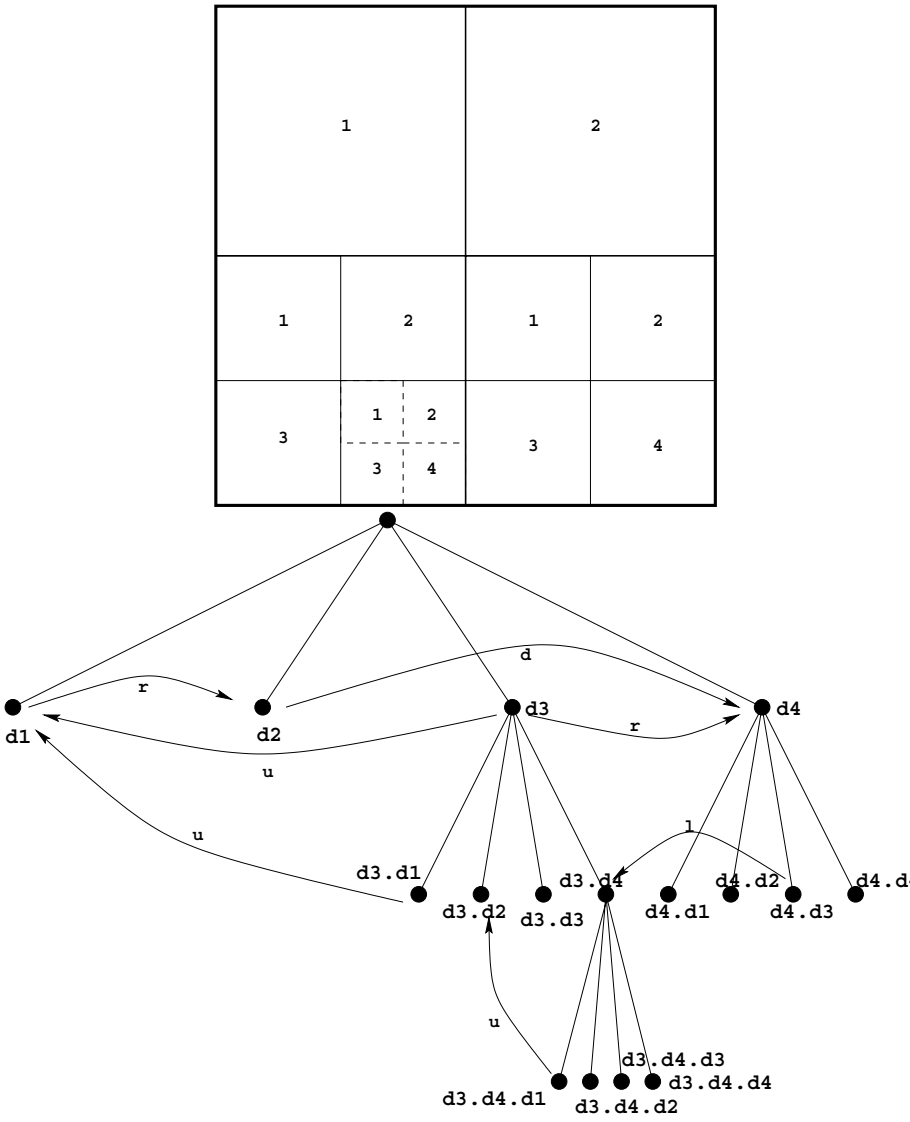


Fig. 1. Above: the variable resolution rectangular mesh. Below: The mesh is represented as a tree structure with the l, r, d, u links to adjacent rectangles. (Not all the links are shown, for the sake of brevity.)

$$d(x.d1) = x.d3$$
$$d(x.d2) = x.d4$$
$$d(x.d3) = d(x).d1|d(x)$$
$$d(x.d4) = d(x).d2|d(x)$$

$$u(x.d1) = u(x).d3|u(x)$$
$$u(x.d2) = u(x).d4|u(x)$$
$$u(x.d3) = x.d1$$
$$u(x.d4) = x.d2$$

$$l(x.d1) = l(x).d2|l(x)$$
$$l(x.d2) = x.d1$$
$$l(x.d3) = l(x).d4|l(x)$$
$$l(x.d4) = x.d3$$

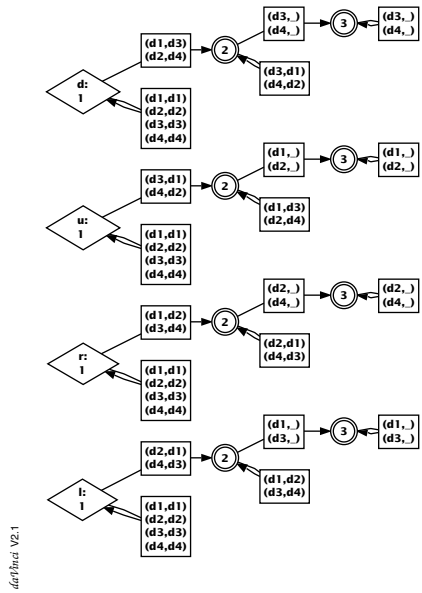


Fig. 2. Equations and 2FSA descriptions for the link directions in the mesh structure.

These equations are next converted into a 2FSA for the r direction. Note that one pathname is converted into another by working from the end of the pathname back to the beginning. The rule '$r(x.d4) = r(x).d3|r(x)$' does the following: if $d4$ is the last symbol in the string, replace it with a $d1$ or an ϵ, then apply the r direction to the remainder of the string. This is similarly true for the '$r(x.d2) = r(x).d1|r(x)$' rule. In the rule '$r(x.d1) = x.d2$' the last $d1$ symbol is replaced by a $d2$, and then outputs the same string of symbols as its input. Viewed in this way, we can create a 2FSA that accepts the paths, but in reverse order.

The correct 2FSA is produced by reversing the transitions in the automata and exchanging initial and accept states. Note that the presence of transitions labelled with ϵ may make this reversal impossible. We can, however, use this process to produce 2FSA descriptions for the other link directions l, d and u. Figure 2 illustrates the remaining set of equations and 2FSA descriptions.

3.2 The Program

```
main (Tree *root) {
  if (root != NULL) {
A:    traverse(root->d2);
B:    traverse(root->d4);
  }
}
traverse(Tree *t) {
  if (t!=NULL) {
C:    sweepl(t);
D:    traverse (t->d1);
E:    traverse (t->d2);
F:    traverse (t->d3);
G:    traverse (t->d4);
  }
}
sweepl(Tree *x) {
  if (x!=NULL) {
H:    x->l->data = x->data + x->r->data;
I:    sweepl(x->l);
  }
}
```

Fig. 3. Sample recursive functions. The $A, B, C, D, E, F, G, H, I$ are statement labels and are not part of the original code.

The recursive functions in Fig. 3 operate over the rectangular mesh structure illustrated in Fig. 1. The function `main` branches down the right hand side of

the tree calling `traverse` to traverse the whole sub-trees. The function `sweepl` then propagates `data` values out along the `l` field.

The descriptions for single fields can be used to build composite fields. For instance, the update in statement H requires the composite fields: $r \rightarrow data$ and $l \rightarrow data$, both built up from the basic descriptions by composing the 2FSAs. Each runtime statement is uniquely identified by a *control word*, defined by labelling each source code line with a symbol, and forming a word by appending the symbol for each recursive function that has been called. Each time the `traverse` function is called recursively from statement D, we append a D symbol to the control word. Source code line H therefore expands to the set of runtime control words $(A|B).(D|E|F|G)^*.C.I^*.H$.

A pathname parameter to a recursive function is called an *induction parameter*. Each time we call the function recursively from statement D, we append a $d1$ field to the induction parameter t and for statement E we similarly append a $d2$ field. The same is true for statements F and G. This information can be captured in a 2FSA that converts a given control word into the value of the parameter t, for that function call. The resulting 2FSA is shown in Fig. 4.

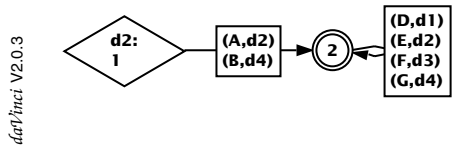


Fig. 4. A 2FSA that maps control words to values of t.

3.3 Building a General Induction Parameter 2FSA

In general for each induction parameter V_i in each function, we need to create a 2FSA, F_{V_i}, that maps all control words to the pathname value of that parameter at that point in the execution of the program. Fig. 5 outlines the construction of such a 2FSA.

If a function is called in the form `F(...,Vk,...)` where `Vk` is an induction parameter, then the substitution 2FSA will contain transition symbols of the form (A, ϵ). We aim to remove these by repeated composition with a 2FSA that will remove one ϵ at a time from the output. Provided that the transition is not in a recursive loop, *i.e.*, we can bound the number of times that the function can be called in any run of the program, applying this repeatedly will remove all such transitions. The construction of this epsilon-removal 2FSA is outlined in Fig. 6.

Since the program in Fig. 3 has a function `sweepl` that recurses by updating its induction parameter by a non-generator direction `l`, we have to approximate the dependency information for this function. This is because, in general, the required induction parameter relation will not necessarily be a 2FSA; hence we approximate with one.

- Create a non-deterministic 2FSA with a state for each induction variable, plus an additional initial state. State $i+1$ (corresponding to variable i) is the only accepting state.
- For each pathname parameter j of the `main` function, add an epsilon transition from state 1 to state $j+1$.
- For each induction variable k, we seek all call statements of the form `A: F(...,Vk->g,...)`. If `Vk->g` is passed as variable m in function F, then we add transition from state $k+1$ to state $m+1$, with symbol (A, g). Here we assume that g is a generator. If g is empty, we use an epsilon symbol in its place, then attempt to remove it later. If g is a non-generator we still use the symbol (A, g), but we need to apply the closure approximation described later.
- Determinise the nondeterministic 2FSA.

Fig. 5. The Substitution 2FSA for the variable V_i

- Create a 2FSA with one state for each generator (and link), plus another three states, an initial state, an accepting final state and an epsilon state.
- Add transitions from initial state. (d, d) symbols where d is a field (generator or link), make the transition back to the initial state. (ϵ, d) symbols move to the state for that field $(d+1)$. $(\epsilon, -)$ symbols move to the final state.
- Add transitions for each field state, (field i corresponds to state $i+1$). Symbol (i, i) leave it at state $i+1$. (i, d) moves it to state $d+1$. $(i, -)$ moves to the final state.
- Add transitions for the epsilon state. (ϵ, d) moves to $d+1$. (ϵ, ϵ) leaves it at the epsilon state. $(\epsilon, -)$ moves it to the final state.
- There are no transitions from the final state.

Fig. 6. The remove ϵ 2FSA

3.4 Formation of Approximate Closures of 2FSA

We next consider the problem of producing a 2FSA that safely approximates access information of functions that recurse in the non-generator fields. This

sort of approximation will give coarse-grained access information for a whole nest of recursive calls.

We first take the simplest example of a function that calls itself recursively, which appends a non-generator field to the induction parameter at each call. We then show how this can be used to produce approximate information for any nest of recursive calls. Consider the following program fragment:

```
sweepl(Tree *x) {
  if (x!=NULL) {
H:    x->l->data = x->data + x->r->data;
I:    sweepl(x->l);
  }
}
```

Recursion in one field can be approximated by a number of steps in that field with a 2FSA that approximates any number of recursions. In the second iteration of the function, the value of x will be `l` appended to its initial value. Iteration k can be written as $\mathtt{n}^{k-1}$, and can be readily computed for any finite value of k, although the complexity of the 2FSA becomes unmanageable for a large k. We wish to approximate all possible combinations in one 2FSA.

Definition 1. *The relation* $[=]$ *is the equality relation,* $(x, y) \in [=] \iff x = y$.

Definition 2. *The* closure *of the field* p*, written* p^**, is defined as*

$$p^* = \bigvee_{k=0}^{\infty} p^k$$

where p^0 *is defined as* $[=]$

In the example of the `l` field, the closure can be represented as a 2FSA, and the approximation is therefore exact. In general, however, this is not always the case, but we aim to approximate it safely as one. A *safe* approximation S to a relation R, implies that if $(x, y) \in R$, then $(x, y) \in S$.

We have developed a test to demonstrate that a given 2FSA R is a safe approximation to a particular closure: we use a heuristic to produce R, and then check that it is safe.

Theorem 1. *If* R *and* p *are relations such that* $R \supseteq R.p \vee [=]$*, then* R *is a safe approximation to* p^*.

Proof. Firstly, $R \supseteq R.p \vee [=] \Rightarrow R \supseteq R.p^{k+1} \bigvee_{i=1}^{k} p^i$, for any k (Proof: induction on k). If $(x, y) \in p^*$, then $(x, y) \in p^r$ for some integer r. Applying above with $k = r$ implies that $(x, y) \in R$.

Therefore, given a 2FSA, R, that we suspect might be a safe approximation, we can test for this by checking if $R \supseteq R.p \vee [=]$. This is done by forming $R \wedge (R.p \vee [=])$, and testing equality with $R.p \vee [=]$. It is worth noting that safety does not always imply that the approximation is useful; a relation that accepts every pair (x, y) is safe but will probably be a poor approximation.

To prove an approximation exact we need equality in the above comparison and an additional property; that for every node x travelling in the direction of p we will always reach the edge of the structure in a finite number of moves.

Theorem 2. *If R and p are relations such that $R = R.p \vee [=]$, and for each x there exists a k_x such that for all y, $(x, y) \notin p^{k_x}$, then $R = p^*$, and the approximation is exact.*

Proof. If $(x, y) \in R$, then $(x, y) \in R.p^{k_x} \bigvee_{i=1}^{k_y - 1} p^i$. Since (x, y) cannot be in $R.p^{k_x}$, it must be in p^i for some $i < k_x$. So $(x, y) \in p^*$.

Unfortunately we cannot always use this theorem to ascertain that an approximation is exact, as we know of no method to verify that the required property holds for an arbitrary relation. However, this property will often hold for a link field (it does for most of the examples considered here), so an approximation can be verified for exactness.

The method that we use to generate these closures creates a (potentially infinite) automaton. In practice we use a number of techniques to create a small subset of this state space:

- Identifying fail states.
- 'Folding' a set of states into one approximating state. The *failure transitions* for a state is the set of symbols that leads to the failure state from that state. If two states have the same set of failure transitions, we assume they are the same state.
- Changing transitions out of the subset so that they map to states in the subset.

We have tested the closures of around 20 2FSAs describing various structures. All have produced safe approximations, and in many cases exact ones.

3.5 Generalising to Any Loop

We can use these closure approximations in any section of the function call graph where we have a recursive loop containing a link field in addition to other fields. Starting from the 2FSA for the value of an induction variable v, we build up an expression for the possible values of v using *Arden's rules* for converting an automaton to a regular expression. This allows us to solve a system of i equations for regular expressions E_i . In particular, we can find the solution of a recursive equation $E_1 = E_1.E_2|E_3$ as

$$E_1 = E_3.(E_2)^*$$

The system of equations is solved by repeated substitution and elimination of recursive equations using the above formula. We thus obtain a regular expression for the values of v within the loop. We can then compute an approximation for this operation using OR, *composition* and *closure* manipulations of 2FSAs.

3.6 Definition and Use 2FSAs

For each *read* statement, X, that accesses `p->w`, we can append to $\mathcal{F}_p$ the statement symbol, X, and the read word, w, for that statement. Thus if $\mathcal{F}_p$ accepts (C, y), then this new 2FSA will accept $(C.X, y \rightarrow w)$, and is formed from two compositions. The conjunction of this set of 2FSAs produces another 2FSA, $\mathcal{F}_r$, that maps from any control word to all the nodes of the structure that can be read by that statement. Similarly, we can produce a 2FSA that maps from control words to nodes which are written, denoted by $\mathcal{F}_w$. These write and read 2FSAs, are derived automatically by the dependency analysis.

1. The *definition* 2FSA, that accepts the pair (control word, path-name) if the control word writes (or defines) that node of the structure.
2. The *use* 2FSA for describing nodes that are read by a particular statement.

We can now describe all the nodes which are read from and written to, by any statement. By combining the read and write 2FSAs we create a 2FSA that link statements when one writes a value that the other reads, *i.e* the statements *conflict*. The conflict 2FSA, $\mathcal{F}_{conf}$, is given by $\mathcal{F}r.(\mathcal{F}w)^{-1}$. We can now describe all the nodes which have been read from, and written to, by any statement. By combining these read and write 2FSAs, we can now create 2FSAs that links statements for each of the possible types of dependency (read after write, write after read and write after write). The conjunction of these 2FSAs forms the *conflict* 2FSA:

$$\mathcal{F}_{conf} = \mathcal{F}r.(\mathcal{F}w)^{-1} \cup \mathcal{F}w.(\mathcal{F}r)^{-1} \cup \mathcal{F}w.(\mathcal{F}w)^{-1}$$

We may be interested in producing *dataflow* information, *i.e.* associating with each read statement, the write statement that produced that value. We define the *causal* 2FSA, $\mathcal{F}_{causal}$ which accepts control words X, Y, only if Y occurs before X in the sequential running of the code. The 2FSA in Fig.7 is the *conflict* 2FSA for the example program, which has also been ANDed with $\mathcal{F}_{causal}$, to remove any false sources that occur after the statement.

3.7 Partitioning the Computation

We now consider what useful information can be gleaned from the conflict graph with regard to the parallel execution of the program. Our approach is as follows:

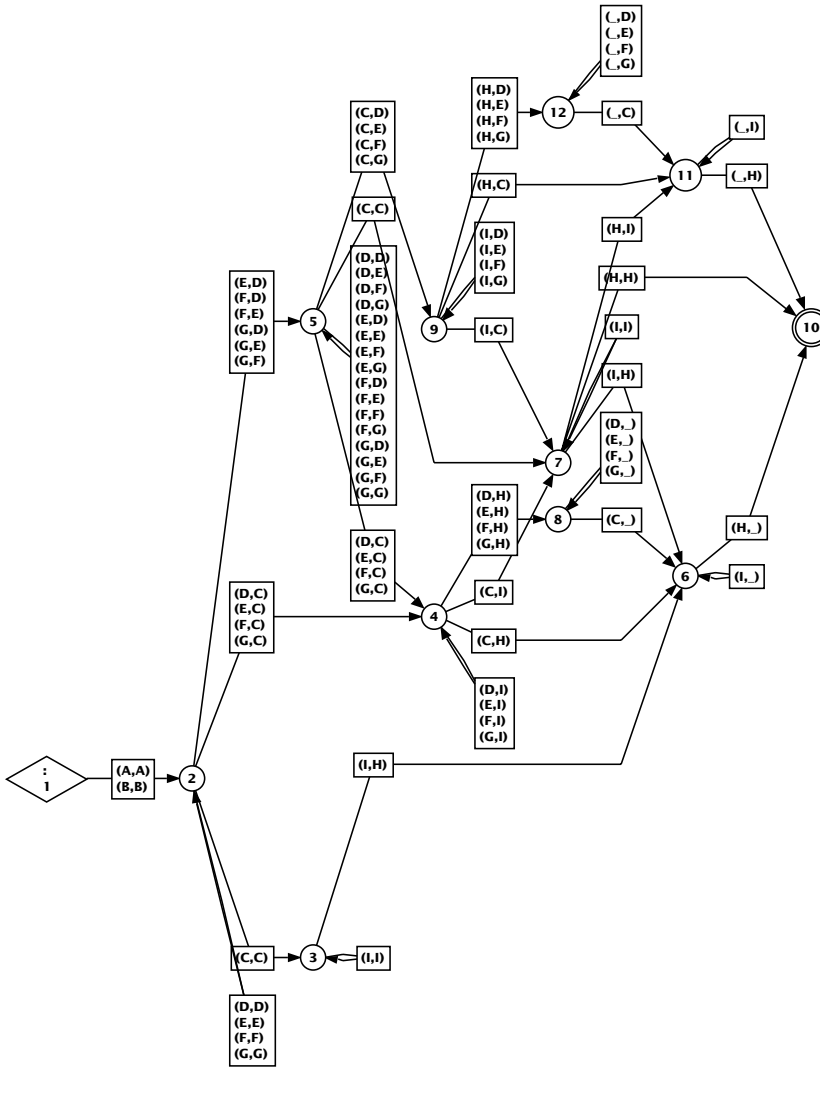


Fig. 7. The conflict 2FSA for the example code.

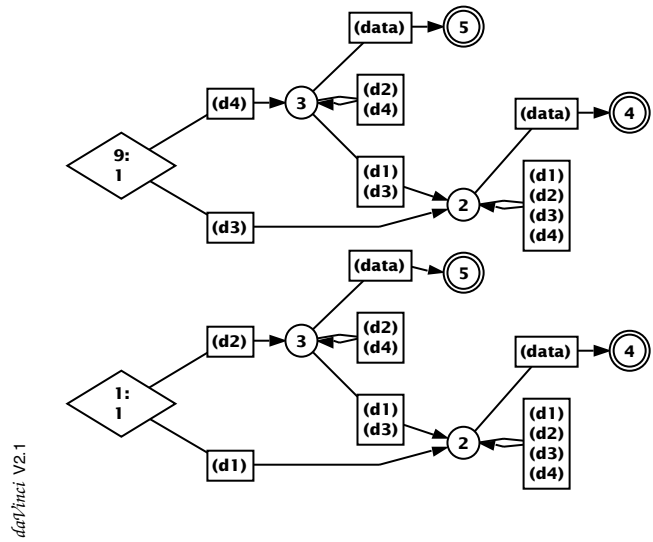


Fig. 8. The values read by and written to by each of the threads.

- Define execution threads by splitting the set of control words into separate partitions. Each partition of statements will execute as a separate thread. At present these are regular expressions supplied by the programmer.
- Use the *conflict* 2FSA to compute the dependencies between threads.

We apply this approach to the running example. We can take as our two partitions of control words:

1. $(A).(D|E|F|G)^*.C.I^*.H$
2. $(B).(D|E|F|G)^*.C.I^*.H$

We can then compute the total set of values read by and written to each statement in each thread. This is shown in Fig.8. Since there is no overlap the two threads are completely independent and can therefore be spawned safely.

4 A Second Example

We next describe how this approach can be applied to a larger program, as shown in Fig. 9. Consider the function calls in statements `A` and `B` being spawned as separate threads. The resulting *conflict* 2FSA for this example has 134 nodes, too large to interpret by manual inspection. We describe an approach for extracting precise information that can be used to aid the parallelisation process.

```
main (Tree *node) {
  if (node!=NULL) {
A:    update(node);
B:    main (node->d1);
C:    main (node->d2);
  }
}
update(Tree *w) {
  if (w!=NULL) {
D:    sweepl(w);
E:    propagate(w);
  }
}
propagate(Tree *p) {
  if (p!=NULL) {
    p->data = p->l->data + p->r->data
F:      + p->u->data + p->d->data;
G:    propagate(p->d1);
H:    propagate(p->d2);
I:    propagate(p->d3);
J:    propagate(p->d4);
  }
}
sweepl(Tree *x) {
  if (x!=NULL) {
K:    x->l->data = x->data;
L:    sweepl(x->l);
  }
}
```

Fig. 9. The second example program

A particular statement, X say, in the second thread may conflict with many statements in the first thread. Delaying the execution of X until all the conflicting statements have executed ensures that the computation is carried out in the correct sequence. We can therefore compute information that links a statement to the ones whose execution it must wait for.

We map X to Y, such that if X conflicts with Z, then Z must be executed earlier than Y. We produce the *wait-for* 2FSA by manipulating the *conflict* and *casual* 2FSAs:

$$\mathcal{F}_{\textit{wait-for}} = \mathcal{F}_{\textit{conf}} \wedge \mathrm{Not}(\mathcal{F}_{\textit{conf}} . \mathcal{F}_{\textit{causal}})$$

We use this approach to trim the conflict information and produce the one in Fig. 10. We are interested in clashes *between* the two threads, so only the portion

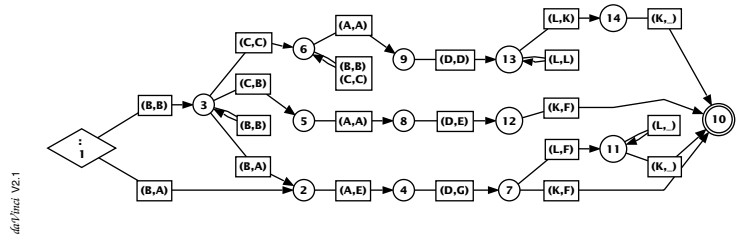


Fig. 10. The 'waits-for' information

that begins with the (B, A) symbol is relevant. We extract the information that the statements given by $B.A.D.L.(L)^*.K$ must wait for $A.E.G.F$ to execute. This implies that to ensure the threads execute correctly we would need to insert code to block these statements from executing until the statement $A.E.G.F$ has done so.

4.1 Implementation

We have implemented the techniques described in this paper in C++ using the 2FSA manipulation routines contained in the 'kbmag' library [Hol]. In addition we use the graph visualisation application 'daVinci' [FW] for viewing and printing our 2FSA diagrams. We ran our analysis codes on an 400MHz Pentium II Gnu/Linux machine. The smaller example took 283 seconds, and the larger one 497 seconds. Much of this time is spent in the closure computation routines, and in the case of the larger example the production of the waits-for information from the large conflict graph.

5 Related Work

The ASAP approach [HHN94b] uses three different types of axiom to store information about the linkages in the structure.

$$
\begin{aligned}
&1) \quad \forall p \quad p.RE1 \neq p.RE2 \\
&2) \quad \forall p \neq q \quad p.RE1 \neq q.RE2 \\
&3) \quad \forall p \quad p.RE1 = p.RE2
\end{aligned}
$$

where p, q are any path names and $RE1, RE2$ are regular expressions over the alphabet of fields . These axioms are then used to prove whether two pointers can be aliased or not. In comparison, our properties are of the form $p.RE1.n = p.RE2$ where $RE1$ and $RE2$ are regular expressions in the generator directions and n is

a link. They are slightly more powerful in that the 2FSA description allows $RE1$ and $RE2$ to be dependent. Provided that the regular expressions do not include link directions, we can express ASAP axioms as 2FSAs and combine them into our analysis. In fact, even if the expression has link directions inside a Kleene star component, we can use the closure approximation method to approximate this information and use it in our analysis. ASAP descriptions are an improvement on an earlier method, ADDS, which used different dimensions and linkages between them to describe structures. Thus ASAP is more suited to structures with a multi-dimensional array-like backbone.

Comparison with the ADDS/ASAP description for a binary tree with linked leaves (see [HHN94a]), shows that our specification is much more complicated. To justify this, we demonstrate that the 2FSA description can resolve dependencies more accurately. Consider the ADDS description of the binary tree,

```
type Bintree [down][leaves] {
  Bintree *left,*right is uniquely forward along down;
  Bintree *next        is uniquely forward along leaves;
}
```

This description names two dimensions of the structure `down` and `leaves`. The `left` and `right` directions form a binary tree in the `down` direction, the `next` pointers create a linked list in the `leaves` dimension. The two dimensions are not described as disjoint, since the same node *can* be reached via different routes along the two dimensions. Now consider the following code fragment, where two statements write to some subnodes of a pointer 'p'.

```
p->l->next->next = ...
p->r = ...
```

Dependency analysis of these statements will want to discover if the two pointers on the left sides can ever point to the same node. Since the sub-directions contain a mixture of directions from each dimension `[down]` and `[leaves]`, ADDS analysis must assume conservatively that there may be a dependence. The 2FSA description however can produce a 2FSA that accepts all pathnames p for which these two pointers are aliased. This FSA is empty, and thus these writes are always independent.

The 'Graph Types' [KS93] are not comparable to ours since they allow descriptions to query the type of nodes and allow certain runtime information, such as whether a node is a leaf to be encoded. If we drop these properties from their descriptions then we can describe many of their data structures using 2FSAs. This conversion can probably not be done automatically. Their work does not consider dependency analysis; it is mainly concerned with program correctness and verification. [SJJK97] extends this to verification of list and tree programs, and hints that this analysis could be extended to graph types.

Shape types [FM97] uses grammars to describe structures, which is more powerful than our 2FSA-based approach. Structure linkages are represented by a multi-set of field and node names. Each field name is followed by the pair of nodes

that it links. The structure of the multi-set is given by a context-free grammar. In addition, operations on these structures are presented as *transformers*, rewrite rules that update the multi-sets. However, many of the questions we want to ask in our dependence analysis may be undecidable in this framework. If we drop their convention that the pointers from a leaf node point back to itself, we can describe the dependence properties of their example structures (e.g. skip lists of level two, binary trees with linked leaves and left-child, right-sibling trees) using our formalism.

In [CC98], the authors use formal language methods (pushdown automata and rational transducers) to store dependency information. The pushdown automata are used for array structures, the rational transducers for trees; more complex structures are not considered. Rational transducers are more general than 2FSAs, since they allow ϵ transitions that are not at the end of paths. This extension causes problems when operations such as intersections are considered. Indeed only ones which are equivalent to 2FSAs can be fully utilised. Handling multidimensional arrays in their formalism will require a more sophisticated language. Our method is not intended for arrays, although they can be described using 2FSAs. However, the analysis will be poor because we have to approximate whichever direction the code recurses through them.

[BRS99] allows programs to be annotated with reachability expressions, which describe properties about pointer-based data structures. Their approach is generally more powerful than ours, but they cannot, for example, express properties of the form $x.RE1 = y.RE2$. Our 2FSAs can represent these sort of dependencies. Their approach is good for describing structures during phases where their pointers are manipulated dynamically, an area our method handles poorly at best. Their logic is decidable, but they are restricted in that they do not have a practical decision algorithm.

[Fea98] looks at dependencies in a class of programs using *rational transducers* as the framework. These are similar, but more general than the 2FSAs we use here. The generality implies that certain manipulations produce problems that are undecidable, and a semi-algorithm is proposed as a partial solution. Also the only data structures considered are trees, although extension to doubly linked lists and trees with upward pointers is hinted at. However, even programs that operate over relatively simple structures like trees can have complex dependency patterns. Our method can be used to produce the same information from these programs. This indicates a wider applicability of our approach from the one involving structures with complex linkages.

In [Deu94] the author aims to extract aliasing information directly from the program code. A system of symbolic alias pairs *SAPs* is used to store this information. Our 2FSAs form a similar role, but we require that they are provided separately by the programmer. For some 2FSAs the aliasing information can be expressed as a SAP. For example, the information for the n direction in the binary tree can be represented in (a slightly simplified version of) SAP notation as:-

$$(< X(\rightarrow l)(\rightarrow r)^{k_1} \rightarrow n, X(\rightarrow r)(\rightarrow l)^{k_2} >, k_1 = k_2)$$

The angle brackets hold two expressions that aliased for all values of the parameters k_1, k_2 that satisfy the condition $k_1 = k_2$. SAPs can handle a set of paths that are not regular, so they are capable of storing information that a finite state machine cannot. These expressions are not, however, strictly more powerful than 2FSAs. For example, the alias information held in the 2FSA descriptions for the rectangular mesh cannot be as accurately represented in SAP form.

In [RS98] the authors describe a method for transforming a program into an equivalent one that keeps track of its own dataflow information. Thus each memory location stores the statement that last wrote into it. This immediately allows methods that compute alias information to be used to track dataflow dependencies. The method stores the statement as a line number in the source code, rather than as a unique run time identifier, such as the control words we use. This means that dependencies between different procedures, or between different calls of the same procedure, will not be computed accurately. Although we describe how dataflow information can be derived by our approach, we can still produce alias information about pointer accesses. So the techniques of [RS98] could still be applied to produce dependency information.

6 Conclusions and Future Work

This paper builds on our earlier work in which the analysis was restricted to programs which recursed in only the generator directions. We have now extended the analysis, to non-generator recursion. The approximation is safe in the sense that no dependencies will be missed. The next step would be to extend the analysis to code that uses pointer assignment to dynamically update the structure.

As it stands the computation of the waits-for dependency information involves a large intermediate 2FSA, which will make this approach intractable for longer programs. We are currently working on alternative methods for simplifying the conflict 2FSA that would avoid this.

We are also pursuing the idea of attaching probabilities to the links in the structure description. A pointer could link a number of different nodes together, with each pair being associated with a probability. We could then optimise the parallelisation for the more likely pointer configurations while still remaining correct for all possible ones.

References

[AL98] D. K. Arvind and Tim Lewis. Dependency analysis of recursive data structures using automatic groups. In Siddhartha Chatterjee et al, editor, *Languages and Compilers for Parallel Computing, 11th International Workshop LCPC'98*, Lecture Notes in Computer Science, Chapel Hill, North Carolina, USA, August 1998. Springer-Verlag.

[BRS99] Michael Benedikt, Thomas Reps, and Mooly Sagiv. A decidable logic for describing linked data structures. In S. D. Swierstra, editor, *ESOP '99: European Symposium on Programming, Lecture Notes in Computer Science*, volume 1576, pages 2–19, March 1999.

[CC98] A. Cohen and J.-F. Collard. Applicability of algebraic transductions to data-flow analysis. In *Proc. of PACT'98*, Paris, France, October 1998.

[Coh99] Albert Cohen. *Program Analysis and Transformation: From the Polytope Model to Formal Languages*. PhD thesis, University of Versailles, 1999.

[Deu94] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 230–241, New York, NY, 1994. ACM Press.

[ECH+92] David B. A. Epstein, J. W. Cannon, D. E. Holt, S. V. F. Levy, M. S. Paterson, and W. P. Thurston. *Word Processing in Groups*. Jones and Bartlett, 1992.

[Fea98] Paul Feautrier. A parallelization framework for recursive programs. In *Proceedings of EuroPar*, volume LNCS 1470, pages 470–479, 1998.

[FM97] P. Fradet and D. Le Metayer. Shape types. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1997.

[FW] Michael Frohlich and Mattias Werner. The davinci graph visualisation tool. http://www.informatik.uni-bremen.de/daVinci/.

[HHN94a] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. A general data dependence test for dynamic pointer-based data structures. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 100–104, June 1994.

[HHN94b] Joseph Hummel, Laurie J Hendren, and Alexandru Nicolau. A language for conveying the aliasing properties of dynamic, pointer-based data structures. In *Proceedings of the 8th International Parallel Processing Symposium*, April 1994.

[Hol] Derek Holt. Package for knuth-bendix in monoids, and automatic groups. http://www.maths.warwick.ac.uk/ dfh/.

[KS93] Nils Klarlund and Michael I. Schwartzbach. Graph types. In *Proceedings of the ACM 20th Symposium on Principles of Programming Languages*, pages 196–205, January 1993.

[RS98] J. L. Ross and M. Sagiv. Building a bridge between pointer aliases and program dependences. *Nordic Journal of Computing*, (8):361–386, 1998.

[SJJK97] Michael I. Schwartzbach, Jakob L. Jensen, Michael E. Jorgensen, and Nils Klarlund. Automatic verification of pointer programs using monadic second order logic. In *Proceedings of the Conference on Programming Language design and implementation*. ACM, 1997.

OpenMP Extensions for Thread Groups and Their Run-Time Support

Marc Gonzalez, Jose Oliver, Xavier Martorell, Eduard Ayguade,
Jesus Labarta, and Nacho Navarro

Computer Architecture Department, Technical University of Catalonia,
cr. Jordi Girona 1-3, Mòdul D6, 08034 - Barcelona, Spain

Abstract. This paper presents a set of proposals for the OpenMP shared–memory programming model oriented towards the definition of thread groups in the framework of nested parallelism. The paper also describes the additional functionalities required in the runtime library supporting the parallel execution. The extensions have been implemented in the OpenMP NanosCompiler and evaluated in a set of real applications and benchmarks. In this paper we present experimental results for one of these applications[1].

1 Introduction

Parallel architectures are becoming affordable and common platforms for the development of computing-demanding applications. Users of such architectures require simple and powerful programming models to develop and tune their parallel applications with reasonable effort. These programming models are usually offered as library implementations or extensions to sequential languages that express the available parallelism in the application. Language extensions are defined by means of directives and language constructs (e.g. OpenMP [8], which is the emerging standard for shared–memory parallel programming).

In general, multiple levels of parallelism appear in the majority of numerical applications in science and engineering. Although OpenMP accepts the specification of multiple levels of parallelism (through the nesting of parallel constructs), most current programmers only exploit a single level of parallelism. This is because their applications achieve satisfactory speed-ups when executed in mid-size parallel platforms or because most current systems supporting OpenMP (compilers and associated thread–level layer) sequentialize nested parallel constructs. Exploiting a single level of parallelism means that there is a single thread (master) that produces work for other threads (slaves). Once parallelism is activated, new opportunities for parallel work creation are ignored by the execution environment. Exploiting a single–level of parallelism (usually around loops) may incur in low performance returns when the number of processors to run the

[1] The reader is referred to the extended version of this paper (available at *http://www.ac.upc.es/nanos*) for additional details about the rest of applications.

S.P. Midkiff et al. (Eds.): LCPC 2000, LNCS 2017, pp. 324–338, 2001.

application increases. When multiple levels of parallelism are allowed, new opportunities for parallel work creation result in the generation of work for all or a restricted number of threads (as specified by the `NUM_THREADS` clause in v2.0 of the OpenMP specification).

In current practice, the specification of these multiple levels of parallelism is done through the combination of different programming models and interfaces, like for example the use of MPI coupled with either OpenMP or High Performance Fortran [2]. The message passing layer is used to express outer levels of parallelism (coarse grain) while the data parallel layer is used to express the inner ones (fine grain). The parallelism exploited in each layer is statically determined by the programmer and there are no chances to dynamically modify these scheduling decisions.

Other proposals consists in offering work queues and an interface for inserting application tasks before execution, allowing several task descriptions to be active at the same time (e.g. the Illinois–Intel Multithreading library [3]). Kuck and Associates, Inc. has also made proposals to OpenMP to support multi–level parallelism through the WorkQueue mechanism [10], in which work can be created dynamically, even recursively, and put into queues. Within the WorkQueue model, nested queuing permits a hierarchy of queues to arise, mirroring recursion and linked data structures. These proposals offer multiple levels of parallelism but do not support the logical clustering of threads in the multilevel structure, which we think is a necessary aspect to allow good work distribution and data locality exploitation. The main motivation of this paper is to make a proposal for an extension of the OpenMP programming model to define how the user might have control over the work distribution when dealing with multiple levels of parallelism.

Scaling to a large number of processors faces with another challenging problem: the poor interaction between the parallel code and the deep memory hierarchies of contemporary shared–memory systems. To surmount this problem, some vendors and researchers propose the use of user–directed page migration and data layout directives with some other directives and clauses to perform the distribution of work following the expressed data distribution [12, 1]. Our proposal relieves the user from this problem and proposes clauses that allow the programmer to control the distribution of work to groups of threads and ensure the temporal reuse of data (both across multiple parallel constructs and multiple instantiations of the same parallel construct). Although not used in the experimental evaluation in this paper, we assume the existence of a smart user–level page migration engine [7]. This engine, in cooperation with the compiler, tries at runtime to accurately and timely fix both poor initial page placement schemes (emulating static data distributions in data parallel languages) and in some cases, follow the dynamic page placement requirements of the application.

The paper is organized as follows. Section 2 describes the extensions to the OpenMP programming model proposed in this paper. These extensions have been included in the experimental NanosCompiler based on Parafrase–2 [9]. Section 3 describes the main functionalities of the run–time system that supports

the proposed extensions. Experimental results are reported in Section 4. Finally, some conclusions are presented in Section 5.

2 Extensions to OpenMP

This section presents the extensions proposed to the OpenMP programming model. They are oriented towards the organization of the threads that are used to execute the parallelism in a nest of `PARALLEL` constructs when exploiting multiple levels of parallelism and the allocation of work to them.

2.1 OpenMP v2.0

We assume the standard execution model defined by OpenMP[8]. A program begins execution as a single process or thread. This thread executes sequentially until the first parallel construct is found. At this time, the thread creates a team of threads and it becomes its master thread. The number of threads in the team is controlled by environment variables, the `NUM_THREADS` clause, and/or library calls. Therefore, a team of threads is defined as the set of threads that participate in the execution of any work inside the parallel construct. Threads are consecutively numbered from zero to the number of available threads (as returned by the intrinsic function `omp_get_num_threads`) minus one. The master thread is always identified with 0. Work–sharing constructs (`DO`, `SECTIONS` and `SINGLE`) are provided to distribute work among the threads in a team.

If a thread in a team executing a parallel region encounters another (nested) parallel construct, most current implementations serialize its execution (i.e. a new team is created to execute it with only one thread). The actual definition assumes that the thread becomes the master of a new team composed of as many threads as indicated by environment variables, the `NUM_THREADS` clause applied to the new parallel construct, and/or library calls. The `NUM_THREADS` clause allows the possibility of having different values for each thread (and therefore customizing the amount of threads used to spawn the new parallelism a thread may find).

In the next subsections we propose a generalization of the execution model by allowing the programmer to define a hierarchy of *groups of threads* which are going to be involved in the execution of the multiple levels of parallelism. We also propose to export the already existing thread name space and make it visible to the programmer. This allows the specification of work allocation schemes alternative to the default ones. The proposal to OpenMP mainly consists of two additional clauses: `GROUPS` and `ONTO`. Moreover, the proposal also extends the scope of the synchronization constructs with two clauses: `GLOBAL` or `LOCAL`.

2.2 Definition of Thread Groups

In our proposal, a *group of threads* is composed by a number of consecutive threads following the active numeration in the current team. In a parallel construct, the programmer may define the number of groups and the composition

of each one. When a thread in the current team encounters a parallel construct defining groups, the thread creates a new team and it becomes its master thread. The new team is composed of as many threads as groups are defined; the rest of threads are reserved to support the execution of nested parallel constructs. In other words, the groups definition establishes the threads that are involved in the execution of the parallel construct plus an allocation strategy or scenario for the inner levels of parallelism that might be spawned. When a member of this new team encounters another parallel construct (nested to the one that caused the group definition), it creates a new team and deploys its parallelism to the threads that compose its group. Groups may overlap and therefore, a thread may belong to more than one group.

Figure 1 shows different definitions of groups. The example assumes that 8 threads are available at the time the corresponding parallel construct is found. In the first scenario, 4 groups are defined with a disjoint and uniform distribution of threads; each group is composed of 2 threads. In the second scenario, groups are disjoint with a non uniform distribution of the available threads: 4 groups are defined with 4, 2, 1 and 1 threads. The third scenario shows a uniform definition of overlapped groups. In this case 3 groups are defined, each one with 4 threads; therefore, each group shares 2 threads with another group.

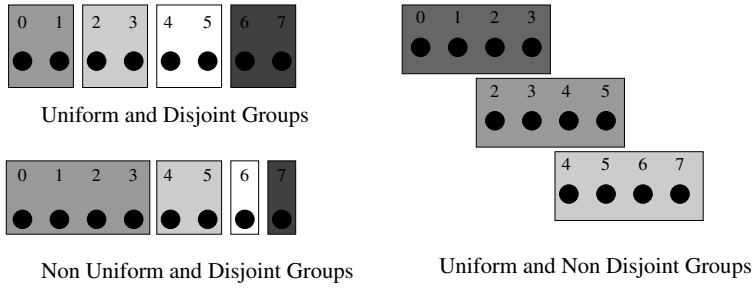


Fig. 1. Examples of group definitions.

The `GROUPS` Clause. This clause allows the user to specify any of the aforementioned group definitions. It can only appear in a `PARALLEL` construct or combined `PARALLEL DO` and `PARALLEL SECTIONS` constructs.

```
C$OMP PARALLEL [DO|SECTIONS] [GROUPS(gspec)]
```

The most general format for the groups specifier `gspec` allows the specification of all the parameters in the group definition: the number of groups, the identifiers of the threads that participate in the execution of the parallel region, and the number of threads composing each group:

```
GROUPS(ngroups, masters, howmany)
```

The first argument (`ngroups`) specifies the number of groups to be defined and consequently the number of threads in the team that is going to execute the parallel construct. The second argument (`masters`) is an integer vector with the identifiers (using the active numeration in the current team) of the threads that

will compose the new team. Finally, the third argument (howmany) is an integer vector whose elements indicate the number of threads that will compose each group. The vectors have to be allocated in the memory space of the application and their content and correctness have to be guarenteed by the programmer. This format for the GROUPS clause allows the specification of all the scenarios shown in Figure 1. For example, the second scenario in this figure could be defined with:

```
ngroups = 4
masters[4] = {0, 4, 6, 7}
howmany[4] = {4, 2, 1, 1}
```

Our experience in the use of thread groups has shown that, in a large number of real applications, the definition of non–overlapping groups is sufficient. This observation motivated the following alternative format for the groups specifier gspec:

```
GROUPS(ngroups,weight)
```

In this case, the user specifies the number of groups (ngroups) and an integer vector (weight) indicating the relative weight of the computation that each group has to perform. From this information and the number of threads available in the team, the runtime is in charge of computing the two previous vectors (masters and howmany). This eases the use of groups because the programmer is relieved from the task of determining their exact composition. Vector weight is allocated by the user in the application address space and it has to be computed from information available within the application itself (for instace iteration space, computational complexity or even information collected at runtime).

In our current implementation, the composition of the groups is computed using a predefined algorithm. The algorithm assigns all the available threads to the groups and ensures that each group at least receives one thread. The main body of the algorithm is as follows (using Fortran90 sintax):

```
howmany(1:ngroups) = 1
do while (sum(howmany(1:ngroups)) .lt. nthreads)
   pos = maxloc(weight(1:ngroups)/
                           howmany(1:ngroups))
   howmany(pos(1)) = howmany(pos(1)) + 1
end do

masters(1) = 0
do i = 1, ngroups-1
  masters(i+1) = masters(i) + howmany(i)
end do
```

In this algorithm, nthreads is the number of threads that are available to spawn the parallelism in the parallel construct containing the group definition. Notice that the last scenario in Figure 1 cannot be expressed using this format because it requires overlapping groups.

A second shortcut is used to specify uniform non-overlapping groups, i.e. when all the groups have similar computational requirements:

```
GROUPS(ngroups)
```

The argument `ngroups` specifies the number of groups to be defined. This format assumes that work is well balanced among groups and therefore all of them require the same number of threads to exploit inner levels of parallelism. The runtime is in charge of computing the composition of each group by simply equidistributing the available threads among the groups. Notice that this format only allows the specification of the first scenario in Figure 1.

Finally, the last format allows the user to specify an scenario in which the number of non–overlapping groups is constant and can be specified in the source code of the application. The programmer also specifies the relative weight of each group. The format is as follows:

```
GROUPS(gdef[,gdef])
```

where each `gdef` has the following format:

```
gdef = [name]:nthreads
```

The number of groups to be defined is established by the number of `gdef` fields in the directive. Each group definer includes an optional `name` (which can be used to identify the groups at the application level, see next subsection) and an expression `nthreads` specifying the weight of the group. The runtime applies the same algorithm to compute the actual composition of each group.

2.3 Work Allocation

OpenMP establishes a thread name space that numbers the threads composing a team from 0 to the total number of threads in the team minus one. The master of the team has number 0. Our proposal consists on exporting this thread name space to the user with the aim of allowing him to stablish a particular mapping between the work divided by the OpenMP work–sharing constructs and the threads in the group. This is useful to improve data locality along the execution of multiple parallel constructs or instantiations of the same parallel construct.

The `ONTO` Clause. The work–sharing constructs in OpenMP are `DO`, `SECTIONS` and `SINGLE`. The `ONTO` clause enables the programmer to change the default assignment of the chunks of iterations originated from a `SCHEDULE` clause in a `DO` work–sharing construct or the default lexicographical assignment of code sections in a `SECTIONS` work–sharing construct.

The syntax of the `ONTO` clause applied to a `DO` work–sharing construct is as follows:

```
C$OMP DO [ONTO(target)]
```

The argument `target` is an expression that specifies which thread in the current team will execute each particular chunk of iterations. If the expression contains the loop control variable, then the chunk number (numbered starting at 0) is used to determine which thread in the team has to execute it. The loop control variable is substituted by the chunk number and then a *'modulo the number of threads in the team'* function is applied to the resulting expression.

For instance, assume a parallel `DO` annotated with an `ONTO(2*i)` clause, `i` being the loop control variable. This clause defines that only those members

of the team with even identifiers will execute the chunks of iterations coming from the loop. If `i` is a loop invariant variable, then the thread with identifier `mod(2*i,nthreads)`, `nthreads` being the number of threads in the team will execute all the chunks originated for the parallel execution of the loop. If not specified, the default clause is `ONTO(i)`, `i` being the loop control variable of the parallel loop.

For the `SECTIONS` work–sharing construct, the `ONTO(target)` clause is attached to each `SECTION` directive. Each expression `target` is used to compute the thread that will execute the statements parceled out by the corresponding `SECTION` directive. If the `ONTO` clause is not specified the compiler will assume an assignment following the lexicographical order of the sections.

For the `SINGLE` work–sharing construct, the `ONTO` clause overrides the dynamic nature of the directive, thus specifying the thread that has to execute the corresponding code. If not specified, the first thread reaching the work–sharing construct executes the code.

2.4 Thread Synchronization

The definition of groups affects the behavior of implicit synchronizations at the end of parallel and work–sharing constructs as well as synchronization constructs (`MASTER`, `CRITICAL` and `BARRIER`). Implicit synchronizations only involve those threads that belong to the team participating in the execution of the enclosing parallel construct. However, the programmer may be interested in forcing a synchronization which affects to all the threads in the application or just to the threads that compose the group. In order to differentiate both situations, clauses `GLOBAL` and `LOCAL` are provided. By default, `LOCAL` is assumed.

2.5 Intrinsic Functions

One new intrinsic function has been added to the OpenMP API and two existing ones have been redefined:

- Function `omp_get_num_threads` returns the number of threads that compose the current team (i.e. the number of groups in the scope of the current `PARALLEL` construct).
- `omp_get_threads` returns the number of threads that compose the group the invoking thread belongs to (and that are available to execute any nested `PARALLEL` construct).
- Function `omp_get_thread_num` returns the thread identifier within the group (between 0 and `omp_get_num_threads`-1).

3 Example: MBLOCK Kernel

In this section we illustrate the use of the proposed directives with *MBLOCK*, a generic multi-block kernel. For additional examples (which include some other kernels and SPEC95 benchmarks), the reader is referred to the extended version of this paper [4].

3.1 MBLOCK Kernel

The *MBLOCK* kernel simulates the propagation of a constant source of heat in an object. The propagation is computed using the Laplace equation. The output of the benchmark is the temperature on each point of the object. The object is modeled as a multi-block structure composed of a number of rectangular blocks. Blocks may have different size (defined by vectors `nx`, `ny` and `nz`). They are connected through a set of links at specific positions. Blocks are stored consecutively in a vector named `a`. The connections between blocks are stored in a vector named `link` in which each element contains two indices to vector `a`.

The algorithm has two different phases. A first one where all data is read (sizes of the blocks and connections points). All points are assumed to have 0 degrees of temperature when starting the simulation except for a single point (`ihot`) that has a fixed (forced) temperature over the whole simulation time. The second phase is the solver, which consists of an iterative time-step loop that computes the temperature at each point of the object. The initialization done exploiting two levels of parallelism. The outer level exploits the parallelism at the level of independent blocks (loop `iblock`). The inner level of parallelism is exploited in the initialization of the elements that compose each block. The same multilevel structure appears in the solver (computation for each independent block and computation within each block). The interaction between the points that belong to two blocks is done exploiting a single level of parallelism.

Figure 2 shows the skeleton for an OpenMP parallel version of the kernel. In this version, groups of threads are defined using a vector that stores the size of each block. This vector is used to compute, at runtime, the actual composition of each group according to the total number of processors available. For instance, Figure 3.a shows the composition of the groups when 32 processors are available and 9 independent blocks are defined in the application. Notice that the algorithm currently implemented assumes that at least a processor is assigned to each group. The rest of processors are assigned according to the weight of each group.

Figure 3.b shows a different allocation which assumes that the largest block is efficiently executed exploiting a single level of parallelism (i.e. devoting all the processors available to exploit the parallelism within each block). However, the rest of the blocks are not able to efficiently exploit this large number of processors. In this case, computing several blocks in parallel and devoting a reduced number of processors to compute each block leads to a more efficient parallel execution. Finally, several small blocks can also be executed sharing a single processor in order to increase the global utilization. This definition of groups requires the use of the general specifier for the `GROUPS` clause and the definition of a function that computes vectors `masters` and `howmany`. In the example, the vectors should be initialized as follows:

```
ngroups = 9
masters[9] = {0, 0, 15, 30, 30, 30, 31, 31, 31}
howmany[9] = {32, 15, 15, 1, 1, 1, 1, 1, 1}
```

Notice that the first group overlaps with the rest of groups and that the last 6 groups are also overlapped in a single processor.

4 The Run-Time Support: NthLib

The execution model defined in section 2 requires some functionalities not available in most runtime libraries supporting parallelizing compilers. These functionalities are provided by our threads library NthLib.

4.1 Work–Descriptors and Stacks

The most important aspect to consider is the support for spawning nested parallelism. Multi–level parallelism enables the generation of work from different si-

```
      PROGRAM MBLOCK
      ...
C Initialize work and location
      do iblock=1,nblock
        work(iblock) = nx(iblock)*ny(iblock)*nz(iblock)
      enddo
      ...
C Solve within a block
      ...
10    tres=0.0
      a(ihot)=100000.0
C$OMP PARALLEL DO SCHEDULE(STATIC) GROUPS(nblock,work)
C$OMP& REDUCTION(+:tres) PRIVATE(res)
      do iblock=1,nblock
          call solve(a(loc(iblock)),nx(iblock),ny(iblock),nz(iblock),res,tol)
          tres=tres+res
      enddo
C$OMP END PARALLEL DO

C Perform inter block interactions
C$OMP PARALLEL DO PRIVATE(val)
      do i=1,nconnect
          val=(a(link(i,1))+a(link(i,2)))/2.0
          a(link(i,1))=val
          a(link(i,2))=val
      enddo
C$OMP END PARALLEL DO
      ...
      if (tres.gt.tol) goto 10
      ...
     subroutine solve(t,nx,ny,nz,tres,tol)
      ...
      res=0.0
C$OMP PARALLEL DO SCHEDULE(STATIC) REDUCTION(+:res)
C$OMP& PRIVATE(i,j,k)
      do k=1,nz
        do j=1,ny
          do i=1,nx
            t(i,j,k)=(told(i,j,k-1)+told(i,j,k+1)+
     +                told(i,j-1,k)+told(i,j+1,k)+
     +                told(i-1,j,k)+told(i+1,j,k)+told(i,j,k)*6.0)/12.0
            res=res+(t(i,j,k)-told(i,j,k))**2
          enddo
        enddo
      enddo
C$OMP END PARALLEL DO
      ...
      end
```

Fig. 2. Source code for mblock benchmark.

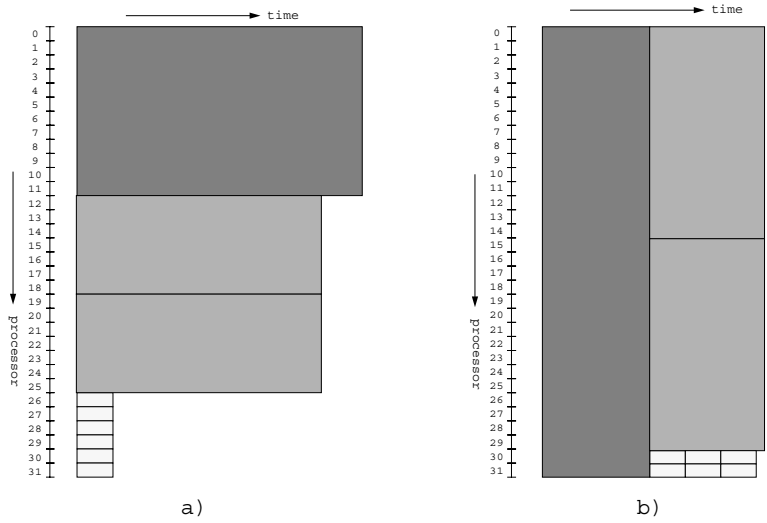


Fig. 3. Groups defined assuming 32 processors and relative weights for the 9 blocks in the multi-block structure {96,48,48,1,1,1,1,1,1}.

multaneously executing threads. In this case, new opportunities for parallel work creation result in the generation of work for all or a restricted set of threads. NthLib provides different mechanisms to spawn parallelism, depending on the hierarchy level in which the application is running. When spawning the deepest (fine grain) level of parallelism, a mechanism based on work–descriptors is available to supply the work to all the threads participating in the parallel region [5]. The mechanism is implemented as efficiently as the ones available in current thread packages. Although efficient, this mechanism does not allow the exploitation of further levels of parallelism. This requires the generation of work using a more costly interface that provides work descriptors with a stack [6]. Owning a stack is necessary for the higher levels of parallelism to spawn an inner level; the stack is used to maintain the context of the higher levels, along with the structures needed to synchronize the parallel execution, while executing the inner levels.

In both cases, kernel threads are assumed to be the execution vehicles that get work from a set of queues where either the user or the compiler has decided to issue these descriptors. Each kernel thread is assumed to have its own queue from where it extracts threads to execute. These queues are identified with a global identifier between 0 and the number of available processors minus 1.

4.2 Thread Identification

The OpenMP extensions proposed in this paper 1) define a mechanism to control the thread distribution among the different levels of parallelism; and 2) export the thread name space to the application in order to specify the work distribution

among threads. The runtime library is in charge of mapping an identifier in the thread name space (identifier between `0` and `nteam-1`, `nteam` being the number of threads in the team the thread belongs to) to one of the above mentioned queues.

The descriptor of a thread contains the context that identifies it within the group of threads it belongs to. The following information is available:

1. `nteam`: indicates the number of threads in the team.
2. `rel_id`: thread identifier relative to the team the thread belongs to: `0`–`nteam-1`.
3. `abs_id`: thread identifier absolute to the number of processors available: `0`–`omp_get_max_threads-1`.
4. `master`: it indicates the absolute identifier of the master of the team.

4.3 Groups

The descriptor also contains information about the number of threads available for nested parallel constructs and the definition of groups for them.

5. `nthreads`: indicates the number of threads that the executing thread has available to spawn nested parallelism.
6. `ngroups`: indicates the number of groups that will be created for the execution of subsequent parallel regions.
7. `where`: integer vector with the identifiers of the threads that will behave as the master of each new group.
8. `howmany`: integer vector specifying the number of threads available within each defined group.

The three last fields are computed when the GROUPS clause is found through the execution of a library function `nthf_compute_groups` by the thread that is defining the groups (master thread).

Figure 4 shows an example of how these fields in the thread descriptor are updated during execution. The left upper part of the figure shows the source code with a group definition. On the left lower part there is the code emitted by the compiler with the appropriate run–time calls to spawn the parallelism expressed by the OpenMP directives. This code consists of a call to routine `nthf_compute_groups` and a loop where the run–time routine for thread creation is invoked (`nthf_create`). Notice that the loop will execute as many iterations as the number of groups that have to be created. In our example 4 groups will be created. On the right upper part of the same figure, we show the descriptor for the thread that finds the `PARALLEL` construct. It is assumed that the thread was already executing in a parallel region where another group definition was performed defining two groups. At that point, 16 threads were available and 8 threads were assigned to it. Its identifier as master thread of the group is `8` (absolute_id) and it has 8 threads reserved to spawn nested parallelism (`nthreads = 8`). Once function `nthf_compute_groups` is called by the master thread, its thread descriptor is updated with the appropriate information. The number of

groups that are going to be created is specified by the `ngroups` field. The thread grouping described by the `GROUPS` clause is used to fill the `where` and `howmany` vectors. The absolute thread identifiers in the `where` vector show the thread reservation for inner levels of parallelism. Those identifiers have been computed adding to the master absolute identifier the thread reservation for each group. When the master thread creates an slave thread, it creates the descriptors and initializes the fields `rel_id`, `absolute_id`, `nthreads` and `nteam`. The value for the `rel_id` is obtained from the variable `nth_p` in the compiler emitted code. The variable appears as a parameter of the call to the thread creation routine nthf_create. The field `absolute_id` is filled copying the value from the corresponding cell in the `where` vector. The values for the fields `nthreads` and `nteam` reflect the fact that the thread belongs to a group. Notice that the numbers in vector `howmany` are copied to the field `nthreads` in the slave descriptors. This will cause that when those slaves reach a parallel construct, they will be able to spawn further parallelism on a limited number of threads. The field `nteam` is set to 4 as indicated by field `ngroups` in the master thread.

For each `GROUPS` syntax described in the previous section, a run–time call is invoked to take the appropriate actions. The case where the programmer is just supplying the number of groups to be created and the vector indicating how many threads have to reserved for each group, is treated in a same manner as in the previous example. The vector `where` is empty, and the run–time computes the threads to be involved in the execution of the parallel region, following the algorithm described in the previous section.

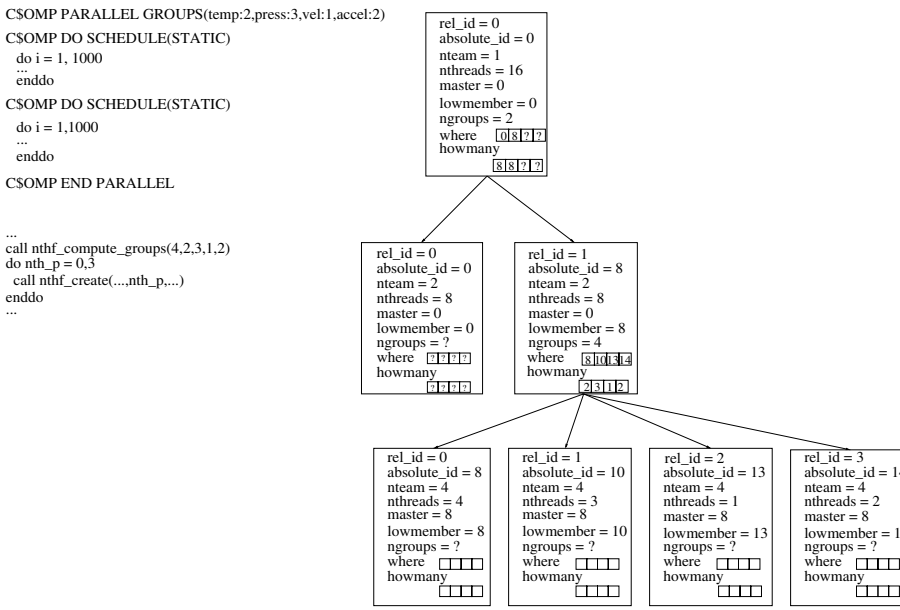


Fig. 4. Run–time example of group definition.

The `BARRIER` construct or the implicit synchronization at the end of each work–sharing construct implies a synchronization point for the members of the group and not in the global execution of the application. In any case, our proposal establishes the possibility to define a `BARRIER` with a local or global behavior (`GLOBAL` clause). In order to have a local or global behavior for the synchronizations points, the run–time library has to know the identifiers of the threads involved in the barrier. The run–time is able to determine which threads are composing a team because of the relative thread identifiers in a team that are always in the range `0` to `nteam-1`. Using the translation mechanism, it is possible to determine the absolute thread identifiers of the threads in a team.

5 Experimental Evaluation

In this section we evaluate the behavior of the *MBLOCK* kernel on a Silicon Graphics Origin2000 system [11] with 64 R10k processors, running at 250 MHz with 4 Mb of secondary cache each. For all compilations we use the native `f77` compiler (flags `-64 -Ofast=ip27 -LNO:prefetch_ahead=1:auto_dist=on`). For an extensive evaluation, please refer to the extended version of this paper [4].

In this section we compare the performance of three OpenMP compliant parallelization strategies and a parallel version which uses the extensions for groups proposed in this paper. The *OmpOuter* version corresponds to a single level parallelization which exploits the existing inter–block parallelism (i.e. the blocks are computed in parallel and the computation inside each block are sequentialized). The *OmpInner* version corresponds to a single level parallelization in which the intra–block parallelism is exploited (i.e. the execution of the blocks is serialized). The *Omp2Levels* version exploits the two levels of parallelism and the *GroupsOmp2Levels* uses the clauses proposed in this paper to define groups. The program is executed with a synthetic input composed of 8 blocks: two with 128x128x128 elements each and the rest with 64x64x64 elements. Notice that the size of the two large blocks is 8 times the size of the small ones.

Figure 5 shows the speed–up of the four parallel versions with respect to the original sequential version. Performance figures for the *OmpOuter* version have been obtained for 1 and 8 processors. Notice that with 8 processors this version achieves an speed–up close to 3 due to the imbalance that appears between the large and small blocks. This version does not benefit from using more processors than the number of blocks in the computation. The speed–up for the *OmpInner* version is reasonable up to 32 processors. The efficiency of the parallelization is considerably reduced when more processors are used to execute the loops in the solver (due to insufficient work granularity). The *Omp2Levels* reports some performance improvement but suffers from the same problem: each loop is executed with all the processors. The *GroupsOmp2Levels* version performs the best when more that 8 processors are used. In this version, the work in the inner level of parallelism is distributed following the groups specification. Therefore, 8 processors are devoted to exploit the outer level of parallelism, and all the processors are distributed among the groups following the proportions dictated by the `work`

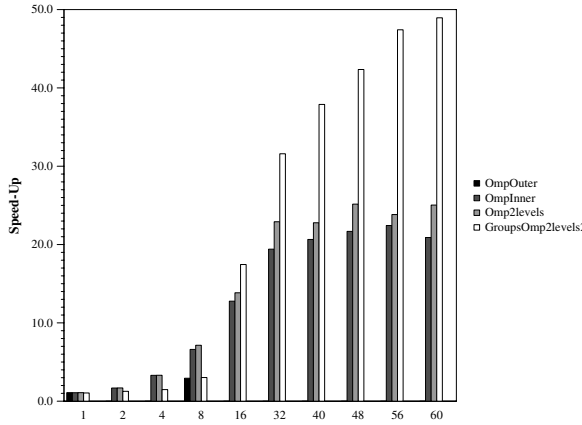


Fig. 5. Speed up for four different parallelization strategies of *MBLOCK*.

vector in Figure 2. Therefore, the inner loops in the *GroupsOmp2Levels* version are executed with a number of processors that allow to continue improving performance while the number of available threads increases.

For instance, notice that when 8 processors are available, the default distribution of threads performed by the runtime when the `GROUPS` clause is executed leads to a high load unbalance (see Figure 3.a). The extensions proposed allow the user to specify some group overlapping which results in a better load balancing (thus reducing the execution time from 32.96 to 15.69 seconds).

6 Conclusions

In this paper we have presented a set of extensions to the OpenMP programming model oriented towards the specification of thread groups in the context of multilevel parallelism exploitation. Although the majority of the current systems only support the exploitation of single–level parallelism around loops, we believe that multi–level parallelism will play an important role in future systems. In order to exploit multiple levels of parallelism, several programming models can be combined (e.g. message passing and OpenMP). We believe that a single programming paradigm should be used and should provide similar performance. The extensions have been implemented in the NanosCompiler and runtime library NthLib. We have analyzed the performance of some applications on a Origin2000 platform. The results show that in these applications, and when the number of processors is high, exploiting multiple levels of parallelism with thread groups results in better work distribution strategies and thus higher speed ups than both the single level version and the multilevel version without groups. For instance, in a generic multi–block *MBLOCK* code, the performance is improved by a factor in the range between 1.5 and 2 (using a synthetic input with a small number of very unbalanced blocks). The speed–up would be higher when

the number of blocks is increased. For other benchmarks [4] the performance is improved by a similar factor.

Acknowledgments

This research has been supported by the Ministry of Education of Spain under contracts TIC98-511 and TIC97-1445CE, the ESPRIT project 21907 NANOS and the CEPBA (European Center for Parallelism of Barcelona).

References

[1] J. Bircsak, P. Craig, R. Crowell, J. Harris, C. A. Nelson and C.D. Offner. Extending OpenMP for NUMA Machines: The Language. In *Workshop on OpenMP Applications and Tools WOMPAT 2000*, San Diego (USA), July 2000

[2] I. Foster, D.R. Kohr, R. Krishnaiyer and A. Choudhary. Double Standards: Bringing Task Parallelism to HPF Via the Message Passing Interface. In *Supercomputing'96*, November 1996.

[3] M. Girkar, M. R. Haghighat, P. Grey, H. Saito, N. Stavrakos and C.D. Polychronopoulos. Illinois-Intel Multithreading Library: Multithreading Support for Intel Architecture–based Multiprocessor Systems. *Intel Technology Journal*, Q1 issue, February 1998.

[4] M. Gonzalez, J. Oliver, X. Martorell, E. Ayguadé, J. Labarta and N. Navarro. OpenMP Extensions for Thread Groups an Their Run-time Support. Extended version of this paper, available at *http://www.ac.upc.es/nanos*, 2000.

[5] X. Martorell, E. Ayguadé, J.I. Navarro, J. Corbalán, M. González and J. Labarta. Thread Fork/join Techniques for Multi–level Parallelism Exploitation in NUMA Multiprocessors. In *13th Int. Conference on Supercomputing ICS'99*, Rhodes (Greece), June 1999.

[6] X. Martorell, J. Labarta, J.I. Navarro, and E. Ayguadé. A library implementation of the nano-threads programming model. In *Euro-Par'96*, August 1996.

[7] D.S. Nikolopoulos, T.S. Papatheodorou, C.D. Polychronopoulos, J. Labarta and E. Ayguade. UPMLib: A Runtime System for Tuning the Memory Performance of OpenMP Programs on Scalable Shared–memory Multiprocessors. In *5th ACM Workshop on Languages, Compilers and Run–time Systems for Scalable Computers* (LCR2000), Univ. of Rocherster NY (USA), May 2000.

[8] OpenMP Organization. OpenMP Fortran Application Interface, v. 2.0, www.openmp.org, June 2000.

[9] C.D. Polychronopoulos, M. Girkar, M.R. Haghighat, C.L. Lee, B. Leung, and D. Schouten. Parafrase–2: An environment for parallelizing, partitioning, and scheduling programs on multiprocessors. *International Journal of High Speed Computing*, 1(1), 1989.

[10] S. Shah, G. Haab, P. Petersen and J. Throop. Flexible Control Structures for Parallelism in OpenMP. In *1st European Workshop on OpenMP, Lund (Sweden), September 1999.*

[11] Silicon Graphics Computer Systems SGI. Origin 200 and Origin 2000 Technical Report, 1996.

[12] Silicon Graphics Inc. MIPSPro Fortran90 Commands and Directives Reference Manual, 1996.

[13] SPEC Organization. *The Standard Performance Evaluation Corporation*, www.spec.org.

Compiling Data Intensive Applications with Spatial Coordinates*

Renato Ferreira[1], Gagan Agrawal[2], Ruoming Jin[2], and Joel Saltz[1]

[1] Department of Computer Science
University of Maryland, College Park MD 20742
{renato,saltz}@cs.umd.edu
[2] Department of Computer and Information Sciences
University of Delaware, Newark DE 19716
{agrawal,jrm}@cis.udel.edu

Abstract. Processing and analyzing large volumes of data plays an increasingly important role in many domains of scientific research. We are developing a compiler which processes data intensive applications written in a dialect of Java and compiles them for efficient execution on cluster of workstations or distributed memory machines.
In this paper, we focus on data intensive applications with two important properties: 1) data elements have *spatial coordinates* associated with them and the distribution of the data is not regular with respect to these coordinates, and 2) the application processes only a subset of the available data on the basis of spatial coordinates. These applications arise in many domains like satellite data-processing and medical imaging. We present a general compilation and execution strategy for this class of applications which achieves high locality in disk accesses. We then present a technique for hoisting conditionals which further improves efficiency in execution of such compiled codes.
Our preliminary experimental results show that the performance from our proposed execution strategy is nearly two orders of magnitude better than a naive strategy. Further, up to 30% improvement in performance is observed by applying the technique for hoisting conditionals.

1 Introduction

Analysis and processing of very large multi-dimensional scientific datasets (i.e. where data items are associated with points in a multidimensional attribute space) is an important component of science and engineering. An increasing number of applications make use of very large multidimensional datasets. Examples of such datasets include raw and processed sensor data from satellites [21], output from hydrodynamics and chemical transport simulations [17], and archives of medical images [1].

* This work was supported by NSF grant ACR-9982087, NSF CAREER award ACI-9733520, and NSF grant CCR-9808522.

S.P. Midkiff et al. (Eds.): LCPC 2000, LNCS 2017, pp. 339–354, 2001.

We are developing a compiler which processes data intensive applications written in a dialect of Java and compiles them for efficient execution on cluster of workstations or distributed memory machines [2, 3, 12]. Our chosen dialect of Java includes data-parallel extensions for specifying collection of objects, a parallel for loop, and reduction variables. We extract a set of functions (including subscript functions used for accessing left-hand-side and right-hand-side object collections, aggregation functions, and range functions) from the given data intensive loop by using the technique of interprocedural program slicing. Data partitioning, retrieval, and processing is performed by utilizing an existing runtime system called *Active Data Repository* [7, 8, 9].

Data intensive computations from a number of domains share two important charactestics. First, the input data elements have *spatial coordinates* associated with them. For example, the pixels in the satellite data processing application have latitude and longitude values associated with them [21]. The pixels in a multi-resolution virtual microscope image have the x and y coordinates of the image associated with them [1]. Moreover, the actual layout of the data is not regular in terms of the spatial coordinates. Second, the application processes only a subset of the available data, on the basis of spatial coordinates. For example, in the satellite data processing application, only the pixels within a bounding box specified by latitudes and longitudes may be processed. In the virtual microscope, again, only the pixels within a rectangular region may be processed.

In this paper, we present a compilation and execution strategy for this class of data intensive applications. In our execution strategy, the right-hand-side data is read one disk block at a time. Once this data is brought into the memory, corresponding iterations are performed. A compiler determined mapping from right-hand-side elements to iteration number and left-hand-side element is used for this purpose. The resulting code has a very high locality in disk accesses, but also has extra computation and evaluation of conditionals in every iteration. We present an analysis framework for code motion of conditionals which further improves efficiency in execution of such codes.

The rest of the paper is organized as follows. The dialect of Java we target and an example of data intensive application with spatial coordinates is presented in Section 2. Basic compiler technique and the loop execution model are presented in Section 3. The technique for code motion of conditionals is presented in Section 4. Experimental results are presented in Section 5. We compare our work with related research efforts in Section 6 and conclude in Section 7.

2 A Data Intensive Application with Spatial Coordinates

2.1 Data-Parallel Constructs

We borrow two concepts from object-oriented parallel systems like Titanium, HPC++, or Concurrent Aggregates [11, 23].

- *Domains* and *Rectdomains* are collections of objects of the same type. *Rectdomains* have a stricter definition, in the sense that each object belonging

to such a collection has a *coordinate* associated with it that belongs to a pre-specified rectilinear section.
- The *foreach* loop, which iterates over objects in a domain or rectdomain, and has the property that the order of iterations does not influence the result of the associated computations.

We introduce a Java interface called *Reducinterface*. Any object of any class implementing this interface acts as a *reduction variable* [14]. A reduction variable has the property that it can only be updated inside a *foreach* loop by a series of operations that are associative and commutative. Furthermore, the intermediate value of the reduction variable may not be used within the loop, except for self-updates.

2.2 Satellite Data Processing Example

```
Interface Reducinterface {
  // Any object of any class implementing
  // this interface is a reduction variable
}
public class pixel {
  short bands[5] ;
  short geo[2] ;
}
public class block {
  pixel bands[204*204] ;
  pixel getData(Point[2] p) {
    { * Search for the (lat, long) on geo data *}
    { * Return the pixel if it exists *}
    { * Else return null *}
  }
}
public class SatData {
  public class SatOrigData {
    block[1d] satorigdata ;
    void SatOrigData(RectDomain[1] InputDomain) {
      satorigdata = new block[InputDomain] ;
    }
    pixel getData(Point[3] q) {
      Point[1] time = (q.get(0));
      Point[2] p = (q.get(1), q.get(2));
      return satorigdata[time].getData(p) ;
    }
  }
  void SatData(RectDomain[1] InputDomain) {
    SatOrigData(InputDomain) ;
  }
  pixel getData(Point[3] q) {
    return SatOrigData(q) ;
  }
}
```

```
public class Image
      implements Reducinterface {
  void Accumulate(pixel input ) {
    { * Accumulation function *}
  }
}

public class Satellite {
  Point[1] LoEnd = ...
  Point[1] HiEnd = ...
  SatData satdata ;
  RectDomain[1] InputDomain = [LoEnd : HiEnd];
  satdata.SatData(InputDomain);
  public static void main(int[] args) {
    Point[1] lowtime = (args[0]);
    Point[1] hightime = (args[1]);
    RectDomain[1] TimeDomain = [lowtime : hightime];
    Point[2] lowend = (args[2], args[4]);
    Point[2] highend = (args[3], args[5]);
    Rectdomain[2] OutputDomain = [lowend : highend];
    Point[3] low = (args[0], args[2], args[4]);
    Point[3] high = (args[1], args[3], args[5]);
    Rectdomain[3] AbsDomain = [low : high];
    Image[2d] Output = new Image[OutputDomain];

    foreach (Point[3] q in AbsDomain) {
      if (pixel val = satdata.getData(q))
        Point[2] p = (q.get(1), q.get(2));
        Output[p].Accumulate(val);
    }
  }
}
```

Fig. 1. A Satellite Data-Processing Code

In Figure 1, we show the essential structure associated with the satellite data processing application [9, 10]. The satellites generating the datasets contain sensors for five different bands. The measurements produced by the satellite are short values (16 bits) for each band. As the satellite orbits the earth, the sensors sweep the surface building scan lines of 408 measurements each. Our data file consists of blocks of 204 half scan lines, which means that each block is a 204×204

array with 5 short integers per element. Latitude and longitude are also stored within the disk block for each measure.

The typical computation on this satellite data is as follows. A portion of earth is specified through latitudes and longitudes of end points. A time range (typically 10 days to one year) is also specified. For any point on the earth within the specified area, all available pixels within that time-period are scanned and the *best* value is determined. Typical criteria for finding the *best* value is cloudiness on that day, with the least cloudy image being the best. The best pixel over each point within the area is used to produce a composite image. This composite image is used by researchers to study a number of properties, like deforestation over time, pollution over different areas, etc [21].

The main source of irregularity in this dataset and computation comes because the earth is spherical, whereas the satellite sees the area of earth it is above as a rectangular grid. Thus, the translation from the rectangular area that the satellite has captured in a given band to latitudes and longitudes is not straight forward.

We next explain the data-parallel Java code representing such computation (Figure 1). The class `block` represents the data captured in each time-unit by the satellite. This class has one function (`getData`) which takes a (latitude, longitude) pair and sees if there is any pixel in the given block for that location. If so, it returns that pixel. The class `SatData` is the interface to the input dataset visible to the programmer writing the main execution loop. Through its access function `getData`, this class gives the view that a 3-dimensional grid of pixels is available. Encapsulated inside this class is the class `SatOrigData`, which stores the data as a 1-dimensional array of `bands`. The constructor and the access function of the class `SatData` invoke the constructor and the access function, respectively of the class `SatOrigData`.

The main processing function takes 6 command line arguments as the input. The first two specify a time range over which the processing is performed. The next four are the latitudes and longitudes for the two end-points of the rectangular output desired. We consider an abstract 3-dimensional rectangular grid, with time, latitude, and longitude as the three axes. This grid is abstract, because pixels actually exist for only a small fraction of all the points in this grid. However, the high-level code just iterates over this grid in the `foreach` loop. For each point q in the grid, which is a `(time, lat, long)` tuple, we examine if the block `SatData[time]` has any pixel. If such a pixel exists, it is used for performing a reduction operation on the object `Output[(lat,long)]`.

The code, as specified above, can lead to very inefficient execution for at least two reasons. First, if the look-up is performed for every point in the abstract grid, it will have a very high overhead. Second, if the order of iterations in the loop is not carefully chosen, the locality can be very poor.

3 Compilation Model for Applications with Spatial Coordinates

The main challenge in executing a data intensive loop comes from the fact that the amount of data accessed in the loop exceeds the main memory. While the virtual memory support can be used for correct execution, it leads to very poor performance. Therefore, it is compiler's responsibility to perform memory management, i.e., determine which portions of output and input collections are in the main memory during a particular stage of the computation.

Based upon the experiences from data intensive applications and developing runtime support for them [9, 8], the basic code execution scheme we use is as follows. The output data-structure is divided into tiles, such that each tile fits into the main memory. The input dataset is read disk block at a time. This is because the disks provide the highest bandwidth and incur lowest overhead while accessing all data from a single disk block. Once an input disk block is brought into main memory, all iterations of the loop which read from this disk block and update an element from the current tile are performed. A tile from the output data-structure is never allocated more than once, but a particular disk block may be read to contribute to the multiple output tiles.

To facilitate the execution of loops in this fashion, our compiler first performs loop fission. For each resulting loop after loop fission, it uses the runtime system called Active Data Repository (ADR) developed at University of Maryland [7, 8] to retrieve data and stage the computation.

3.1 Loop Fission

Consider any loop. For the purpose of our discussion, collections of objects whose elements are modified in the loop are referred to as *left hand side* or LHS collections, and the collections whose elements are only read in the loop are considered as *right hand side* or RHS collections.

If multiple distinct subscript functions are used to access the right-hand-side (RHS) collections and left-hand-side (LHS) collections and these subscript functions are not known at compile-time, tiling output and managing disk accesses while maintaining high reuse and locality is going to be difficult. Particularly, the current implementation of ADR runtime support requires only one distinct RHS subscript function and only one distinct LHS subscript function. Therefore, we perform *loop fission* to divide the original loop into a set of loops, such that all LHS collections in any new loop are accessed with the same subscript function and all RHS collections are also accessed with the same subscript function.

The terminology presented here is illustrated by the example loop in Figure 2. The range (domain) over which the iterator iterates is denoted by $\mathcal{R}$. Let there be n RHS collection of objects read in this loop, which are denoted by $I_1, \ldots, I_n$. Similarly, let the LHS collections written in the loop be denoted by $O_1, \ldots, O_m$. Further, we denote the subscript function used for accessing right hand side collections by $\mathcal{S}_R$ and the subscript function used for accessing left hand side collections by $\mathcal{S}_L$.

$$
\begin{array}{l}
\text{foreach(r } \in \mathcal{R}) \text{ \{} \\
\quad O_1[\mathcal{S}_L(r)] \quad op_1 = \quad \mathcal{A}_1(I_1[\mathcal{S}_R(r)], \ldots, I_n[\mathcal{S}_R(r)]) \\
\quad \ldots \\
\quad O_m[\mathcal{S}_L(r)] \quad op_m = \quad \mathcal{A}_m(I_1[\mathcal{S}_R(r)], \ldots, I_n[\mathcal{S}_R(r)]) \\
\text{\}}
\end{array}
$$

Fig. 2. A Loop In Canonical Form After Loop Fission

Given a point r in the range for the loop, elements $\mathcal{S}_L(r)$ of the LHS collections are updated using one or more of the values $I_1[\mathcal{S}_R(r)], \ldots, I_n[\mathcal{S}_R(r)]$, and other scalar values in the program. We denote by $\mathcal{A}_i$ the function used for updating LHS collection O_i.

Consider any element of a RHS or LHS collection. Its abstract address is referred to as its *l-value* and its actual value is referred to as its *r-value*.

3.2 Extracting Information

Our compiler extracts the following information from a given data-parallel loop after loop fission.

1. We extract the range $\mathcal{R}$ of the loop by examining the domain over which the loop iterates.

2. We extract the accumulation functions used to update the LHS collections in the loop. For extracting the function $\mathcal{A}_i$, we look at the statement in the loop where the LHS collection O_i is modified. We use interprocedural program slicing [24], with this program point and the value of the element modified as the slicing criterion.

3. For a given element of the RHS collection (with its l-value and r-value), we determine the iteration(s) of the loop in which it can be accessed. Consider the example code in Figure 1. Consider a pixel with the l-value $< t, num >$, i.e., it is the num^{th} pixel in the `SatData[t]` block. Suppose its r-value is $< c1, c2, c3, c4, c5, lat, long >$. From the code, it can be determined that this element will be and can only be accessed in the iteration $< t, lat, long >$.

Formally, we denote it as

$$IterVal(e \; = \; (< t, num >; < c1, c2, c3, c4, c5, lat, long >)) \; = < t, lat, long >$$

While this information can be extracted easily from the loop in Figure 1, computing such information from a loop is a hard problem in general and a subject of future research.

4. For a given element of the RHS collection (with its l-value and r-value), we determine the l-value of the LHS element which is updated using its value. For example, for the loop in Figure 1, for a RHS element with l-value $< t, num >$, and r-value $< c1, c2, c3, c4, c5, lat, long >$, the l-value of the LHS element updated using this is $< lat, long >$.

Formally, we denote it as

$$OutVal(e = (< t, num >; < c1, c2, c3, c4, c5, lat, long >)) = < lat, long >$$

This can be extracted by composing the subscript function for the LHS collections with the function $IterVal$.

3.3 Storing Spatial Information

To facilitate decisions about which disk blocks have elements that can contribute to a particular tile, the system stores additional information about each disk block in the *meta-data* associated with the dataset. Consider a disk block b which contains a number of elements. Explicitly storing the spatial coordinates associated with each of the elements as part of the meta-data will require very large additional storage and is clearly not practical. Instead, the range of the spatial coordinates of the elements in a disk block is described by a *bounding box*.

A bounding box for a disk block is the minimal rectilinear section (described by the coordinates of the two extreme end-points) such that the spatial coordinates of each element in the disk block falls within this rectilinear section.

Such bounding boxes can be computed and stored with the meta-data during a preprocessing phase when the data is distributed between the processors and disks.

3.4 Loop Planning

The following decisions need to made during the loop planning phase:

- The size of LHS collection required on each processor and how it is tiled.
- The set of disk blocks from the RHS collection which need to read for each tile on each processor.

The static declarations on the LHS collection can be used to decide the total size of the output required. Not all elements of this LHS space need to be updated on all processors. However, in the absence of any other analysis, we can simply replicate the LHS collections on all processors and perform global reduction after local reductions on all processors have been completed.

The memory requirements of the replicated LHS space is typically higher than the available memory on each processor. Therefore, we need to divide the replicated LHS buffer into chunks that can be allocated on each processor's main memory. We have so far used only a very simple strip mining strategy. We query the run-time system to determine the available memory that can be allocated on a given processor. Then, we divide the LHS space into blocks of that size. Formally, we divide the LHS domain into a set of smaller domains (called *strips*) $\{S_1, S_2, \ldots, S_r\}$. Since each of the LHS collection of objects in the loop is accessed through the same subscript function, same strip mining is used for each of them.

We next decide which disk blocks need to be read for performing the updates on a given tile. A LHS tile is allocated only once. If elements of a particular disk block are required for updating multiple tiles, this disk block is read more than once.

On a processor j and for a given RHS collection I_i, the bounding box of spatial coordinates for the disk block b_{ijk} is denoted by $BB(b_{ijk})$. On a given processor j and for a given LHS strip l, the set of disk blocks which need to read is denoted by L_{jl}. These sets are computed as follows:

$$L_{jl} \;=\; \{k \,|\, (BB(b_{ijk}) \cap S_l) \;\neq\; \phi\}$$

3.5 Loop Execution

The generic loop execution strategy is shown in Figure 3. The LHS tiles are allocated one at a time. For each LHS tile S_l, the RHS disk blocks from the set Ljl are read successively. For each element e from a disk block, we need to determine:

1. If this element is accessed in one of the iterations of the loop. If so, we need to know which iteration it is.
2. The LHS element that this RHS element will contribute to, and if this LHS element belongs to the tile currently being processed.

We use the function $IterVal(e)$ computed earlier to map the RHS element to the iteration number and the function $OutVal(e)$ to map a RHS element to the LHS element.

```
For each LHS strip S_l:
    Execute on each Processor P_j:
        Allocate and initialize strip S_l for O_1, ..., O_m
        Foreach k ∈ L_jl
            Read blocks b_ijk, i = 1, ..., n from disks
            Foreach element e in b_ijk
                i = IterVal(e)
                o = OutVal(e)
                If (i ∈ R) ∧ (o ∈ S_l)
                    Evaluate functions A_1, ..., A_m
    Global reduction to finalize the values for S_l
```

Fig. 3. Loop Execution on Each Processor

Though the above execution sequence achieves very high locality in disk accesses, it performs considerably higher computation than the original loop. This is because the mapping from the element e to the iteration number and the

LHS element needs to be evaluated and intersected with the range and tile in every iteration. In the next section, we describe a technique for hoisting conditional statements which eliminates almost all of the additional computation associated with the code shown in Figure 3.

4 Code Motion for Conditional Statements

In this section, we present a technique which eliminates redundant conditional statements and merges the conditionals with loop headers or other conditionals wherever possible.

Program Representation We consider only structured control flow, with `if` statements and `loops`. Within each control level, the *definitions* and *uses* of variables are linked together with *def-use* links. Since we are looking at *def-use* links within a single control level, each use of a variable is linked with at most one definition.

Candidates for Code Motion The candidates for code motion in our framework are if statements. One common restriction in code hoisting frameworks like Partial Redundancy Elimination (PRE) [18] and the existing work on code hoisting for conditionals [5, 20] is that syntactically different expressions which may have the same value are considered different candidates. We remove this restriction by following *def-use* links and considering multiple *views* of the expressions in conditionals and loops.

To motivate this, consider two conditional statements, one of which is enclosed in another. Let the outer condition be `x > 2` and let the inner condition be `y > 3`. Syntactically, these are different expressions and therefore, it appears that both of them must be evaluated. However, by seeing the definitions of `x` and `y` that reach these conditional statements, we may be able to relate them. Suppose that `x` is defined as `x = z -3` and `y` is defined as `y = z - 2`. By substituting the definitions of `x` and `y` in the expressions, the conditions become `z - 3 > 2` and `z - 2 > 3`, which are identical.

We define a view of a candidate for code motion as follows. Starting from the conditional, we do a forward substitution of the definition of zero or more variables occurring in the conditional. This process may be repeated if new variables are introduced in the expression after forward substitution is performed. By performing every distinct subset of the set of the all possible forward substitutions, a distinct view of the candidate is obtained. Since we are considering *def-use* within a single control level, there is at most one reaching definition of a use of a variable. This significantly simplifies the forward substitution process.

Views of a loop header are created in a similar fashion. Forward substitution is not done for any variable, including the induction variable, which may be modified in the loop.

Phase I: Downward Propagation In the first phase, we propagate *dominating constraints* down the levels, and eliminate any conditional which may be redundant. Consider any loop header or conditional statement. The range of the loop

or the condition imposes a constraint for values of variables or expression in the control blocks enclosed within. As described previously, we compute the different views of the constraints by performing a different set of forward substitutions. By composing the different views of the loop headers and conditionals statements, we get different views of the *dominating constraints*.

Consider any conditional statement for which the different views of the dominating constraints are available. By comparing the different views of this conditional with the different views of dominating constraints, we determine if this conditional is redundant. A redundant conditional is simple removed and the statements enclosed inside it are merged with the control block in which the conditional statement was initially placed.

Phase II: Upward Propagation After the redundant conditionals have been eliminated, we consider if any of the conditionals can be folded into any of the conditionals or loops enclosing it. The following steps are used in the process. We compute all the views of the conditional which is the candidate for hoisting.

Consider any statement which dominates the conditional. We compute two terms: *anticipability* of the candidate at that statement, and *anticipable views*. The candidate is anticipable at its original location and all views of the candidate computed originally are anticipable views.

The candidate is considered anticipable at the beginning of a statement if it is anticipable at the end of the statement and any assignment made in the statement is not live at the end of the conditional. This reason behind this condition is as follows. A statement can be folded inside the conditional only if the values computed in it are used inside the conditional only. To compute the set of anticipable views at the beginning of a statement, we consider two cases:

Case 1. If the variable assigned in the statement does not influence the expression inside the conditional, all the views anticipable at the end of the statement are anticipable at the beginning of the statement.

Case 2. Otherwise, let the variable assigned in this statement be v. From the set of views anticipable at the end of the statement, we exclude the views in which the definition of v at this statement is not forward substituted.

Now, consider any conditional or loop which encloses the original candidate for placement, and let this candidate be anticipable at the beginning of the first statement enclosed in the conditional or loop. We compare all the views of this conditional or loop against all anticipable views of the candidate for placement. If either the left-hand-side or the right-hand-side of the expression are identical or separated by a constant, we fold in the candidate into this conditional or loop.

5 Experimental Results

In this section we present results from the experiments we conducted to demonstrate the effectiveness of our execution model. We also present preliminary evidence of the benefits from conditional hoisting optimization. We used a cluster

of 400 MHz Pentium II based computers connected by a gigabit switch. Each node has 256 MB of main memory and 18 GB of local disk. We ran experiments using 1, 2, 4 and 8 nodes of the cluster.

The application we use for our experiments closely matches the code presented in Figure 1 and is referred to as `sat` in this section. We generated code for the satellite template the compilation strategy described in this paper. This version is referred to as the `sat-comp` version. We also had access to a version of the code developed by customizing the runtime system Active Data Repository (ADR) by hand. This version is referred to as the `sat-manual` version. We further created two more versions. The version `sat-opt` has the code hoisting optimization applied by hand. The version `sat-naive` is created to measure the performance using a naive compilation strategy. This naive compilation strategy is based upon an execution model we used in our earlier work for the regular codes [12].

The data for the satellite application we used is approximately 2.7 gigabytes. This corresponds to part of the data generated over a period of 2 months, and only contains data for bands 1 and 2, out of the 5 available for the particular satellite. The data spawns the entire surface of the planet over that period of time. The processing performed by the application consists of generating a composite image of the earth approximately from latitude 0 to latitude 78 north and from longitude 0 to longitude 78 east over the entire 2 month period. This involves composing over about 1/8 of the available data and represents an area that covers almost all of Europe, northern Africa, the Middle East and almost half of Asia. The output of the application is a 313×313 picture of the surface for the corresponding region.

Figure 4 compares three versions of `sat`: `sat-comp`, `sat-opt` and `sat-manual`. The difference between the execution times of `sat-comp` and `sat-opt` shows the impact of eliminating redundant conditionals and hoisting others. The improvement in the execution time by performing this optimization is consistently between 29% and 31% on 1, 2, 4, and 8 processor configurations. This is a significant improvement considering that a relatively simple optimization is applied.

The `sat-opt` version is the best performance we expect from the compilation technique we have. Comparing the execution times from this version against a hand generated version show us how close the compiler generated version can be to hand customization. The versions `sat-opt` and `sat-manual` are significantly different in terms of the implementation. The hand customized code has been carefully optimized to avoid all unnecessary computations by only traversing the parts of each disk block that are effectively part of the output. The compiler generated code will traverse all the points of the data blocks. However, our proposed optimization is effective in hoisting the conditionals within the loop to the outside, therefore minimizing that extra computation. Our experiments show that after optimizations, the compiler is consistently around 18 to 20% slower than the hand customized version.

Figure 5 shows the execution time if the execution strategy used for regular data intensive applications is applied for this code (the `sat-naive` version).

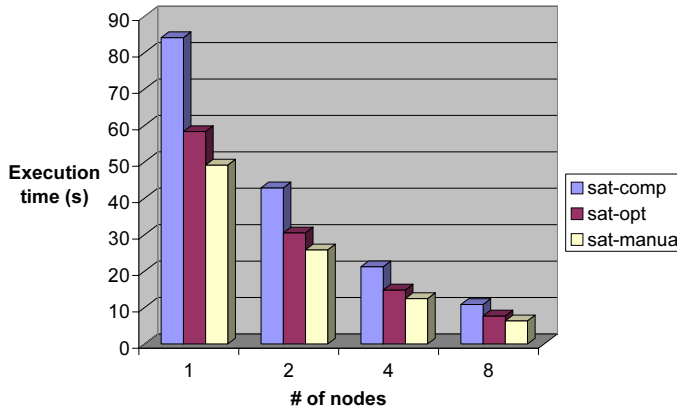


Fig. 4. Satellite: Comparing `sat-comp`, `sat-opt`, and `sat-manual` versions

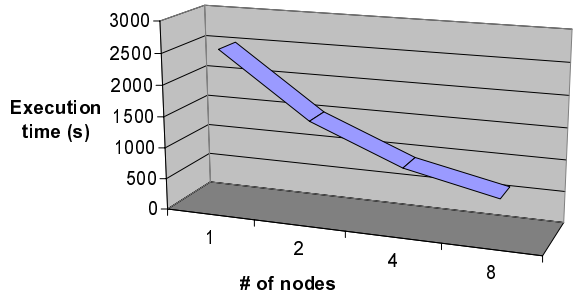


Fig. 5. Performance of the naive implementation of the satellite template

In this strategy, each input disk block is read like the strategy proposed in this paper. However, rather than iterating over the elements and mapping each element to an iteration of the loop, the bounding box of the disk block is mapped into a portion of the iteration space. Then the code is executed for this iteration space.

As can be seen from Figure 5, the performance of this version is very poor and the execution times are almost two orders of magnitudes higher than the other versions. The reason is that this code iterates over a very large iteration space for each disk block and checks whether or not there is input data for each point in the domain. Due to the nature of the problem, the blocks towards the poles of the planet will spawn a very big area over the globe, which leads to a huge number of iterations. Clearly, the execution strategy for regular codes is not applicable for an application like `sat`.

6 Related Work

The work presented in this paper is part of our continuing work on compiling data intensive applications [12, 3]. Our previous work did not handle applications with sparse accesses and datasets with spatial coordinates. This paper has two main original contributions beyond our previous publications. First, we have presented a new execution strategy for sparse accesses which is significantly different from the execution strategy for dense accesses presented previously [12]. Second, we have presented a technique for code motion of conditionals.

Our code execution model has several similarities to the *data-centric* locality transformations proposed by Pingali *et al.* [16]. We fetch a data-chunk or shackle from a lower level in the memory hierarchy and perform the iterations from the loop which use elements from this data-chunk. We have focused on applications where no computations may be performed as part of many iterations from the original loop. So, instead of following the same loop pattern and inserting conditionals to see if the data accessed in the iteration belongs to the current data-chunk, we compute a mapping function from elements to iterations and iterate over the data elements. To facilitate this, we simplify the problem by performing loop fission, so that all collections on the right-hand-side are accessed with the same subscript function.

Several other researchers have focused on removing fully or partially redundant conditionals from code. Mueller and Whalley have proposed analysis within a single loop-nest [20] and Bodik, Gupta, and Soffa perform demand-driven interprocedural analysis [5]. Our method is more aggressive in the sense we associate the definitions of variables involved in the conditionals and loop headers. This allows us to consider conditionals that are syntactically different. Our method is also more restricted than these previously proposed approaches in the sense that we do not consider partially redundant conditionals and do not restructure the control flow to eliminate more conditionals. Many other researchers have presented techniques to detect the equality or implies relationship between conditionals, which are powerful enough to take care of syntactic differences between expressions [4, 13, 25].

Our work on providing high-level support for data intensive computing can be considered as developing an out-of-core Java compiler. Compiler optimizations for improving I/O accesses have been considered by several projects. The PASSION project at Northwestern University has considered several different optimizations for improving locality in out-of-core applications [6, 15]. Some of these optimizations have also been implemented as part of the Fortran D compilation system's support for out-of-core applications [22]. Mowry *et al.* have shown how a compiler can generate prefetching hints for improving the performance of a virtual memory system [19]. These projects have concentrated on relatively simple stencil computations written in Fortran. Besides the use of an object-oriented language, our work is significantly different in the class of applications we focus on. Our technique for loop execution is particularly targeted towards reduction operations, whereas previous work has concentrated on stencil computations.

7 Conclusions and Future Work

Processing and analyzing large volumes of data plays an increasingly important role in many domains of scientific research. We have developed a compiler which processes data intensive applications written in a dialect of Java and compiles them for efficient execution on cluster of workstations or distributed memory machines. In this paper, we focus on data intensive applications with two important properties: 1) data elements have *spatial coordinates* associated with them and the distribution of the data is not regular with respect to these coordinates, and 2) the application processes only a subset of the available data on the basis of spatial coordinates. We have presented a general compilation model for this class of applications which achieves high locality in disk accesses. We have also outlined a technique for hoisting conditionals and removing redundant conditionals that further achieves efficiency in execution of such compiled codes.

Our preliminary experimental results show that the performance from our proposed execution strategy is nearly two orders of magnitude better than a naive strategy. Further, up to 30% improvement in performance is observed by applying the technique for hoisting conditionals.

References

[1] Asmara Afework, Michael D. Beynon, Fabian Bustamante, Angelo Demarzo, Renato Ferreira, Robert Miller, Mark Silberman, Joel Saltz, Alan Sussman, and Hubert Tsang. Digital dynamic telepathology - the Virtual Microscope. In *Proceedings of the 1998 AMIA Annual Fall Symposium*. American Medical Informatics Association, November 1998.

[2] Gagan Agrawal, Renato Ferreira, Joel Saltz, and Ruoming Jin. High-level programming methodologies for data intensive computing. In *Proceedings of the Fifth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, May 2000.

[3] Gagan Agrawal, Renato Ferriera, and Joel Saltz. Language extensions and compilation techniques for data intensive computations. In *Proceedings of Workshop on Compilers for Parallel Computing*, January 2000.

[4] W. Blume and R. Eigenmann. Demand-driven, symbolic range propagation. *Proceedings of the 8th Workshop on Languages and Compilers for Parallel Computing*, pages 141–160, August 1995.

[5] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Interprocedural conditional branch elimination. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 146–158. ACM Press, June 1997.

[6] R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel, and M. Paleczny. A model and compilation strategy for out-of-core data parallel programs. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 1–10. ACM Press, July 1995. ACM SIGPLAN Notices, Vol. 30, No. 8.

[7] C. Chang, A. Acharya, A. Sussman, and J. Saltz. T2: A customizable parallel database for multi-dimensional data. *ACM SIGMOD Record*, 27(1):58–66, March 1998.

[8] Chialin Chang, Renato Ferreira, Alan Sussman, and Joel Saltz. Infrastructure for building parallel database systems for multi-dimensional data. In *Proceedings of the Second Merged IPPS/SPDP (13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing)*. IEEE Computer Society Press, April 1999.

[9] Chialin Chang, Bongki Moon, Anurag Acharya, Carter Shock, Alan Sussman, and Joel Saltz. Titan: A high performance remote-sensing database. In *Proceedings of the 1997 International Conference on Data Engineering*, pages 375–384. IEEE Computer Society Press, April 1997.

[10] Chialin Chang, Alan Sussman, and Joel Saltz. Scheduling in a high performance remote-sensing data server. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, March 1997.

[11] A.A. Chien and W.J. Dally. Concurrent aggregates (CA). In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 187–196. ACM Press, March 1990.

[12] Renato Ferriera, Gagan Agrawal, and Joel Saltz. Compiling object-oriented data intensive computations. In *Proceedings of the 2000 International Conference on Supercomputing*, May 2000.

[13] M. Gupta, S. Mukhopadhyay, and N. Sinha. Automatic parallelization of recursive procedures. In *Proceedings of Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 1999.

[14] High Performance Fortran Forum. Hpf language specification, version 2.0. Available from *http://www.crpc.rice.edu/HPFF/versions/hpf2/files/hpf-v20.ps.gz*, January 1997.

[15] M. Kandemir, J. Ramanujam, and A. Choudhary. Improving the performance of out-of-core computations. In *Proceedings of International Conference on Parallel Processing*, August 1997.

[16] Induprakas Kodukula, Nawaaz Ahmed, and Keshav Pingali. Data-centric multi-level blocking. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 346–357, June 1997.

[17] Tahsin M. Kurc, Alan Sussman, and Joel Saltz. Coupling multiple simulations via a high performance customizable database system. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, March 1999.

[18] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[19] Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic compiler-inserted i/o prefetching for out-of-core applications. In *Proceedings of the Second Symposium on Operating Systems Design and plementation (OSDI '96)*, Nov 1996.

[20] Frank Mueller and David B. Whalley. Avoiding conditional branches by code replication. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 56–66, La Jolla, California, 18–21 June 1995. *SIGPLAN Notices* 30(6), June 1995.

[21] NASA Goddard Distributed Active Archive Center (DAAC). Advanced Very High Resolution Radiometer Global Area Coverage (AVHRR GAC) data. *http://daac.gsfc.nasa.gov/CAMPAIGN_DOCS/ LAND_BIO/origins.html*.

[22] M. Paleczny, K. Kennedy, and C. Koelbel. Compiler support for out-of-core arrays on parallel machines. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 110–118. IEEE Computer Society Press, February 1995.

[23] John Plevyak and Andrew A. Chien. Precise concrete type inference for object-oriented languages. In *Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '94)*, pages 324–340, October 1994.

[24] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[25] Peng Tu and David Padua. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *Proceedings of the 1995 International Conference on Supercomputing*, pages 414–423, 1995.

Efficient Dynamic Local Enumeration for HPF

Will Denissen and Henk J. Sips

Delft University of Technology, the Netherlands
sips@its.tudelft.nl

Abstract. In translating HPF programs, a compiler has to generate *local iteration and communication sets*. Apart from local enumeration, local storage compression is an issue, because in HPF array alignment functions can introduce local storage inefficiencies. Storage compression, however, may not lead to serious performance penalties. A problem in semi-automatic translation is that a compiler should generate efficient code in all cases the user may expect efficient translation (no surprises). However, in current compilers this turns out to be not always true. A major cause for this inefficiencies is that compilers use the same fixed enumeration scheme in all cases. In this paper, we present an efficient dynamic local enumeration method, which always selects the optimal solution at run-time and has no need for code duplication. The method is compared with the PGI and the Adaptor compiler.

Dynamic Selection of Enumeration Orders

Once a data mapping function is given in an HPF program, we know exactly which element of an array is owned by which processor. However, the storage of all elements owned by a single processor still needs to be determined. When a compiler is to generate code for local iteration or communication sets, it needs to enumerate the local elements efficiently. Efficient in the sense that only a small overhead is allowed compared to the work inside the iteration space. Because in both phases local elements need to be referenced, storage and enumeration are closely linked to each other. An overview of the basic schemes for local storage and local enumeration is given in [1].

In a *cyclic(m)* distribution the template data elements are distributed in blocks of size m in a round robin fashion. The relation between an array index i and the row, column, and processor tuple (r, c, p) is given by the position equation [1]. To avoid inefficient memory storage, local storage is compressed by removing unused template elements. There are various compression techniques, each with their own compression factor for rows (Δr) and columns (Δc). For a *cyclic(m)* distribution, the original (normalized) volume assignment is transformed into a two-deep loopnest. The outer loop enumerates the global indices of the starting points of the rows (order=row_wise) or the columns (order=column_wise), depending on the enumeration order as specified by order.

In most compilers the order of enumeration is fixed. However, we have mofified the method outlined in [1] such that the generated code for both enumeration orders is identical, by adding a parameter 'order' to the run-time routines that

S.P. Midkiff et al. (Eds.): LCPC 2000, LNCS 2017, pp. 355–358, 2001.

determine the local loop nests. The only penalty is that a small test must be performed inside the run-time function, that calculates the length of the inner loop for both enumeration orders. The test comprises the evaluation and comparison of the following expressions: order = row_wise if $R_{ext}/\Delta r < C_{ext}/\Delta c$ and order = column_wise if $R_{ext}/\Delta r \geq C_{ext}/\Delta c$, where R_{ext} and C_{ext} are the number of rows and columns, respectively.

Measurements

We have implemented the modified method into the prototype HPF compiler developed in the Esprit PREPARE project [2]. The compiler can make an automatic choice between enumeration orders, but for comparison, we can also select one of the enumeration orders manually. We have compared the performance of our prototype compiler with the *pghpf* compiler from PGI and the *Adaptor* compiler from GMD. The parallel computer used is a Myrinet-based Pentium Pro cluster. All HPF compilers use the same MPI library.

The local enumeration performance is tested using a simple distributed array initialization, i.e. $A(i) = 1$. The program template is shown below. To generate a test program from the template, the template name and the appropriate value of the D coordinate is needed. The test program can be derived by removing the !*! comment prefix form each line which contains the appropriate value of D. Lines containing a !*! comment prefix but none of the space coordinates can simply be removed.

We are mainly interested in the performance when 1) different alignments, 2) different kinds of templates, and 3) different enumerations (row-wise versus column-wise) are used. We distinguish four kinds of templates: $D1$: *block* distributed template, when there is only one row in the template; $D2$: *block-like* distributed template, when there is in the template only a small amount of rows compared to the number of columns; $D3$: *cyclic-like* distributed template, when there is in the template only a small amount of columns compared to the number of rows in the local data layout; $D4$: *cyclic* distributed template, when there is only one column in the template.

```
PROGRAM init1D
INTEGER, parameter :: N=100000,M=N/(4*NUMBER_OF_PROCESSORS()),imaxa=1000
INTEGER            :: i,iter,l,u,s
REAL               :: A(N),tim1,tim2,time_us
!HPF$ PROCESSORS P(NUMBER_OF_PROCESSORS())
!D 1, 2, 3, 4!!HPF$ TEMPLATE T(N)
!D 5, 6, 7, 8!!HPF$ TEMPLATE T(N+2)
!D 9,10,11,12!!HPF$ TEMPLATE T(3*N+7)
!D 1, 5, 9   !!HPF$ DISTRIBUTE T(BLOCK    ) ONTO P
!D 2, 6, 10  !!HPF$ DISTRIBUTE T(CYCLIC(M)) ONTO P ! block-like
!D 3, 7, 11  !!HPF$ DISTRIBUTE T(CYCLIC(4)) ONTO P ! cyclic-like
!D 4, 8, 12  !!HPF$ DISTRIBUTE T(CYCLIC   ) ONTO P
!D 1, 2, 3, 4!!HPF$ ALIGN (i) WITH T(  i  ) :: A
!D 5, 6, 7, 8!!HPF$ ALIGN (i) WITH T(  i+2) :: A
```

```
!D 9,10,11,12!!HPF$ ALIGN (i) WITH T(3*i+7) :: A
        tim1 = time_us()
        DO iter = 1, imaxa
          CALL values(N,iter,l,u,s)
          FORALL (i=l:u:s)
            A(i) = 1.0
          END FORALL
        END DO
        tim2 = time_us()
        print tim2-tim1
      END PROGRAM
```

The measured results in seconds are given in the table below. The column labeled with *np* shows the number of processors. The column labeled $Tkind$ gives the kind of template the array was aligned with. The alignment itself is given in the second row. For each of these alignments, three columns are given for the PRE-HPF compiler, labeled $(bc\_col, bc\_row, bl/cy)$. The column labeled *bc_col* gives the timings of a column-wise enumeration with a double loop. The column labeled *bc_row* gives the timings of a row-wise enumeration with a double loop. The third column labeled with bl/cy gives the timings of a block enumeration when the template is block distributed and the timings of a cyclic enumeration when the template is cyclic distributed, respectively. The columns labeled with *pgi* and *gmd* are the timings of the PGI-HPF and GMD-HPF generated codes, respectively. The blank table entries denote that the program did not compile or crashed at run-time. The gray table entries denote the best timings.

We will first concentrate on the timings of the PRE-HPF compiler. When looking at the results, many columns are the same. For instance, all timings of the direct alignment *'Align A(i) with T(i)'* are almost the same as the timings of the shifted alignment *'Align A(i) with T(i+2)'*, as expected. The strided alignment *'Align A(i) with T(3i+7)'* however gives totally different timings for the white columns of the block-like, cyclic-like, and cyclic distributions. This is a result of the fact that the stride is three and in worst case, the compiler might need to check three times as much rows or columns when enumerating *bc_col* or *bc_row*. Whether the compiler knows if a distribution is *block* or *cyclic* does not matter for the performance, as long as the compiler enumerates *bc_row* for a block distribution and *bc_col* for a cyclic distribution. In fact, this is no surprise because the generated outer loop will only loop once and hence does not generate extra loop overhead. On the other hand, when the inner loop would only loop once, the worst-case loop overhead will be measured, as shown in the columns marked with the ellipses. In the PRE-HPF compiler, row wise storage compression is used followed by a column wise compression (tile-wise compression).

The generated code for $cyclic(m)$ enumeration allows a selection of *bc_col* or *bc_row* enumeration at run-time. The run-time system then always selects the largest loop as the inner loop. This yields the best performance, independent of the alignment and the distribution, except for the distribution block-like. For the block-like distribution, the best timings are shared between the PGI-HPF compiler and the PRE-HPF compiler. For a few processors, the PGI-HPF com-

init1D		Align A(i) with T(i)					Align A(i) with T(i+2)					Align A(i) with T(3i+7)				
		pre			pgi	gmd	pre			pgi	gmd	pre			pgi	gmd
Tkind	np	bc_col	bc_row	bl/cy			bc_col	bc_row	bl/cy			bc_col	bc_row	bl/cy		
block	1	184.29	5.75	5.75	5.97	5.93	184.31	5.86	5.87	6.21	5.93	184.32	5.86	5.86	6.04	6.23
	2	91.90	2.26	2.28	2.70	2.17	91.86	2.10	2.12	2.68	1.75	91.90	2.25	2.28	2.63	2.06
	3	61.18	1.26	1.30	1.54	1.01	61.13	1.14	1.15	1.53	1.15	61.18	1.27	1.30	1.52	1.02
	4	45.81	0.76	0.76	1.21	0.76	45.80	0.76	0.76	1.15	0.76	45.81	0.76	0.76	1.14	0.76
	5	36.64	0.61	0.61	0.93	0.61	36.65	0.61	0.60	0.93	0.61	36.65	0.61	0.61	0.92	0.61
	6	30.54	0.51	0.50	0.76	0.51	30.54	0.51	0.51	0.78	0.51	30.53	0.51	0.51	0.77	0.51
block-like	1	56.24	22.66		6.18		56.24	22.66		6.04		177.70	37.31		6.23	
	2	27.99	6.40		2.91		28.00	6.96		2.95		89.36	15.47		2.86	
	3	18.61	2.38		1.90		18.64	3.20		2.01		18.61	2.38		1.97	
	4	13.94	1.14		1.43		13.94	1.15		1.42		44.46	3.37		1.44	
	5	11.15	0.92		1.15		11.15	0.92		1.14		35.50	1.37		1.15	
	6	9.29	0.77		0.95		9.29	0.77		0.95		9.29	0.77		0.96	
cyclic-like	1	6.04	57.57		30.99	191.11	6.27	56.41		31.19		6.06	177.91		117.38	
	2	2.43	28.40		15.23	285.55	2.48	28.04		15.19		2.43	88.72		58.81	
	3	1.28	18.75		10.13	250.50	1.14	18.58		10.08		0.76	58.71		38.82	
	4	0.77	13.90		7.56	231.08	0.77	13.91		7.56		0.76	44.18		28.95	
	5	0.61	11.19		6.05	220.07	0.61	11.13		6.05		0.61	35.35		23.19	
	6	0.52	9.31		5.04	212.27	0.51	9.27		5.05		0.39	29.35		19.30	
cyclic	1	6.27	183.09	6.27	8.68	20.14	6.05	183.04	6.05	108.13		5.95		5.95	154.20	
	2	2.47	91.35	2.49	4.33	120.30	2.43	91.33	2.45	55.36		2.41	348.18	2.43	78.26	
	3	1.27	60.78	1.30	2.89	79.99	1.15	60.75	1.15	36.79		1.15	230.34	1.15	154.21	
	4	0.76	45.52	0.80	2.17	59.89	0.76	45.52	0.80	26.70		0.77	173.89	0.80	39.82	
	5	0.61	36.41	0.65	1.73	47.97	0.61	36.41	0.65	21.35		0.61	139.11	0.64	32.31	
	6	0.51	30.35	0.53	1.44	39.86	0.51	30.35	0.54	17.80		0.54	115.17	0.54	76.88	

piler performs best, because the local elements are stored row-wise and therefore the efficient stride-one memory access occurs. The PRE-HPF compiler timings correspond to a less efficient strided memory access, because it uses a fixed column-wise storage of local elements. If we would have selected a row-wise storage for the local elements, the timings of the (*bc_row*, block-like) columns can be swapped with the timings of the (*bc_col*, cyclic-like) columns. The PRE-HPF compiler then outperforms the PGI-HPF compiler for all three alignments. Looking at the PGI-HPF timings, we conclude that it uses a fixed row-wise enumeration scheme and the local elements are stored row-wise. This results in the same timings for a block kind of template as for the PRE-HPF compiler. Better timings are observed for the block-like templates, due to the row-wise local storage. If the PRE-HPF compiler also selects a row-wise storage, the above mentioned columns can be swapped and it then outperforms the PGI-HPF compiler by a factor between 1.5 and 4. Slow execution times occur for templates where the PGI-HPF compiler should have switched over to column-wise enumeration, like in the cyclic-like templates. The PGI-HPF timings of a cyclic template are strongly dependent on the alignment used. It varies from twice as large for the direct alignment up to 40 times as large for the strided alignment.

References

1. H.J. Sips, W.J.A. Denissen, C. van Reeuwijk, 'Analysis of local enumeration and storage schemes in HPF,' *Parallel Computing*, 24, pp. 355-382, 1998.
2. F. Andre, P. Brezany, O. Cheron, W. Denissen, J.L. Pazat, K. Sanjari, 'A new compiler technology for handling HPF data parallel constructs,' *Proc. 3-rd Workshop Languages, Compilers, Run-time Systems for Scalable Computing*, Kluwer, 1995.

Issues of the Automatic Generation of HPF Loop Programs

Peter Faber, Martin Griebl, and Christian Lengauer

Fakultät für Mathematik und Informatik
Universität Passau, D–94030 Passau, Germany
{faber,griebl,lengauer}@fmi.uni-passau.de

Abstract. Writing correct and efficient programs for parallel computers remains a challenging task, even after some decades of research in this area. One way to generate parallel programs is to write sequential programs and let the compiler handle the details of extracting parallelism. *LooPo* is an automatic parallelizer that extracts parallelism from sequential loop nests by transformations in the polyhedron model. The generation of code from these transformed programs is an important step. We report on problems met during code generation for HPF, and existing methods that can be used to reduce some of these problems.

1 Introduction

Writing correct and efficient programs for parallel computers is still a challenging task, even after several decades of research in this area. Basically, there are two major approaches: one is to develop parallel programming paradigms and languages which try to simplify the development of parallel programs (e.g., data-parallel programming [PD96] and HPF [Hig97]), the other is to hide all parallelism from the programmer and let an automatically parallelizing compiler do the job.

Parallel programming paradigms have the advantage that they tend to come with a straightforward compilation strategy. Optimizations are mostly performed based on a textual analysis of the code. This approach can yield good results for appropriately written programs. Modern HPF compilers are also able to detect parallelism automatically based on their code analysis.

Automatic parallelization, on the other hand, often uses an abstract mathematical model to represent operations and dependences between them. Transformations are then done in that model. A crucial step is the generation of actual code from the abstract description.

Because of its generality, we use the *polyhedron model* [Fea96, Len93] for parallelization. Parallel execution is then defined by an affine *space-time mapping* that assigns (virtual) processor and time coordinates to each iteration. Our goal is then to feed the resulting loop nest with explicit parallel directives to an HPF compiler. The problem here is that transformations in the polyhedron model can, in general, lead to code that cannot be handled efficiently by the HPF compiler. In the following section, we point to some key problems that occur during this phase.

S.P. Midkiff et al. (Eds.): LCPC 2000, LNCS 2017, pp. 359–362, 2001.

2 Problems and Solutions

The first step in the generation of loop programs from a set of affine conditions is the scanning of the index space; several methods have been proposed for this task [KPR94, GLW98, QR00]. However, the resulting program, may contain array accesses that cannot be handled efficiently by an `HPF` compiler.

This can be partly avoided by converting to *single assignment* (SA) form. This transformation [Fea91, Coh99] is often used to increase the amount of concurrency that can be exploited by the compiler (since, in this form, only true dependences have to be preserved). Converting to SA form *after* loop skewing – which is proposed by Collard in [Col94] – yields relatively simple index functions: index functions on the left-hand side of an assignment are given by the surrounding loop indices, and index functions on the right-hand side (RHS) are simplified because uniform dependences lead to simple numerical offsets and, thus, to simple shifts that can be detected and handled well by `HPF` compilers.

However, there are three points that cause new problems:

1. SA form in its simple form is extremely memory-consuming.
2. Conversion to SA form may lead to the introduction of so-called *ϕ-functions* that are used to reconstruct the flow of data.
3. Array occurrences on the RHS of a statement may still be too complex for the `HPF` compiler in the case of non-uniform dependences, which may again lead to serialized load communications.

The first point is addressed by Lefebvre and Feautrier [LF98]. Basically, they introduce modulo operators in array subscripts that cut down the size of the array introduced by SA conversion to the length of the longest dependence for a given write access. The resulting arrays are then partially renamed, using a graph coloring algorithm with an interference relation (write accesses may conflict for the same read) as edge relation. Modulo operators are very hard to analyze, but introducing them for array dimensions that correspond to loops that enumerate time steps (in which the array is not distributed) may still work, while spatial array dimensions should remain without modulo operators. In the distributed memory setting, this optimization should generally not be applied directly, since this would result in some processors owning the data read and written by others. The overall memory consumption may be smaller than that of the original array but, on the other hand, buffers and communication statements for non-local data have to be introduced. One solution is to produce a tiled program and not use the modulo operator in distributed dimensions.

ϕ-functions may be necessary in SA form due to several possible sources of a single read since, in SA form, each statement writes to a separate, newly introduced array. ϕ-functions select a specific source for a certain access; thus, their function is similar to the ?-operator of `C`. In the case of selections based on affine conditions, ϕ-functions can be implemented by copy operations executed for the corresponding part of the index, which can be scanned by standard methods. Yet, even this implementation introduces memory copies that can be

avoided by generating code for the computation statement for some combinations of possible sources directly, trading code size for efficiency.

For non-affine conditions, additional data structures become necessary to manage the information of which loop iteration performs the last write to a certain original array cell. Cohen [Coh99] offers a method for handling these accesses and also addresses optimization issues for shared memory; in the distributed memory setting, however, generation of efficient management code becomes more complicated, since the information has to be propagated to all processors.

A general approach for communication generation that can also be used to vectorize messages for affine index expressions is described by Coelho [ACKI95]: send and receive sets are computed, based on the meet of the data owned by a given sending processor with the read accesses of the other processors – all given by affine constraints. The corresponding array index set is then scanned to pack the data into a buffer; an approach that has also been taken by the dHPF compiler [AMC98]. In a first attempt to evaluate this generalized message vectorization using portable techniques, we implemented `HPF_LOCAL` routines for packing and sending data needed on a remote processor and copying local data to the corresponding array produced by single-assignment conversion. However, our first performance results with this compiler-independent approach were not encouraging due to very high overhead in loop control and memory copies.

Communication generation may also be simplified by communicating a superset of the data needed. We are currently examining this option. Another point of improvement that we are currently considering is to recompute data locally instead of creating communication, if the cost of recomputing (and the communication for this recomputation) is smaller than the cost for the straightforward communication. Of course, this scheme cannot be used to implement purely pipelined computation, but may be useful in a context where overlapping of computation with communication (see below) and/or communication of a superset of data can be used to improve overall performance.

Overlapping of communication with computation is also an important optimization technique. Here, it may be useful to fill the temporaries that implement the sources of a read access directly after computing the corresponding value. Data transfers needed for these statements may then be done using non-blocking communication, and the operations, for which the computations at a given time step must wait, are given directly by an affine function. Although our preliminary tests did not yield positive results, we are still pursuing this technique.

A further issue is the size and performance of the code generated by a polyhedron scan. Quilleré and Rajopadhye [QR00] introduce a scanning method that separates the polyhedra to be scanned such that unnecessary `IF` statements inside a loop – which cause much run-time overhead – are completely removed. Although this method still yields very large code in the worst case, it allows to trade between performance and code size by adjusting the dimension in which the code separation should start, similar to the Omega code generator [KPR94]. So, the question is: which separation depth should be used for which statements? A practical heuristics may be to separate the loops surrounding computation state-

ments on the first level, scan the loops implementing ϕ-functions separately, and replicate these loop nests at the beginning of time loops.

3 Conclusions

We have learned that straightforward output of general skewed loop nests leads to very inefficient code. This code can be optimized by converting to SA form and leveraging elaborate scanning methods. Yet, both of these methods also have drawbacks that need to be weighed against their benefits. There is still room left for optimization by tuning the variable factors of these techniques. Code size and overhead due to complicated control structures have to be considered carefully.

Acknowledgements This work is being supported by the DFG through project *LooPo/HPF*.

References

[ACKI95] C. Ancourt, F. Coelho, R. Keryell, and F. Irigoin. A linear algebra framework for static HPF code distribution. Technical Report A-278, ENSMP-CRI, November 1995.

[AMC98] V. Adve and J. Mellor-Crummey. Using integer sets for data-parallel program analysis and optimization. *ACM SIGPLAN Notices*, 33(5), May 1998.

[Coh99] A. Cohen. *Program Analysis and Transformation: From the Polytope Model to Formal Languages*. PhD thesis, Laboratoire PRiSM, Université de Versailles, December 1999.

[Col94] J.-F. Collard. Code generation in automatic parallelizers. In C. Girault, editor, *Proc. of the Int. Conf. on Applications in Parallel and Distributed Computing, IFIP W.G 10.3*, Caracas, Venezuela, April 1994. North Holland.

[Fea91] P. Feautrier. Dataflow analysis of array and scalar references. *Int. J. Parallel Programming*, 20(1):23–53, February 1991.

[Fea96] P. Feautrier. Automatic parallelization in the polytope model. In G.-R. Perrin and A. Darte, editors, *The Data Parallel Programming Model*, LNCS 1132, pages 79–103. Springer-Verlag, 1996.

[GLW98] M. Griebl, C. Lengauer, and S. Wetzel. Code generation in the polytope model. In *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT'98)*, pages 106–111. IEEE Computer Society Press, 1998.

[Hig97] High Performance Fortran Forum. *HPF Language Specification*, 1997.

[KPR94] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. Technical Report 3317, Dept. of Computer Science, Univ. of Maryland, 1994.

[Len93] C. Lengauer. Loop parallelization in the polytope model. In E. Best, editor, *CONCUR'93*, LNCS 715, pages 398–416. Springer-Verlag, 1993.

[LF98] V. Lefebvre and P. Feautrier. Automatic storage management for parallel programs. *Parallel Computing*, 24(2):649–671, 1998.

[PD96] G.-R. Perrin and A. Darte, editors. *The Data Parallel Programming Model*, LNCS 1132. Springer-Verlag, 1996.

[QR00] F. Quilleré and S. Rajopadhye. Code generation for automatic paralelization in the polyhedral model. *Int. J. Parallel Programming*, 28(5), 2000. to appear.

Run-Time Fusion of MPI Calls in a Parallel C++ Library

Antony J. Field, Thomas L. Hansen*, and Paul H.J. Kelly

Department of Computing, Imperial College,
180 Queen's Gate, London SW7 2BZ, U.K.
A.J.Field@ic.ac.uk

Abstract. CFL (Communication Fusion Library) is a C++ library for MPI programmers. It uses overloading to distinguish private variables from replicated, shared variables, and automatically introduces MPI communication to keep such replicated data consistent. This paper concerns a simple but surprisingly effective technique which improves performance substantially: CFL operators are executed lazily in order to expose opportunities for run-time, context-dependent, optimisation such as message aggregation and operator fusion. We evaluate the idea in the context of a large-scale simulation of oceanic plankton ecology. The results demonstrate the software engineering benefits that accrue from the CFL abstraction and show that performance close to that of manually optimised code can be achieved automatically in many cases.

1 Introduction

In this paper we describe an experimental abstract data type for representing shared variables in SPMD-style MPI programs. The operators of the abstract data type have a simple and intuitive semantics and hide any required communication from the user. Although there are some interesting issues in the design of the library, the main contribution of this paper is to show how lazy evaluation is used to expose run-time optimisations which may be difficult, or even impossible, to spot using conventional compile-time analyses. Figure 1 shows the basic idea. The CFL class library can be freely mixed with standard MPI operations in a SPMD application. C++ operator overloading is used to simplify the API by using existing operator symbols (e.g. +, *, += etc.) although these may have a parallel reading when the target of an assignment is another shared variable.

Figure 2 shows how this code could be optimised manually by fusing the three reductions. This paper shows how our implementation automatically achieves this behaviour.

Related work There has been a lot of interesting work using C++ to simplify parallel programming [1]. The idea of delaying library function evaluation to create

* Thomas Hansen is now with Telcordia Inc, NJ

S.P. Midkiff et al. (Eds.): LCPC 2000, LNCS 2017, pp. 363–366, 2001.

```
double x, y, z, a, b, c, d, e, f, n;
for(i=0;i<NO_ITER;i++) {
 a = ...new value for a...;
 MPI_Allreduce(&a, &d, 1, MPI_DOUBLE,
               MPI_SUM, MPI_COMM_WORLD);
 x += d;
 b = ...new value for b...;
 MPI_Allreduce(&b, &e, 1, MPI_DOUBLE,
               MPI_SUM, MPI_COMM_WORLD);
 y -= e;
 c = ...new value for c...;
 MPI_Allreduce(&c, &f, 1, MPI_DOUBLE,
               MPI_SUM, MPI_COMM_WORLD);
 z += f;

 n += x + y + z;
}
```

```
CFL_Double x, y, z;
double n;
for(i=0;i<NO_ITER;i++) {
 x += ...new value for a...;
 y -= ...new value for b...;
 z += ...new value for c...;
 n += x + y + z;
}
```

Fig. 1. In the MPI code (left), x, y and z are replicated and updated explicitly using MPI. Using CFL, (above), replicated shared variables are declared as `CFL_Double`. Arithmetic and assignment operators are overloaded to implement any communication needed to keep each processor's data up to date.

optimisation opportunities is fairly well-known (eg. [5], [2]). Compile-time techniques are applicable, but require interprocedural analysis. An alternative is for the programmer to build an execution plan explicitly (eg. [7]). Weak consistency in an application-specific cache coherency protocol creates similar optimisation opportunities.

Implementation Space precludes a full explanation of how CFL works. As computation proceeds, we accumulate deferred updates to each `CFL_Double`, together with a list of deferred communications. Execution is forced when a `CFL_Double` is assigned to a local variable, appears in a conditional, or is passed as a `double` parameter. Some care is needed to handle expression intermediate values properly. Note that the semantics of a statement like `x=y+a` depends crucially on which of `x`, `y` and `a` are `double`s and which are `CFL_Double`s.

2 Experimental Evaluation and Application Experience

Our performance results are from dedicated runs on a Fujitsu AP3000, a distributed-memory MPP comprising 32 300MHz Sparc Ultra II processors each with 128MB RAM, linked by a fast proprietary interconnect which is accessed directly by the vendor's MPI implementation.

We present two experiments to help evaluate the work. The first is the artificial application shown at the beginning of the paper. For the local calculations to compute `a`, `b` and `c` we chose trivial arithmetic operations, so the computation is entirely dominated by communication. The results are presented in Table 1.

```
double x, y, z, sbuf[3], rbuf[3], n;
for(i=0;i<NO_ITER;i++) {
 sbuf[0] = ...new value for a...;
 sbuf[1] = ...new value for b...;
 sbuf[2] = ...new value for c...;
 MPI_Allreduce(sbuf, rbuf, 3, MPI_DOUBLE,
               MPI_SUM, MPI_COMM_WORLD);
 x += rbuf[0];
 y -= rbuf[1];
 z += rbuf[2];
 n += x + y + z;
}
```

Fig. 2. Manually optimised implementation of example from Figure 1. The communications can be delayed, but have to occur in order to produce the correct value for the non-shared variable `n`. At that point, the three scalar `MPI_Allreduce` operations can be combined into a single `MPI_Allreduce` operating on a three-element vector.

Table 1. With the contrived communication-bound example of Figures 1 ("Unoptimised" and "Automatic") and 2 ("Manual"), optimisation reduces three scalar `MPI_Allreduce` calls to one operating on a three-element vector. Due to run-time overheads, CFL does not quite realise a 3-fold speedup. With one processor, the CFP overheads dominate (the redundant MPI calls were still executed).

Procs	Execution time(s)		
	Unoptimised	Manual	Automatic
1	1.11	1.12	5.02
2	49.9	15.0	17.9
4	48.8	14.2	17.0
8	37.9	11.3	12.5
16	32.6	9.4	10.5
32	20.0	6.0	6.4

A full-scale application: Ocean Plankton Ecology The second experiment is a full-scale application which models plankton ecology in the upper ocean using the Lagrangian Ensemble method [6] (the "ZB" configuration).

Because of the modular nature of the application, environment variables such as concentrations of nutrients are assigned in one module, and frequently not used until a later module. The relatively large distance between the producer and consumer provides good scope for message aggregation.

The results are shown in Figure 2. With a larger 3.2M-particle problem with 32 processors the CFP performance gain is around 12%, improving efficiency from 87% to 97%.

3 Conclusions

This paper presents a simple idea, which works remarkably well. We have built a small experimental library on top of MPI which enables shared scalar variables in parallel SPMD-style programs to be represented as an abstract data type. Using operator overloading, the familiar arithmetic operators can be used, although the operators may have a parallel reading when the target of an assignment is

Table 2. Ocean Plankton Ecology application (320,000 particles). Synchronisations are reduced from 27 to 3 per time step with both manual and automatic optimisation. CFL overheads are too small to measure (the small superlinear speedup is due to cache effects, which probably also account for the good CFP performance on 32 processors).

Procs	Unoptimised		Manual		Automatic		CFL
	secs	speedup	secs	speedup	secs	speedup	benefit
1	3721	1.00	3721	1.00	3738	0.995	
2	1805	2.06	1779	2.09	1790	2.08	1%
4	934	3.98	869	4.28	866	4.30	8%
8	491	7.58	433	8.59	418	8.90	17%
16	317	11.74	244	15.25	257	14.48	23%
32	292	12.74	191	19.48	182	20.45	60.5%

another shared variable. We have shown how delayed evaluation can be used to aggregate the reduction messages needed to keep such variables consistent, with potentially substantial performance benefits. This is demonstrated across the modules of a large oceanography application. The approach avoids reliance on sophisticated compile-time analyses and exploits opportunities which arise from dynamic data dependencies. Using a contrived test program and a realistic case study we have demonstrated very pleasing performance improvements.

Extending the library to support reductions of shared arrays should be straightforward. Extending the idea to other communication patterns presents interesting challenges which we are investigating [4].

References

1. Susan Atlas, Subhankar Banerjee, Julian C. Cummings, Paul J. Hinker, M. Srikant, John V. W. Reynders, and Marydell Tholburn. *POOMA: A high performance distributed simulation environment for scientific applications*. In Supercomputing '95, 1995.
2. O. Beckmann, P. H. J. Kelly: *Efficient Interprocedural Data Placement Optimisation in a Parallel Library*, LCR '98
3. O. Beckmann, P. H. J. Kelly: *Runtime Interprocedural Data Placement Optimisation for Lazy Parallel Libraries (extended abstract)*, EuroPar '97
4. O.B. Beckmann and P.H.J. Kelly, *A Linear Algebra Formulation for Optimising Replication in Data Parallel Programs*. In LCPC'99, Springer Verlag (2000).
5. R. Parsons and D. Quinlan. *Run time recognition of task parallelism within the P++ parallel array class library*. In Proceedings of the Scalable Parallel Libraries Conference, pages 77–86. IEEE Comput. Soc. Press, October 1993.
6. J. Woods and W. Barkmann, *Simulation Plankton Ecosystems using the Lagrangian Ensemble Method*, Philosophical Transactions of the Royal Society, B343, pp. 27-31.
7. S.J. Fink, S.B. Baden and S.R. Kohn, *Efficient Run-time Support for Irregular Black-Structured Applications*, Journal of Parallel and Distributed Programming, Vol 50, No. 1, pp 61–82, 1998.

Set Operations for Orthogonal Processor Groups

Thomas Rauber[1], Robert Reilein[2], and Gudula Rünger[2]

[1] Universität Halle-Wittenberg, Germany,
rauber@informatik.uni-halle.de
[2] Technische Universität Chemnitz, Germany,
{reilein,ruenger}@informatik.tu-chemnitz.de

Abstract. We consider a generalization of the SPMD programming model for distributed memory machines based on orthogonal processor groups. In this model different partitions of the processors into disjoint processor groups exist and can be used simultaneously in a single parallel implementation. Set operations on orthogonal groups are used to express group-SPMD computations on different partitions of the processors. The set operations are implemented in MPI.

1 Introduction

Many applications from scientific computing exhibit a two-dimensional computational structure. Examples are matrix computations or grid-based computations. A typical approach for parallel execution on a distributed memory machine (DMM) is to distribute the data and let each processor perform the computations for its local data. Because of the importance of obtaining efficient programs for grid-based computations, there are many approaches to support the development of efficient programs for novel architectures, see [3] for an overview. Reducing the communication overhead is one of the main concerns when developing programs for DMMs. Grid-based computations with strong locality, such that a computation at a grid point needs only data from neighboring grid points, are best implemented with a blockwise data distribution resulting in point-to-point communication between neighboring processors. But many grid based algorithms exhibit a more diverse dependence pattern and the communication needed to realize the dependencies is affected strongly by the data distribution. To minimize the communication overhead, highly connected computations should reside on the same processor.

We consider applications with a two-dimensional structure that exhibits dependencies in both the vertical and horizontal directions, in the sense that parts of the computation are column-oriented with similar computations and dependencies within the same column, while other parts of the computation are row-oriented. To realize an efficient implementation for pure horizontal or pure vertical computations, corresponding partitions of processors into disjoint groups are needed. The advantage of using group operations on disjoint processor groups is that collective communication operations performed on smaller processor groups lead to smaller execution times due to the logarithmic or linear dependence of

S.P. Midkiff et al. (Eds.): LCPC 2000, LNCS 2017, pp. 367–371, 2001.

the communication times on the number of processors. For computations on changing or alternating directions, different partitions of the processors have to be used for different directions. Correspondingly, it is useful to arrange the processors in orthogonal partition structures so that each processor belongs to a different group for each of the partitions. Since different directions of computation may be active at different points during the execution, it is suitable to use a data distribution that is not biased towards a specific direction. Examples are blockwise data distributions or data distributions that are cyclic in each direction.

In this paper, we extend a double-cyclic data distribution to a processor grid with orthogonal processor groups, which means that for one set of processors executing the program there simultaneously exist two different partitions into disjoint groups of processors. This leads to programs in a group-SPMD computation model with different groups in different parts of the program. The concurrently executed SPMD-code includes collective communication within each group of a partition so that data transfers can be combined into one communication operation. The processor group concept reduces the number of processors that have to participate in a collective communication operation, resulting in faster execution. Our approach is not simply to construct new groups dynamically. Instead, we define a fixed set of useful orthogonal groups at the outset. As the execution proceeds, the program establishes one grouping, performs SPMD operations and collective communications, and then establishes the other grouping for the next step. Disjoint processor groups can be expressed in the communication library MPI. In addition, we use set operations for the concurrently executing processor partitions, which are also implemented in MPI. As an example, we consider an LU decomposition for which a double-cyclic distribution leads to a good load balance [4, 5]. This approach can also be used when combining task and data parallel computations [1].

2 Describing Orthogonal Computation Structures

We consider application programs with an orthogonal internal computation structure. For the organization of the computations and the assignment to processors, we use the following abstraction: A parallel application program is composed of a set $\mathcal{T}$ of n one-processor tasks that are organized in a two-dimensional structure. The tasks are numbered with two-dimensional indices in the form T_{ij}, $i = 1, \ldots, n_1, j = 1, \ldots, n_2$, with $n = n_1 * n_2$. Each single task $T \in \mathcal{T}$ consists of a sequence of computations which may need data from other tasks or which may produce data needed by other tasks.

The entire task program is expressed in an SPMD style with explicit constructs for horizontal or vertical executions, which we call *horizontal sections* and *vertical sections* respectively. In horizontal sections a task T_{ij} has interactions with a set of tasks $\{T_{ij'} | j' = 1, ..., n_2, j' \neq j\}$. In vertical sections a task T_{ij} has interactions with a set of tasks $\{T_{i'j} | i' = 1, ..., n_1, i' \neq i\}$. A single task T_{ij} is composed of communication and computation commands. To indicate that a

task participates in an SPMD-like operation together with other tasks in the horizontal or vertical direction, respectively, we introduce specific commands with the following meaning:

- vertical_section(k) { statements } :
  Each task in column k executes statements in an SPMD-like way together with the other tasks in column k; statements may contain computations as well as collective communication and reduction operations involving tasks $T_{1k}, \ldots, T_{n_1,k}$. Tasks T_{ij} with $j \neq k$ perform a `skip`-operation.
- horizontal_section(k) { statements } :
  Similar to vertical_section(k), but uses a horizontal organization.
- vertical_section( ) { statements } :
  Each task executes statements in an SPMD-like way together with the other tasks in the same column. A task in column k may perform computations as well as collective communication and reduction operations involving the other tasks $T_{1k}, \ldots, T_{n_1,k}$ in the same column. Computations in a specific column of the task array are executed in parallel with the other columns. Thus, vertical_section( ) corresponds to a parallel loop over all columns k with each iteration executing vertical_section(k).
- horizontal_section( ) { statements } :
  Similar to vertical_section( ), but uses a horizontal organization.

3 Mapping to Orthogonal Processor Groups

In order to exploit the potential parallelism of orthogonal computation structures, we map the computations onto a two-dimensional processor grid for which we provide two different partitions that exist simultaneously and can be exploited to reduce the communication overhead. For the assignment of tasks to processors, we use parameterized mappings similar to parameterized data distributions [2, 4] which describe the data distribution for arrays of arbitrary dimension. This mechanism is used to define disjoint row and column groups such that each row and column of the task array is assigned to a single row and column group respectively. For the parameterized mapping, the processors are logically arranged in a two-dimensional processor grid that can be described by the number p_1 and p_2 of processors in each dimension. A double-cyclic mapping of the two-dimensional task array to the processor grid is specified by blocksizes b_1 and b_2 in each dimension, which determine the number of consecutive rows and columns that each processor obtains of each cyclic block. The row groups and the column groups are orthogonal processor groups.

The mapping of the tasks to the processors defines the computations that each processor has to perform. Horizontal and vertical sections require the coordination of the participating processors. A vertical_section(k) operation is performed by all processors in the corresponding column group. Similarly, a horizontal_section(k) operation is performed by all processors in the corresponding row group. A vertical_section() operation is performed by all processors, but each

processor exchanges information only with the processors in the same column group. A horizontal_section() is executed analogously.

The realization is performed by using the communicator or group concepts of MPI. Since the parallel computation on the orthogonal groups is often embedded in a larger application, it is convenient to use the global ranks of the processors to direct the computations. Set operations are used to identify members of a processor group responsible for a specific row or column of the task array or to pick processors in the intersection of specific row or column groups.

4 Example

Figure 1 shows runtime results for a double-cyclic LU decomposition with column pivoting on a Cray T3E-1200. The blocksize in each dimension has been set to 1. The diagrams compare two versions which are based on a data distribution on the global group of processors with an implementation resulting from the use of orthogonal groups. These groups are used for computing the pivot element on a single column group, for making the pivot row available to all processors by parallel executions on all column groups, for computing the elimination factors on a single column group, and for broadcasting the elimination factors in the row groups. The processor grid uses an equal number of processors in each dimension, i.e., the row and column groups have equal size. For 24 and 72 processors, processor grids of size 4×6 and 8×9 respectively, are used. All versions result in a good load balance. The diagram shows that for a larger number of processors, the implementation with orthogonal processor groups shows the best runtime results. For the largest number of processors, the percentage improvement over the second best version is 25% and 15% respectively.

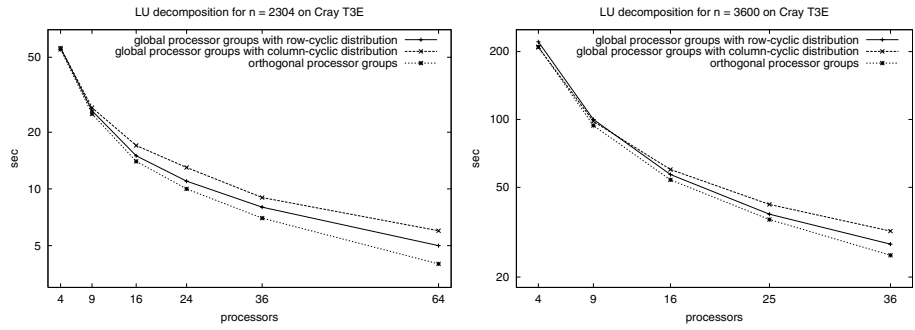


Fig. 1. *Runtimes for the LU decomposition on a Cray T3E-1200.*

Acknowledgement

We thank the NIC Jülich for providing access to the Cray T3E.

References

[1] H. Bal and M. Haines. Approaches for Integrating Task and Data Parallelism. *IEEE Concurrency*, 6(3):74–84, July-August 1998.

[2] A. Dierstein, R. Hayer, and T. Rauber. The ADDAP System on the iPSC/860: Automatic Data Distribution and Parallelization. *Journal of Parallel and Distributed Computing*, 32(1):1–10, 1996.

[3] S.J. Fink. *A Programming Model for Block-Structured Scientific Calculations on SMP Clusters*. PhD thesis, University of California at San Diego, La Jolla, CA, 1998.

[4] T. Rauber and G. Rünger. Deriving Array Distributions by Optimization Techniques. *Journal of Supercomputing*, 15:271–293, 2000.

[5] E. van de Velde. *Concurrent Scientific Computing*. Springer, 1994.

Compiler Based Scheduling of Java Mobile Agents*

Srivatsan Narasimhan and Santosh Pande

PO Box 210030, University of Cincinnati
Cincinnati, OH - 45221
{snarasim,santosh}@ececs.uc.edu

Abstract. This work presents compiler-based scheduling strategies for Java Mobile Agents. We analyze the program using annotations and data sizes. For the different strategies, the compiler produces the best schedule, taking dependence information into account.

1 Introduction

Java Mobile Agents are useful in distributed data mining applications. This work attempts to perform compiler based optimizations of Java mobile agents. Our compiler (called **ajava**) takes an annotated Java program as input and performs various optimizations. It identifies a schedule that carries minimum amount of data through the network and generates mobile agents.

2 Motivating Example

Distributed Data Mining applications can take advantage of Mobile agents. The following example accesses data distributed across three clients in the local network. The database tables used are **Loans** (5 records each of 20 bytes), **Employee**(5000 records each of 40 bytes) and **LoansAvailed** (7000 records of 12 bytes each). In the example given in figure 1, under the constraints of flow dependence, statements S1, S2, S3 can be interchanged but S4 can occur only after S1, S2 and S3. Therefore the possible legal schedules are

(S1,S2,S3,S4),(S1,S3,S2,S4),(S2,S1,S3,S4)
(S2,S3,S1,S4),(S3,S1,S2,S4),(S3,S2,S1,S4)

Among the six schedules only (S2,S3,S1,S4) carries the least amount of data. The amount of data moved around by this schedule is given below.

$$Total\ Size\ of\ Employee\ table\ =\ 5000 * 40\ = 200000\ bytes$$
$$Total\ Size\ of\ Loans\ table\ =\ 5 * 20\ = 100\ bytes$$
$$Total\ Size\ of\ Loans\ Availed\ table\ = 7000 * 12\ = 84000\ bytes$$
$$Total\ data\ carried\ =\ 100 + 84100 + 284100\ = 368300\ bytes$$

* *This work was partially supported by NSF grant #EIA 987135*

S.P. Midkiff et al. (Eds.): LCPC 2000, LNCS 2017, pp. 372–376, 2001.

```
        //& (Employee, atp://stationB.ececs.uc.edu, 5000);
        //& (Loans, atp://stationA.ececs.uc.edu, 5);
        //& (LA, atp://stationC.ececs.uc.edu, 7000);
S1      Employee = getEmployeeRecords();
S2      Loans = getLoansRecords();
S3      LA = getLoansAvailedRecords();
S4      resultSet = getResultSet(Employee,LA,"salary > 10000
        and loan_no = 4");
```

Fig. 1. Scheduling of Mobile Agents

3 Framework for Efficient Schedule Generation

Our compiler framework uses annotations in the Java programs, which identify the distributed data variables and the host from which the data for them can be extracted. It also specifies the approximate size of the data that the variable would be carrying. The syntax of the annotation is

//& (Variable Name, Host Name, Approximate Size);

Carrying Data Strategy (CDS): In this strategy, the mobile agent carries the data around and after getting all the necessary data, it will operate on them using the code at the source host. This strategy is applied if the following is true.

$$|f(arg_1, arg_2, ...arg_n)| > |arg_1| + |arg_2| + ... + |arg_n|,$$

where $|x|$ indicates the data size.

Let us consider an example that multiplies two polynomials that are distributed (Figure 2). The size of the input polynomials as seen from the pseudocode are 100 and 100. This implies that the resultant product will have a maximum size of 10000 coefficients. Comparing the size of the input (100+100 = 200), the size of the result is enormous. In this case, the best approach, is to carry the data to the source and then compute the product at the source so that the amount of data carried will be only 200.

Partial Evaluation Strategy (PES): In this strategy, the mobile agent uses the code to operate on the data that is available at a remote host and then carries only the result from there. This strategy is better when the data operated on is enormous or if it cannot be carried around due to security or proprietary reasons. In Figure 3, the query that was evaluated was *The list of employees with salary* $>$ 15000 *and who have availed loan 5.* The result set of this query is small after the *join* but the result set of *The list of employees with salary* $>$ 15000 *alone is large.* Therfore, *Carrying Data Strategy* will carry unnecessary data through the network before it finds that the result set is small. Since, S3 depends on S2 for getting the value of the loan number, it cannot be executed before S2. Therefore, the available schedules for this example are

```
        //& (P[1], machine1.ececs.uc.edu, 100);
        //& (P[2], machine2.ececs.uc.edu, 100);
S1      machine[1]= machine1.ececs.uc.edu;
S2      machine[2]= machine2.ececs.uc.edu;
        // getPoly returns the array after storing the coefficients
S3      for (i = 1 to 2) do
S4      P[i] = getPoly(machine[i]);
        // The function multiplyPolynomials, multiplies the polynomial
S5      Product = multiplyPolynomial(P[1],P[2]);
```

Fig. 2. Carrying Data Strategy

(S1,S2,S3,S4), (S2,S1,S3,S4), (S2,S3,S1,S4)

The data carried by *Carrying Data Strategy* for the three different schedules are 36040 bytes, 32060 bytes and 52060 bytes. However, when *Partial Evaluation Strategy* is considered, the sizes of data moved around are 10020 bytes, 6040 bytes and 26040 bytes respectively. This proves that *Partial Evaluation Strategy* is better if size of the expected result is lesser than the input.

```
        //& (Employee, atp://stationB.ececs.uc.edu, 1000);
        //& (Loans, atp://stationA.ececs.uc.edu, 5);
        //& (LA, atp://stationC.ececs.uc.edu, 2000);
S1      Employee = getEmployeeRecords(salary < 1000);
S2      Loans = getLoansRecords(name =" ComputerLoan");
S3      LA = getLoansAvailedRecords(loan = Loans.loan_no);
S4      resultSet =getResultSet(Employee,LA,"salary < 1000
                    and loan_no = loan");
```

Fig. 3. Partial Evaluation Strategy

4 Results

Our framework was implemented using Jikes, the Java compiler from IBM and the Aglets Software Development Kit from IBM, Japan. In the database examples, we have assumed that the agent can carry the result of a query but cannot carry the entire database and make it persistent in some other location. We used a local network consisting of 3 Sun Ultra 5 machines and a Sun Ultra 1 machine for testing the framework. For each example, we made the compiler generate the best schedule for both CDS and PES strategies and compared the results.

In the tables 2 and 3, S refers to Source and M1, M2 and M3 are three different machines where the data is distributed. Columns *S-M1* etc., denote the size of data (including the size of agent code) carried from *S* to *M1* in Kilobytes.

Table 1. Result Set size (number of records) of the Individual Queries and the Join

S.No	Query	Result Size	Join Size
1	$salary > 4000$	4362	1540
	$loanId = 4$	1763	
2	$salary > 15000$	2665	2
	$loanId = 5$	3	
3	$salary < 100$	13	3
	$loanId = 2$	1783	

Table 1 gives the size of the result set for various queries for a version of the example given in figure 3. Table 2 gives the results for CDS and PES for various queries. From the table it is evident that the time taken by PES will be lesser than CDS, when the result size is far less than the input data size. Table 3 illustrates the results of the CDS and PES strategy for the example given in figure 2. It is evident from the table that when the result size is large, CDS outperforms the PES.

Table 2. Comparison of CDS and PES for the Query Example

Query	Carrying Data Strategy					Partial Evaluation Strategy				
	S-M1 (KB)	M1-M2 (KB)	M2-M3 (KB)	M3-S (KB)	Time sec	S-M1 (KB)	M1-M2 (KB)	M2-M3 (KB)	M3-S (KB)	Time sec
1	1.3	2.3	239.5	277	80	1.2	2.2	239.4	73.9	51
2	1.3	2.3	149.3	149	52	1.2	2.2	149.2	2.3	23
3	1.3	2.3	3.2	41	45	1.2	2.2	3.0	2.3	18

Table 3. Comparison of CDS and PES for Polynomial Multiplication

Polynomial Degree			Carrying Data Strategy					Partial Evaluation Strategy				
I	II	Result	S-M1 (KB)	M1-S (KB)	M2-S (KB)	Total Size(KB)	Time sec	S-M1 (KB)	M1-M2 (KB)	M2-S (KB)	Total Size	Time sec
10	10	100	2.4	2.6	2.6	10.0	2.3	2.9	3.1	4.1	10.2	4.5
10	1000	10000	2.4	2.7	12.7	20.3	2.7	2.9	3.1	103.3	109.4	7.2
50	1000	50000	2.4	3.1	12.7	20.7	2.9	2.9	3.5	534.9	541.5	42.3

5 Related Work and Conclusion

Traveler [3] and StratOSphere [4] support the creation of mobile agents and allow distributed processing across several hosts. Jaspal Subhlok et.al [2], present a solution for automatic selection of nodes for executing a performance critical parallel application in a shared network. A. Iqbal et.al [1] find the shortest path between the start node and the end node to get optimal migration sequence. This work is oriented towards reducing the distance traveled rather than minimizing the data carried.

In contrast, this work shows how compiler based optimizations will help achieve efficient scheduling of the mobile agents. Compiler based analysis helps in reducing the amount of data carried through the network by identifying those statements that can be partially evaluated and those statements for which data can be carried. In addition, it can generate the most efficient schedule under the program dependence constraints.

References

[1] Iqbal A, Baumann J, and Straβer M. Efficient Algorithms to Find Optimal Agent Migration Strategies. Technical report, IPVR, University of Stuttgart, 1998.

[2] Jaspal Subhlok, Peter Lieu, and Bruce Lowekamp. Automatic Node Selection for High Performance Applications on Networks. In *Principles and Practice of Parallel Programming (PPoPP)*, pages 163–172, 1999.

[3] Brian Wims and Cheng-Zhong Xu. Traveler: A Mobile Agent Infrastructure for Wide Area Parallel Computing. In *Proceedings of the IEEE Joint Symposium ASA/MA'99: First Int. Symp. on Agent Systems and Applications (ASA'99) and Third Int. Symp. on Mobile Agents (MA'99)*, 1999.

[4] Daniel Wu, Divyakant Agrawal, and Amr El Abbadi. Stratosphere:Mobile Processing of Distributed Objects in Java. In *ACM/IEEE international conference on Mobile Computing and Networking (MOBICOM)*, pages 121–132, 1998.

A Bytecode Optimizer to Engineer Bytecodes for Performance*

Jian-Zhi Wu and Jenq Kuen Lee

Department of Computer Science, National Tsing-Hua University, Hsinchu, Taiwan

Abstract. We are interested in the issues on the bytecode transformation for performance improvements on programs. In this work, we focus on the aspect of our bytecode to bytecode optimizing system on the ability to optimize the performances of hardware stack machines. Two categories of the problem are considered. First, we consider the stack allocations for intra-procedural cases with a family of Java processors. We propose a mechanism to report an allocation scheme for a given size of stack allocation according to our cost model. Second, we also extend our framework for stack allocations to deal with inter-procedural cases. Our initial experimental test-bed is based on an ITRI-made Java processor and Kaffe VM simulator[2]. Early experiments indicate our proposed methods are promising in speedup Java programs on Java processors with a fixed size of stack caches.

1 Introduction

In our research work, we are interested in investigating issues related to improving system performances via bytecode engineering. We are working on to develop a bytecode to bytecode optimizing system called JavaO. In this paper, we address the aspect of our bytecode to bytecode optimizing system on the ability to optimize the performances of hardware stack machines. Our system takes a bytecode program and returns an optimized bytecode program, which runs well for hardware stack machines. We consider stack frame allocations for Java processors with a fixed-size stack cache. Our work gives solutions for both intra-procedural and inter-procedural cases.

2 Machine Architectures

Our hardware model basically is a direct implementation of the frame activation allocations of a software JVM. In order to perform a direct hardware implementation of the frame allocation and reference schemes, we have our hardware Java processor contain a stack cache, which all frame structures are created and

* This paper is supported in part by NSC of Taiwan under grant no. NSC 89-2213-E-007-049, by MOE/NSC program for promoting academic excellence of universities under grant no. 89-E-FA0414, and by ITRI of Taiwan under grant no. G388029.

S.P. Midkiff et al. (Eds.): LCPC 2000, LNCS 2017, pp. 377–381, 2001.

destroyed on. ITRI Java processor[4] is an example belonging to this type of machine. An implementation to make JVM a hardware machine but with a fixed size of stack cache will likely fall into this category. Besides, the Java processor with the limited size of a stack cache must have mechanisms like spill and fill to deal with the stack overflow and underflow problem.

3 Reverse Object-Unfolding

We proposed a concept called reverse object-unfolding to be applied for stack cache optimization. The method is to have the reversal effect of a well-known optimization scheme called structure or object unfolding. Our consideration is that we need to combine these two techniques together to have an effective allocation scheme for a fixed size stack cache. Traditionally, a structure or object unfolding technique can be used to transform heap accesses into stack accesses. For the Java processor with a fixed-size stack, unlimited employment of structure unfolding techniques however will result in the size of local variables excelling the stack size. Thus it will reduce the performance gains. To solve this problem, we propose to first perform object unfolding, and then to assign heuristic weights to local variables, after that we then perform the reverse object-unfolding scheme to reverse a certain amount of local variables with least weights into a heap object. In this way, we can improve our performances on stack machines.

4 Inter-procedural Stack Cache Optimization

We also extend our work for inter-procedural cases. We model this problem into equations and propose a heuristic algorithm to solve the stack allocation problems for inter-procedural cases. In our heuristic algorithm, we use domain decomposition approach to partition the call graph into intervals, and then to find the solution for each small interval of the call-graph. We then can find the best solution in each partition. In addition, we use profiling scheme to obtain the call-graph information in our design. As our goal is to find the minimum total cost of reference costs spilled in the memory and the stack flush costs. We need to deal with this problem for an assumed scheme for a stack flush. In the following, we will assume that the basic unit for each stack flush is based on the amount of stack allocations for each method. When a new method is invoked, the system will check if addition of the stack allocation (which is the sum of the activation record and the operand stack allocation) of the new method into current watermark of the stack cache will overflow the stack cache. If the new watermark overflows the stack cache, a stack flush is done to spill the elements in the stack into memory. The spill is done by considering the stack allocation of each method as a basic unit. The system will spill as many units as needed to accommodate the new method invocation. Figures 1 gives the outline of our heuristic algorithm.

Algorithm 1
Input: 1. The call graph G with n methods,f_1,f_2, ...,f_n
2. The block size W for program partitions.
Output: A heuristic scheduling for stack allocation.
Begin
Step 1: To use domain decomposition approach to partition the call graph of the program into block partitions
Step 2: To find the solution for each small block of the call-graph. This can be done by solving the smaller problem for each sub-block, $Block_i$:
$Minimize : (\Sigma_{g_j \in Block_i} \delta^{'}(g_j, \boldsymbol{e_j})) + \phi(Block_i, < g_1, g_2, \ldots, g_n >, < \boldsymbol{e_1}, \boldsymbol{e_2}, \ldots, \boldsymbol{e_n} >)$,
where $\boldsymbol{e_j}$ *is a possible stack allocation for* g_j.
Step 3: Merging the assignments cross different program blocks.
Step 4: Iterative back to Step 2 until the tunings no longer result in performance improvements.
End

Fig. 1. A heuristic algorithm based on domain decomposition for inter-procedural stack allocations

5 Experiments

In the current status of our experiment, we have built-in the cycle time and cost model estimation for an ITRI-made Java processor into a simulator based on Kaffe VM[2] to estimate the cycle time with or without our optimization. The ITRI-made Java processor belongs to the family of Java processor with a fixed-size stack[4]. We are in the process of implementing our bytecode optimizer called JavaO. As our software with bytecode transformations was still not with the full strength to do the optimizations of our proposed algorithms automatically yet at this moment, we report the results by hand-code transformations. The hand-code transformation is done according to our proposed algorithm, and the transformed codes will then run through the simulator to estimate the cycle time.

We look at the "JGFArithBench.java" in the Java Grande benchmarks suite[1] for intra-procedural cases. We contrast the performance effects for two versions. In the base version noted as "w/o Reorder", the reverse object-unfolding is done by converting the last few local variables in the original byte code programs. In the second version of our performance noted as "w/ Reorder", the reverse object-unfolding is done according to our algorithm in Section 3 to choose the local variables for reverse unfolding. Figure 2 shows the intra-procedural results in reversing different number of slots for `JGFrun` method in `JGFArithBench` class. In this case, using our cost model for weight assignments to local variables has the improvement over the base version approximately 59% in the best case.

Now let's consider the inter-procedural cases. We illustrate the performance effects for the following example. Suppose we have methods A, B, C, and D. Methods A calls B, and after B returns then calls C, and then, after C returns, calls D. We also suppose that the size of the stack cache is 64 and the size of the frame state for context switch is 6. In this example, we assume the stack frame size for A is the same as the `JGFrun` method of `JGFArithBench` class (=41), and for B, C, and D are all 24. In addition, we also assume the frequency of

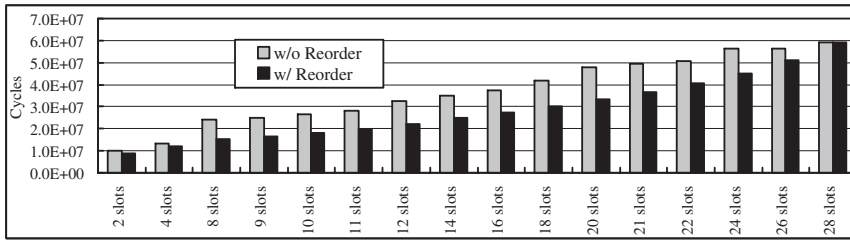


Fig. 2. The intra-procedural results in reversing different number of slots for JGFrun method in JGFArithBench class

Table 1. The compare of stack flushes without our padding scheme, and the version with out padding scheme for inter-procedural cases.

Case	Hardware	Heuristic
Stack Flushes	3 (in A)	0
Stack Spills	3 (in A)	0
Other Expense	0	reverse 2 slots in A
Total Cost (cycles)	1476	529

local variables in A is the same as the `JGFrun` method. In this case, the ITRI Java processor occurred 3 flushes and 3 spills. However, after performing our heuristic algorithm for this program, we reverse 2 local variable slots in A into heap accesses. In summary, using the stack flush schemes without padding the procedure frame will consume an extra 1476 cycle time for stack flushes, while using our inter-procedural frame padding scheme in Section 4 only consumes an extra 529 cycles. Table 1 illustrates this performance result. Our scheme significantly reduces the stack flush overhead in this case.

6 Conclusion

In this paper, two categories of the solutions were reported. We considered the stack allocations for intra-procedural and inter-procedural cases with a family of Java processors. We feel our proposed schemes are important advances on the issues of stack cache allocations on modern stack machines implementing JVM. Currently, we are in the process of implementing our software algorithm by incorporating softwares into a package called JavaClass API[3]. Early experiments indicate our proposed methods are promising in speedup Java programs on Java processors with a fixed size of stack caches.

References

[1] J. M. Bull et al. *A methodology for Benchmarking Java Grande Applications*, ACM Java Grande Conference, 1999.

[2] Transvirtual Technologies, Inc., *Kaffe Open VM Desktop Edition*, this package is made available at `http://www.kaffe.org`.
[3] Markus Dahm, *JavaClass API*, Freie University Berlin, this package is made available at `http://www.inf.fu-berlin.de/~dahm/JavaClass/index.html`.
[4] "User Manuals for Java CPU", Technical Reports, CCL, Industrial Technology and Research Institute, Taiwan, 1999.

Author Index