Vikram Adve
María Jesús Garzarán
Paul Petersen (Eds.)

LNCS 5234

Languages and Compilers for Parallel Computing

20th International Workshop, LCPC 2007
Urbana, IL, USA, October 2007
Revised Selected Papers

Springer

Vikram Adve
María Jesús Garzarán
Paul Petersen (Eds.)

Languages and Compilers for Parallel Computing

20th International Workshop, LCPC 2007
Urbana, IL, USA, October 11-13, 2007
Revised Selected Papers

Springer

Volume Editors

Vikram Adve
María Jesús Garzarán
University of Illinois at Urbana-Champaign
Department of Computer Science
Thomas M. Siebel Center for Computer Science
201 N. Goodwin Ave, Urbana, IL, 61801, USA
E-mail: {vadve,garzaran}@cs.uiuc.edu

Paul Petersen
Intel Corporation
1906 Fox Drive, Champaign, IL, 61820, USA
E-mail: paul.petersen@intel.com

Library of Congress Control Number: 2008932585

CR Subject Classification (1998): D.1.3, C.2.4, D.4.2, H.3.4, D.2

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN	0302-9743
ISBN-10	3-540-85260-3 Springer Berlin Heidelberg New York
ISBN-13	978-3-540-85260-5 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12455096 06/3180 5 4 3 2 1 0

Preface

It is our pleasure to present the papers from the 20th International Workshop on Languages and Compilers for Parallel Computing! For the past 19 years, this workshop has been one of the primary venues for presenting and learning about a wide range of current research in parallel computing. We believe that tradition has continued in this, the 20th year of the workshop.

This year, we received 49 paper submissions from 10 countries. About a quarter of the papers (12 out of 49) included authors from industry. We selected 23 papers to be presented at the workshop, for an acceptance rate of 47%, which was similar to that of the last two years. Each paper received at least three reviews, with about two-thirds of the papers getting four or more reviews each. Most papers also received at least one review from an external reviewer. The committee held a full-day teleconference to discuss the reviews and select papers. Program Committee members who had a conflict with a paper left the call when that paper was being discussed. There were seven submissions that included Program Committee members as co-authors. These papers were evaluated more stringently and four of seven were accepted.

The workshop this year also included two exciting special events. First, David Kirk, Chief Scientist of nVidia and a member of the National Academy of Engineering, gave a keynote talk on using highly multithreaded graphics processors for accelerating general-purpose parallel computing applications. Kirk and nVidia have led the drive to make the high parallelism in graphics processors more easily accessible for a wide range of applications beyond traditional graphics processing, and this talk gave LCPC attendees a valuable perspective on the potential of this work.

Second, a special panel was held on Friday morning to commemorate the 20th year of LCPC. This panel, organized and moderated by Steering Committee Chair David Padua, was scheduled for an entire session to allow seven leaders in parallel computing to give their perspective on how the field has evolved over the past 20 years, and what the major challenges are for the future. The panel included a number of luminaries in parallel computing – Arvind (MIT), David Kuck (Intel), Monica Lam (Stanford University), Alexandru Nicolau (University of California Irvine), Keshav Pingali (University of Texas, Austin), Burton Smith (Microsoft Research), and Michael Wolfe (The Portland Group). Our thanks to David Padua for organizing this panel.

We would like to thank the many people who contributed valuable time and effort to making LCPC 2007 a success. Most importantly, we want to thank all the members of the community who submitted papers to the workshop. This workshop is, in the end, an (incomplete) representation of the results of their hard work. Second, the Program Committee worked hard to review 12–13 papers each and to participate in the all-day Program Committee teleconference. The quality of the technical program owes much to their effort. In his role as Steering Committee Chair, David Padua provided valuable guidance and support. He also lent his stack of 19 previous LCPC proceedings to mark the occasion!

Finally, Sheila Clark put in tremendous effort behind the scenes to manage many organizational details for the workshop and we would not have been able to pull it off without all her help.

October 2007

Vikram Adve
María Jesús Garzarán
Paul Petersen

Organization

LCPC 2007 was organized by the Steering and Program Committees.

General/Program Co-chairs

Vikram Adve	University of Illinois at Urbana-Champaign
María Jesús Garzarán	University of Illinois at Urbana-Champaign
Paul Petersen	Intel Corporation

Program Committee

Vikram Adve	University of Illinois at Urbana-Champaign
Gheorghe Almási	IBM Research
José Nelson Amaral	University of Alberta, Canada
Eduard Ayguadé	Universitat Politécnica de Catalunya, Spain
Gerald Baumgartner	Louisiana State University
Călin Caşcaval	IBM Research
María Jesús Garzarán	University of Illinois at Urbana-Champaign
Sam Midkiff	Purdue University
Paul Petersen	Intel Corporation
J. Ramanujam	Louisiana State University
P. Sadayappan	Ohio State University
Peng Wu	IBM Research

Steering Committee

Rudolf Eigenmann	Purdue University
Alex Nicolau	UC Irvine
David Padua	University of Illinois at Urbana-Champaign
Lawrence Rauchwerger	Texas A&M University

Sponsoring Institution

University of Illinois at Urbana-Champaign
Siebel Center for Computer Science
Urbana, Illinois, October 11–13, 2007

Referees

V. Adve
G. Agrawal
G. Almasi
J.N. Amaral
E. Ayguadé
M. Baskaran
G. Baumgartner
P. Berube
R. Bocchino
U. Bondhugula
J. Brodman
C. Caşcaval
L. Ceze
A. Chakravarti
A. Cohen
G. Dozsa
A. Eichenberger
R. Eigenmann
B. Fraguela
F. Franchetti
R. Garg
M.J. Garzarán
R. Ghiya
M. Girkar
M. Gonzalez
A. Gotsman
J. Guo
S. Hack
J. Hoeflinger
A. Kejariwal
G. Kondrak
D. Koppelman
S. Krishnamoorthy
K. Laufer
O. Lhotak
X. Martorell
L. Meadows
J. Mellor-Crummey
S. Midkiff
R. Niewiadomski
D. Nikolopoulos
R. Nishtala
S. Pande
S. Parthasarathy
P. Petersen
C. Pickett
K. Pingali
J. Ramanujam
A. Ramirez
A. Rountev
R. Rugina
P. Sadayappan
E. Salami
R. Schreiber
P. Sdayappan
A. Sidelnik
K. Strauss
Z. Sura
D. Tarditi
X. Tian
P. Unnikrishnan
N. Vasilache
R. Wilhelm
P. Wu
Y. Zheng
P. Zhao

LCPC at Illinois

1989

2007

Keynote Presentation: NVIDIA CUDA Software and GPU Parallel Computing Architecture

David Kirk, Chief Scientist, nVidia Corp.

Abstract. In the past, graphics processors were special-purpose hard-wired application accelerators, suitable only for conventional rasterization-style graphics applications. Modern GPUs are now fully programmable, massively parallel floating point processors. This talk describes NVIDIA's massively multithreaded computing architecture and CUDA software for GPU computing. The architecture is a scalable, highly parallel architecture that delivers high throughput for data-intensive processing. Although not truly general-purpose processors, GPUs can now be used for a wide variety of compute-intensive applications beyond graphics.

Panel I: How Is Multicore Programming Different from Traditional Parallel Programming?

Panelists:
Arch Robison (Intel)
Călin Caşcaval (IBM Research)
Wen-mei Hwu (University of Illinois at Urbana-Champaign)
Hironori Kasahara (Waseda University, Japan) and
Gudula Rünger (Chemnitz University of Technology, Germany)
Moderator:
Vikram Adve (University of Illinois at Urbana-Champaign)

Panel II: What Have We Learned After 20 LCPCs?

Panelists:
David Kuck (Intel)
Arvind (MIT)
Monica Lam (Stanford University)
Alexandru Nicolau (University of California Irvine)
Keshav Pingali (University of Texas, Austin)
Burton Smith (Microsoft Research) and
Michael Wolfe (the Portland Group)
Moderator:
David Padua (University of Illinois at Urbana-Champaign)

Table of Contents

Compiler-Enhanced Incremental Checkpointing

Greg Bronevetsky[1], Daniel Marques[2], Keshav Pingali[2], and Radu Rugina[3]

[1] Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94551, USA
greg@bronevetsky.com

[2] Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712, USA
daniel@ices.utexas.edu, pingali@cs.utexas.edu

[3] Department of Computer Science, Cornell University, Ithaca, NY 14850, USA
rugina@cs.cornell.edu

Abstract. As modern supercomputing systems reach the peta-flop performance range, they grow in both size and complexity. This makes them increasingly vulnerable to failures from a variety of causes. Checkpointing is a popular technique for tolerating such failures in that it allows applications to periodically save their state and restart the computation after a failure. Although a variety of automated system-level checkpointing solutions are currently available to HPC users, manual application-level checkpointing remains by far the most popular approach because of its superior performance. This paper focuses on improving the performance of automated checkpointing via a compiler analysis for incremental checkpointing. This analysis is shown to significantly reduce checkpoint sizes (upto 78%) and to enable asynchronous checkpointing.

1 Introduction

The dramatic growth in supercomputing system capability from the tera-flop to the peta-flop range has resulted in a dramatic increase in system complexity. While efforts have been made to limit the complexity of the Operating System used by these machines, their component counts have continued to grow. Even as systems like BlueGene/L [3] and the upcoming RoadRunner grow to more than 100,000 processors and tens of TBs of RAM, future designs promise to exceed these limits by large margins. While large supercomputers are made from high-quality components, increasing components counts make them vulnerable to faults, including hardware breakdowns [11] and soft errors [6].

Checkpointing is a popular technique for tolerating failures. The state of the application is periodically saved to reliable storage and on failure, the application rolls back to a prior state. However, automated checkpointing can be very expensive due to the amount of data saved and the amount of time that the application loses while being blocked. For example, dumping all of RAM on a 128k-processor BlueGene/L supercomputer to a parallel file system would take approximately 20 minutes [9]. Incremental checkpointing [10] is one technique that can reduce the cost of checkpointing. A runtime monitor keeps track of any application writes. If it detects that a given memory region has not been modified between two adjacent checkpoints, this region is omitted from the second checkpoint, thus reducing the amount of data that needs to be saved. Possible monitors that have been explored in the past include virtual memory fault handlers [5], page table dirty bits and cryptographic encoding techniques [4].

When virtual memory fault handlers are used to track application writes, it is possible to further optimize the checkpointing process via a technique called "copy-on-write checkpointing" or more generally, "asynchronous checkpointing". At each checkpoint all pages that need to be checkpointed are marked non-writable and placed on a write-out queue. The application is then allowed to continue executing, while a separate thread asynchronously saves pages on the write-out queue. When the checkpointing thread is finished saving a given page, the page is marked writable. If the application tries to write to a page that hasn't yet been saved, the segmentation fault handler is called, a copy of the page is placed in the write-out queue and the application is allowed to resume execution. The result is that checkpointing is spaced out over a longer period of time, reducing the pressure on the I/O system, while allowing the application to continue executing.

In contrast to prior work, which uses runtime techniques for monitoring application writes, this paper presents a compile-time analysis for tracking such writes. Given an application that has been manually annotated with calls to a `checkpoint` function, for each array the analysis identifies points in the code such that either

– there exist no writes to the array between the point in the code and the next checkpoint and/or
– there exist no writes to the array between the last checkpoint and the point in the code

When the analysis detects that a given array is not modified between two checkpoints, this array is omitted from the second checkpoint. Furthermore, the analysis enables asynchronous checkpointing by allowing the checkpointing thread to save a given array during the period of time while there are no writes to it. Because the compiler analysis can identify write-free regions that begin before the checkpoint itself, it allows asynchronous checkpointing to begin earlier than is possible with purely runtime solutions. However, because it works at array granularity rather than the page- or word-granularity of runtime monitoring mechanisms, it can be more conservative in its decisions. Furthermore, the analysis makes the assumption that a checkpoint is taken every time the `checkpoint` function is called, which makes it more complex for users to target a specific checkpointing frequency.

While prior work has looked at compiler analyses for checkpoint optimization [7] [12], it has focused on pure compiler solutions that reduce the amount of data checkpointed. Our work presents a hybrid compiler/runtime approach that uses the compiler to optimize certain portions of an otherwise runtime checkpointing solution. This allows us to both reduce the amount of data being checkpointed as well as support purely runtime techniques such as asynchronous checkpointing.

2 Compiler/Runtime Interface

Our incremental checkpointing system is divided into run-time and compile-time components. The checkpointing runtime may either checkpoint application memory inside of `checkpoint` calls or include an extra thread that checkpoints asynchronously. Two checkpointing policies are offered. Memory regions that do not contain arrays (a small portion of the code in most scientific applications) are saved in a blocking fashion during calls to `checkpoint`. Arrays are dealt with in an incremental and possibly asynchronous fashion, as directed by the annotations placed by the compiler. The compiler annotates the source code with calls to the following functions:

– `add_array(ptr, size)` - Called when an array comes into scope to identify the array's memory region.
– `remove_array(ptr)` - Called when an array leaves scope. Memory regions that have been added but not removed are treated incrementally by the checkpointing runtime.
– `start_chkpt(ptr)` - Called to indicate that the array that contains the address `ptr` will not be written to until the next checkpoint. The runtime may place this array on the write-out queue and begin to asynchronously checkpoint this array.
– `end_chkpt(ptr)` - Called to indicate that the array that contains the address `ptr` is about to be written to. The `end_chkpt` call must block until the checkpointing thread has finished saving the array. It is guaranteed that there exist no writes to the array between any checkpoint and the call to `end_chkpt`.

Overall, the runtime is allowed to asynchronously checkpoint a given array between calls to `start_chkpt` and `end_chkpt` that refer to this array. If `start_chkpt` is not called for a given array between two adjacent checkpoints, this array may be omitted from the second checkpoint because it is known that it was not written to between the checkpoints.

For a more intuitive idea of how this API is used, consider the transformation in Figure 1. The original code contains two `checkpoint` calls, with assignments to arrays `A` and `B` in between. The code within the `...`'s does not contain any writes to `A` or `B`. It is transformed to include calls to `start_chkpt` and `end_chkpt` around the writes. Note that while `end_chkpt(B)` is placed immediately before the write to `B`, `start_chkpt(B)` must be placed at the end of `B`'s write loop. This

Original Code	Transformed Code
`checkpoint();`	`checkpoint();`
`...`	`...`
`A[...]=...;`	`end_chkpt(A);`
`...`	`A[...]=...;`
`for(...) {`	`start_chkpt(A);`
` ...`	`...`
` B[...]=...;`	`for(...) {`
` ...`	` ...`
`}`	` end_chkpt(B);`
`...`	` B[...]=...;`
`checkpoint();`	` ...`
	`}`
	`start_chkpt(B);`
	`...`
	`checkpoint();`

Fig. 1. Transformation example

is because a `start_chkpt(B)` call inside the loop may be followed by writes to `B` in subsequent iterations. Placing the call immediately after the loop ensures that this cannot happen.

3 Compiler Analysis

The incremental checkpointing analysis is a dataflow analysis that consists of forward and backward components. The forward component, called the *Dirty Analysis*, identifies the first write to each array after a checkpoint. The backward, called the *Will-Write* analysis, identifies the last write to each array before a checkpoint.

3.1 Basic Analysis

For each array at each node n in a function's control-flow graph(CFG) the analysis maintains two bits of information:

– $mustDirty[n](array)$: True if there *must* exist a write to *array* along *every* path from a `checkpoint` call to this point in the code; False otherwise. Corresponds to the dataflow information immediately *before* n.
– $mayWillWrite[n](array)$: True if there *may* exist a write to *array* along *some* path from a this point in the code to a `checkpoint` call; False otherwise. Corresponds to the dataflow information immediately *after* n.

This information is propagated through the CFG using the dataflow formulas in Figure 2. The Dirty and Will-Write analyses start at the top and bottom of each function's CFG, respectively, in a state where all arrays are considered to be clean (e.g. consistent with the previous, next checkpoint, respectively). They then propagate forward and backward, respectively, through the CFG, setting each array's write bit to $True$ when it encounters a write to this array. When each analysis reaches a `checkpoint` call, it resets the state of all the arrays to $False$. For the Dirty Analysis this is because all dirty arrays will become clean because they are checkpointed. For the WillWrite Analysis this is because at the point immediately before a checkpoint there exist no writes to any arrays until the next checkpoint, which is the checkpoint in question.

$$mustDirty[n](array) = \begin{cases} False & \text{if } n = \text{first node} \\ \bigcap_{m \in pred(n)} mustDirtyAfter[m](array) & \text{otherwise} \end{cases}$$

$$mustDirtyAfter[m](array) = [\![m]\!](mustDirty[m](array), array)$$

$$mayWillWrite[n](array) = \begin{cases} False & \text{if } n = \text{last node} \\ \bigcup_{m \in succ(n)} mayWillWriteBefore[m](array) & \text{otherwise} \end{cases}$$

$$mayWillWriteBefore[m](array) = [\![m]\!](mayWillWrite[m](array), array)$$

Statement m	$[\![m]\!](val, array)$
`array[expr] = expr`	$True$
`checkpoint()`	$False$
other	val

Fig. 2. Dataflow formulas for Dirty and Will-Write analyses

The application source code is annotated with calls to `start_chkpt` and `end_chkpt` using the algorithm in Figure 3. Such calls are added in three situations. First, `end_chkpt (array)` is inserted immediately before node n if n is a write to *array* and it is not preceded by any other write to *array* along *some* path that starts at a call to `checkpoint` and ends with node n. Second, `start_chkpt(array)` is inserted immediately after node n if n is a write to *array* and there do not exist any more writes to *array* along *any* path that starts with n and ends at a `checkpoint` call. Third, a `start_chkpt(array)` is inserted on a CFG branching edge $m \rightarrow n$ if $mayWillWrite[n](array)$ is true at m, but false at n, due to merging of dataflow information at branching point m. This treatment is especially important when an array is being written inside a loop. In this case, $mayWillWrite[n](array)$ is true at all points in the loop body, since the array may be written in subsequent loop iterations. The flag becomes false on the edge that branches out of the loop, and the compiler inserts the `start_chkpt(array)` call on this edge.

Because the Dirty analysis is based on *must-write* information, `end_chkpt` calls are conservatively placed as late as possible after a checkpoint. Furthermore, the Will-Write analysis' use of *may-write* information conservatively places `start_save` calls as early as possible before a checkpoint.

To provide an intuition of how the analysis works, consider the example in Figure 4. In particular, consider the points in the code where *mustDirty*

```
foreach (array $array$), foreach (CFG node $n$) in application
    // if node $n$ is the first write to $array$ since the last checkpoint call
    if($mustDirty[n](array) = False \wedge mustDirtyAfter[n](array) = True$)
        place end_chkpt($array$) immediately before $n$
    // if node $n$ is the last write to $array$ until the next checkpoint call
    if($mayWillWriteBefore[n](array) = True \wedge mayWillWrite[n](array) = False$)
        place start_chkpt($array$) immediately after $n$
    // if node $n$ follows the last write on a branch where $array$ is no longer written
    if($mayWillWriteBefore[n](array) = False \wedge$
      $\exists m \in pred(n).\ mayWillWrite[m](array) = True$)
        place start_chkpt($array$) on edge $m \rightarrow n$
```

Fig. 3. Transformation for inserting calls to `start_chkpt` and `end_chkpt`

and *mayWillWrite* change from *False* to *True*. These are the points where `end_chkpt` and `start_chkpt` calls are inserted.

3.2 Loop-Sensitive Analysis

While the basic analysis performs correct transformations, it has performance problems when it is applied to loops. This can be seen in the transformed code in Figure 4. While `start_chkpt(B)` is placed immediately after the loop that writes to `B`, `end_chkpt(B)` is placed inside the loop, immediately before the write to `B` itself. This happens because the placement of `end_chkpt` depends on *must-write* information, instead of the *may-write* information used in placing `start_chkpt`. While this placement is conservative, it becomes problematic in the case where the first post-checkpoint write to an array happens in a small, deeply-nested loop, which are very common in scientific computing. In this case `end_chkpt` will be called during each iteration of the loop, causing a potentially severe overhead.

Original Code	Code with Dirty States	Code with Will-Write States	Transformed Code
`checkpoint();`	`checkpoint(); [A→F,B→F]`	`checkpoint(); [A→T,B→T]`	`checkpoint();`
`...`	`... [A→F,B→F]`	`... [A→T,B→T]`	`...`
`A[...]=...;`	`A[...]=...; [A→F,B→F]`	`A[...]=...; [A→F,B→T]`	`end_chkpt(A);`
`...`	`... [A→T,B→F]`	`[A→F,B→T]`	`A[...]=...;`
`for(...) {`	`for(...) { [A→T,B→F]`	`for(...) { [A→F,B→T]`	`start_chkpt(A);`
`...`	`... [A→T,B→F]`	`... [A→F,B→T]`	`...`
`B[...]=...;`	`B[...]=...; [A→T,B→F]`	`B[...]=...; [A→F,B→T]`	`for(...) {`
`...`	`... [A→T,B→T]`	`... [A→F,B→T]`	`...`
`}`	`} [A→T,B→T]`	`} [A→F,B→T]`	`end_chkpt(B);`
`...`	`... [A→T,B→F]`	`... [A→F,B→F]`	`B[...]=...;`
`checkpoint();`	`checkpoint(); [A→T,B→F]`	`checkpoint(); [A→F,B→F]`	`...`
			`}`
			`start_chkpt(B);`
			`...`
			`checkpoint();`

Fig. 4. Analysis example

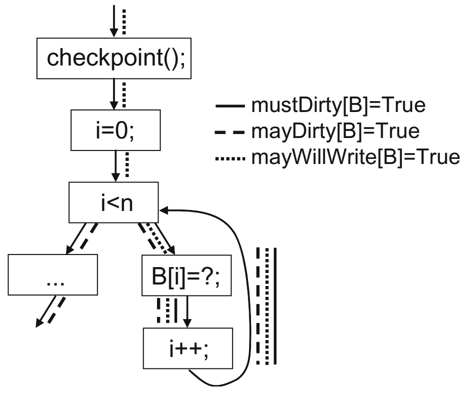


Fig. 5. Dataflow pattern for writes inside loops

To address this problem the above analysis was augmented with a loop-detection heuristic, shown in Figure 5. This heuristic uses may-Dirty information, in addition to the must-Dirty and may-WillWrite information of Section 3 and identifies the patterns of dataflow facts that must hold at the top of the first loop that writes to an array after a checkpoint. Figure 5 contains the CFG of such a loop and identifies the edges in the CFG where the various dataflow facts are $True$. It can be seen that the pattern at node $i < n$ is:

– $mustDirty[i < n](B) = False$
– $mayDirty[i < n](B) = True$
– $mayWillWrite[i < n](B) = True$
– $pred(i < n) > 1$

Furthermore, the CFG edge that points to $i < n$ from outside the loop is the one coming from the predecessor p where $mustDirtyAfter[p](B) = False$. Thus, by placing `end_chkpt(B)` on this incoming edge we can ensure both that `end_chkpt(B)` is called before any write to `B` and that it is not executed in every iteration of the loop.

Since this heuristic only applies to loops, it does not place `end_chkpt(A)` before the write to `A` in Figure 1. As such, we need to use both rules to ensure that `end_chkpt` is placed conservatively. However, if both rules are used then the example in Figure 1 will get two `end_chkpt(B)` calls: one before `B`'s write loop and one before the write itself, negating the purpose of the loop-sensitive placement strategy. To prevent this from happening we propose an extra *EndChkpt-Placed* analysis that prevents `end_chkpt(array)` from being placed at a given node if there already exists an `end_chkpt(array)` on every path from any `checkpoint` call to the node. *EndChkpt-Placed* is a forward analysis that is executed as a separate pass from the Dirty and Will-Write passes. It maintains a bit of information for every array at every CFG node. $mustEndChkptPlaced[n](array)$ is set to $True$ if `end_chkpt(array)` is to be placed immediately before node n. It is set to $False$ if `start_chkpt(array)` is to be inserted at n. The later rule ensures that the "exclusion-zone" of a given insertion of `end_chkpt(array)` doesn't last past the next `checkpoint` call.

To implement this rule the loop-sensitive analysis maintains for each CFG node n the following additional dataflow information:

- $mayDirty[n](array)$: True if there *may* exist a write to *array* along *some* path from a `checkpoint` call to this point in the code; False otherwise. Corresponds to the dataflow information immediately *before* n.
- $mustEndChkptPlaced[n](array)$: true if *all* paths from any `checkpoint` call to this point in the code contain a point where a `end_chkpt(array)` call will be placed.

This information is computed as shown in Figure 6. The modified rules for placing `end_chkpt` calls are shown in Figure 7 and Figure 8 extends the example in Figure 1 with the new *mustEndChkptPlaced* information and the new placement of `end_chkpt` calls.

3.3 Inter-procedural Analysis

We have extended the above analysis with a context-insensitive, flow-sensitive inter-procedural analysis. The inter-procedural analysis works by applying the data-flow analysis from Section 3.2 to the CFG that contains all of the application's functions. When the analysis reaches a function call node for the first time, it computes a summary for this function. This is done by applying the dataflow analysis using the formulas in Figure 6 but with a modified lattice.

In addition to the standard $True$ and $False$, we introduce an additional $Unset$ state that appears below $True$ and $False$ in the lattice. All the dataflow facts for all arrays are initialized to $Unset$ at the start or end of the function (start for the forward analyses and end for the backward analysis). The standard analysis is then executed on the function using the extended lattice, with $Unset$ being treated as $False$ for the purposes of the EndChkpt-Placed analysis. If the state of a given array remains $Unset$ at the end of a given pass, this means that it was not modified by the pass. In the case of the Dirty and Will-Write analyses this means that the array is not written to inside the function. In the case of the EndChkpt-Placed analysis, this means that no `end_chkpt` calls are placed for this array inside the function. The function summary then is the dataflow facts for each array at the opposite end of the function: end for the forward analyses and start for the backward analysis. Function calls are processed by applying the function summary as a mask on all dataflow state. If $dataFlow[array] = Unset$ in the function summary, *array*'s mapping is not changed in the caller. However, if $dataFlow[array] = True$ or $False$ in the summary, the corresponding dataflow fact for *array* is changed to $True$ or $False$ in the caller.

$$mayDirty[n](array) = \begin{cases} False & \text{if } n = \text{first node} \\ \bigcup_{m \in pred(n)} mayDirtyAfter[m](array) & \text{otherwise} \end{cases}$$

$$mayDirtyAfter[m](array) = [\![m]\!](mayDirty[m](array), array)$$

$$mustEndChkptPlaced[n](array) =$$

$$\begin{cases} False & \text{if } n = \text{first node} \\ \quad \bigcap_{m \in pred(n)} mustEndChkptPlacedAfter[m](array) & \text{otherwise} \end{cases}$$

```
$mustEndChkptPlacedAfter[m](array) =$
    if $\neg$ $placeStartChkptNode(m, array)$ $\wedge$
      $\neg \exists l \in pred(m).\ placeStartChkptEdge(l, m, array))$ then
        $False$
    else if $(placeEndChkptNode(m, array)$ $\vee$
      $\exists l \in pred(m).\ placeEndChkptEdge(l, m, array))$ then
        $True$
    else $mustEndChkptPlaced[m](array)$
```

```
// end_chkpt(array) will be placed immediately before node $n$ if
$placeEndChkptNode(n, array) =$
    // node $n$ is the first write to $array$ since the last checkpoint
    $(mustDirty[n](array) = False \wedge mustDirtyAfter[n](array) = True)$
```

```
// end_chkpt(array) will be placed along the edge $m \rightarrow n$ if
$placeEndChkptEdge(m, n, array) =$
    // node $n$ is itself clean but predecessor $m$ is dirty, $n$ contains or is followed
    // by a write and predecessor $m$ is not itself preceded by $end\_chkpt(array)$
    $(mustDirty[n](array) = False \wedge mayDirty[n](array) = True \wedge$
     $mayWillWrite[n](B) = True \wedge mustDirtyAfter[m](array) = False \wedge$
     $mustEndChkptPlaced[m](array) = False)$
```

```
// start_chkpt(array) will be placed immediately after node $n$ if
$placeStartChkptNode(n, array) =$
    // node $n$ is the last write to $array$ until the next checkpoint
    $(mayWillWriteBefore[n](array) = True\ \wedge\ mayWillWrite[n](array) = False)$
```

```
// start_chkpt(array) will be placed along the edge $m \rightarrow n$ if
$placeStartChkptEdge(m, n, array) =$
    // node $n$ follows the last write to $array$ until the next checkpoint
    $(mayWillWriteBefore[n](array) = False \wedge mayWillWrite[m](array) = True)$
```

Fig. 6. Dataflow formulas for the loop-sensitive extension

```
foreach ($array$), foreach (CFG node $n$) in application
    if $placeEndChkptNode(n, array)$
        place end_chkpt($array$) immediately before $n$
    if $\exists m \in pred(n).\ placeEndChkptEdge(m, n, array)$
        place end_chkpt(array) on edge $m \rightarrow n$
```

Fig. 7. Loop-sensitive transformation for inserting calls to `end_chkpt`

Original Code	Code with Must-EndChkptPlaced States	Transformed Code
`checkpoint();`	`checkpoint(); [A→F,B→F]`	`checkpoint();`
`...`	`... [A→F,B→F]`	`...`
`A[...]=...;`	`A[...]=...; [A→F,B→F]`	`end_chkpt(A);`
`...`	`[A→T,B→F]`	`A[...]=...;`
`for(...) {`	`for(...) { [A→T,B→T]`	`start_chkpt(A);`
`...`	`... [A→T,B→T]`	`...`
`B[...]=...;`	`B[...]=...; [A→T,B→T]`	`end_chkpt(B);`
`...`	`... [A→T,B→T]`	`for(...) {`
`}`	`} [A→T,B→T]`	`...`
`...`	`... [A→T,B→T]`	`B[...]=...;`
`checkpoint();`	`checkpoint(); [A→T,B→T]`	`...`
		`}`
		`start_chkpt(B);`
		`...`
		`checkpoint();`

Fig. 8. Transformation example with loop-sensitive optimizations

4 Experimental Evaluation

4.1 Experimental Setup

We have evaluated the effectiveness of the above compiler analysis by implementing it on top of the ROSE [8] source-to-source compiler framework and applying it to the OpenMP versions [1] of the NAS Parallel Benchmarks [2]. We have used these codes in sequential mode and have focused on the codes `BT`, `CG`, `EP`, `FT`, `LU`, `SP`. We have omitted `MG` from our analysis since it uses dynamic multi-dimensional arrays (arrays of pointers to lower-dimensional arrays), which requires additional complex pointer analyses to identify arrays in the code. In contrast, the other codes use simple contiguous arrays, which require no additional reasoning power. Each NAS code was augmented with a `checkpoint` call at the top of its main compute loop and one immediately after the loop.

The target applications were executed on problem classes `S`, `W` and `A` (`S` is the smallest of the three and `A` the largest), on 4-way 2.4Ghz dual-core Opteron SMPs, with 16GB of RAM per node (Atlas cluster at the Lawrence Livermore National Laboratory). Each run was performed on a dedicated node and all reported results are averages of 10 runs. Each application was set to checkpoint

5 times, with the checkpoints spaced evenly throughout the application's execution. This number was chosen to allow us to sample the different checkpoint sizes that may exist in different parts of the application without forcing the application to take a checkpoint during every single iteration, which would have been unrealistically frequent.

The transformed codes were evaluated with a model checkpointing runtime that implements the API from Section 2 and simulates the costs of a real checkpointer. It performs the same state tracking as a real checkpointer but instead of actually saving application state, it simply sleeps for an appropriate period of time. One side-effect of this is the fact that our checkpointer does not simulate the overheads due to saving variables other than arrays. However, since in the NAS benchmarks such variables make up a tiny fraction of overall state, the resulting measurement error is small. Furthermore, since the model checkpointer can sleep for any amount of time, it can simulate checkpointing performance for a wide variety of storage I/O bandwidths.

The model checkpointer can be run in both a blocking and a non-blocking mode. In blocking mode the checkpointer does not spawn an asynchronous checkpointing thread but instead saves all live state inside the main thread's `checkpoint` calls. The model runtime can simulate both a basic checkpointer, which saves all state and an incremental checkpointer, which only saves the state that has changed since the last checkpoint. In particular, in incremental mode the the model checkpointer simulates the checkpointing of any array for which `start_chkpt` has been called since the last `checkpoint` call. Arrays for which `start_chkpt` has not been called are ignored.

In non-blocking mode, the model checkpointer spawns off an asynchronous checkpointing thread. The thread maintains a write-out queue of memory regions to save and continuously pulls memory regions the queue, sleeping for as long at it takes to save the next memory region to disk at the given I/O bandwidth. When the main thread calls `end_chkpt` for one array, it may be that the checkpointing thread is currently sleeping on another array. To control the amount of waiting time, the model checkpointer breaks arrays up into smaller blocks and simulates checkpointing at block granularity. `start_chkpt(array)` inserts `array`'s blocks at the end of the write-out queue. We also tried spreading the new array's blocks evenly across the queue but did not find a substantial or consistent performance difference between the two policies.

4.2 Incremental Checkpointing

We evaluated the effectiveness of the above analysis for incremental checkpointing by comparing the performance of two configurations of the model checkpointer's blocking mode:

– `CHKPT_ALL` - simulates the checkpointing of all application arrays inside each `checkpoint` call.
– `CHKPT_INCR` - same as `CHKPT_ALL` but omits any arrays for which `start_chkpt` hasn't been called since the last `checkpoint` call.

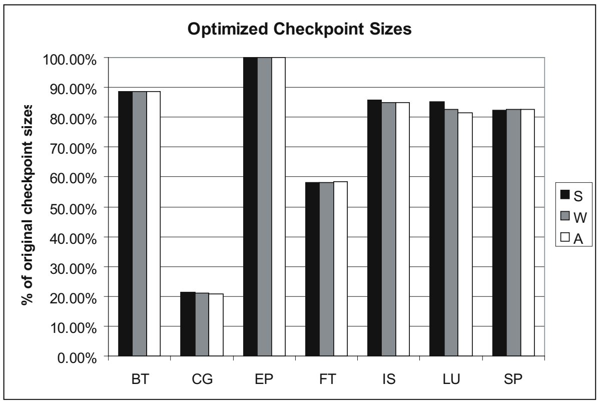


Fig. 9. Checkpoint sizes in `CHKPT_INCR` as a fraction of checkpoint sizes in `CHKPT_ALL`

We first compare the checkpoint sizes generated by the two modes. In the context of the NAS codes, which have a initialization phase, followed by a main compute loop, the primary effect of the above analysis is to eliminate write-once arrays from checkpointing. These are the arrays that are written to during the initialization phase and then only read from during the main compute loop. As such, since there do not exist any `start_chkpt` calls for these arrays during the main compute loop, they are only saved during the first checkpoint and omitted in subsequent checkpoints.

The measured reductions in checkpoint size are shown in Figure 9. It can be seen that even using array granularity, incremental checkpointing can result in a dramatic reduction in checkpoint sizes. Indeed, in `CG` checkpoints drop to below 22% of their original size, while `FT`'s drop to 60%. Other codes see drops from 0% to 18%. While there are small differences in the effect of incremental checkpointing across different problem sizes, the effect is generally small.

Figure 10 shows the the execution time of an application that uses incremental checkpointing as a percentage of the execution time of one that does not. We show only `CG` and `EP`, since the behavior of these codes is similar to that of other codes that have a large or a small checkpoint size reduction, respectively. The x-axis is the I/O bandwidth used in the experiments, ranging from 1 MB/s to 1 GB/s in multiples of 4 and including Infinite bandwidth. This range includes a variety of use-cases, including hard-drives (60MB/s write bandwidth) and 10 Gigabit Ethernet(1GB/s bandwidth). In the case of `EP`, although there is some difference in performance between the two versions, the effect is at noise level at all bandwidths, with no correlation between execution time and performance. However, for `CG`, the effect is quite dramatic, ranging from pronounced difference in execution times for low bandwidths, when the cost of checkpointing is important, to a much smaller difference for high bandwidths.

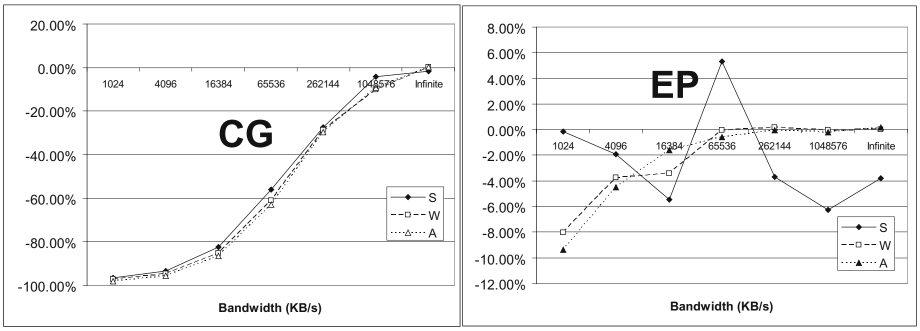


Fig. 10. Relative execution time differences between `CHKPT_INCR` and `CHKPT_ALL`

4.3 Asynchronous Checkpointing

We examined the performance characteristics of asynchronous checkpointing by looking at the relationship between the I/O bandwidth and the block size used for queue management. To this end we examined a range of block sizes ranging from 1KB to 64MB in multiples of 4, using the above bandwidth range. For almost all code/input class/bandwidth combinations we found that the execution times formed a bowl shape, an example of which is shown in Figure 11. This Figure shows `LU` running on input size `W`. For each I/O bandwidth we can see high execution times for the largest and the smallest batch size, with faster runs for intermediate batch sizes. Large batch sizes have high overhead because of the increased probability that an `end_chkpt` call for some array will occur in the middle of a long wait for a large block of another array. Small batch sizes are a problem because of the increased overhead associated with synchronizing on and manipulating a large number of memory regions. While this work does not address the problem of picking an optimal batch size for a given code/input class/bandwidth combination, we did find that in general the bottom of the bowl stays flat, with a wide variety of batch sizes offering similarly good performance. This suggests that near-optimal batch sizes can be found for a wide variety of combinations. In our experiments, 64KB batches provided near-optimal performance in all configurations examined.

We evaluated the performance of asynchronous checkpointing by picking the best batch size for each code/input size/bandwidth combination and compared its performance to that of blocking incremental checkpointing (the `CHKPT_INCR` configuration from above). The result, for input size `W`, is shown in Figure 4.3. For each application it plots:

$$\frac{\text{(execution time w/ asynchronous checkpointing) - (execution time w/ blocking checkpointing)}}{\text{(execution time w/ blocking checkpointing)}}$$

It can be seen that different applications respond very differently to the two algorithms. Whereas `CG` performs better with asynchronous checkpointing, `FT` and `IS`

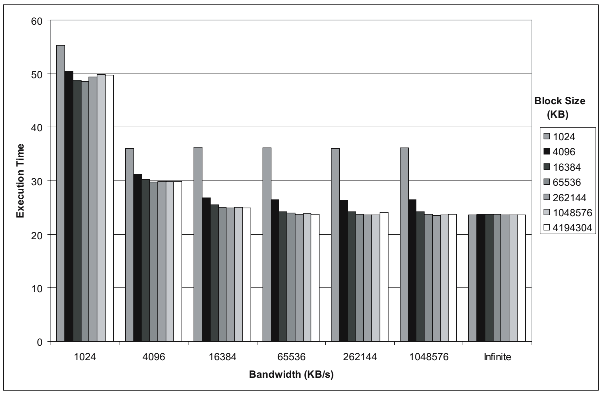


Fig. 11. Execution times for different bandwidths and batch sizes (LU-W)

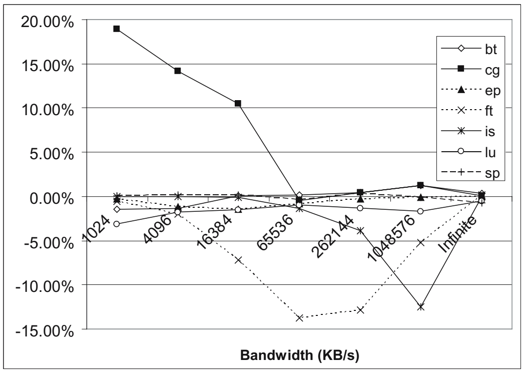


Fig. 12. Relative execution times for asynchronous vs blocking checkpointing (Class W)

tend to perform better with blocking checkpointing. Input class `A` shows very similar results, while class `S` exhibits a stronger preference for blocking checkpointing.

While intuitively it may appear that asynchronous checkpointing should always perform better than blocking checkpointing, these experiments show that there are some interference effects that complicate this simple analysis. While the full reason for this effect is still under investigation, it appears that the additional synchronization required to implement asynchronous checkpointing may have a negative performance impact. In particular, each call by the application thread to `start_chkpt` and `end_chkpt` requires synchronization with the asynchronous checkpointing thread.

5 Summary

We have presented a novel compiler analysis for optimizing automated checkpointing. Given an application that has been augmented by the user with calls to a `checkpoint` function, the analysis identifies regions in the code that do not have any writes to a given array. This information can be used to reduce the amount of data checkpointed and to asynchronously checkpoint this data in a separate thread. In our experiments with the NAS Parallel Benchmarks we have found that this analysis reduces checkpoint sizes by between 15% and 78% for most of the codes. These checkpoint size reductions were found to have a notable effect on checkpointing performance. Furthermore, we evaluated the performance of compiler-enabled asynchronous checkpointing. Although our experiments showed that asynchronous checkpointing can sometimes be better than blocking checkpointing, we discovered that this is frequently not the case. As such, the choice between asynchronous and blocking checkpointing depends on the application itself.

References

1. `http://phase.hpcc.jp/Omni/benchmarks/NPB`
2. `http://www.nas.nasa.gov/Software/NPB`
3. Adiga, N.R., Almasi, G., Almasi, G.S., Aridor, Y., Barik, R., Beece, D., Bellofatto, R., Bhanot, G., Bickford, R., Blumrich, M., Bright, A.A., Brunleroto J.: An overview of the bluegene/l supercomputer. In: IEEE/ACM Supercomputing Conference (2002)
4. Agarwal, S., Garg, R., Gupta, M.S., Moreira, J.: Adaptive incremental checkpointing for massively parallel systems. In: Proceedings of the 18th International Conference on Supercomputing (ICS), pp. 277–286 (2004)
5. Gioiosa, R., Sancho, J.C., Jiang, S., Petrini, F.: Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In: Supercomputing (November 2005)
6. Michalak, S.E., Harris, K.W., Hengartner, N.W., Takala, B.E., Wender, S.A.: Predicting the number of fatal soft errors in los alamos national laboratorys asc q supercomputer. IEEE Transactions on Device and Materials Reliability 5(3), 329–335 (2005)
7. Plank, J.S., Beck, M., Kingsley, G.: Compiler-assisted memory exclusion for fast checkpointing. IEEE Technical Committee on Operating Systems and Application Environments 7(4), 10–14 (Winter 1995)
8. Quinlan, D.: Rose: Compiler support for object-oriented frameworks. Parallel Processing Letters 10(2-3), 215–226 (2000)
9. Ross, K.C.R., Moreirra, J., Preiffer, W.: Parallel i/o on the ibm blue gene /l system. Technical report, BlueGene Consortium (2005)
10. Sancho, J.C., Petrini, F., Johnson, G., Fernandez, J., Frachtenberg, E.: On the feasibility of incremental checkpointing for scientific computing. In: 18th International Parallel and Distributed Processing Symposium (IPDPS), p. 58 (2004)
11. Schroeder, B., Gibson, G.A.: A large-scale study of failures in high-performance computing systems. In: Proceedings of the International Conference on Dependable Systems and Networks (DSN) (June 2006)
12. Zhang, K., Pande, S.: Efficient application migration under compiler guidance. In: Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems, pp. 10–20 (2005)

Techniques for Efficient Software Checking[⋆]

Jing Yu, María Jesús Garzarán, and Marc Snir

University of Illinois at Urbana-Champaign
{jingyu,garzaran,snir}@cs.uiuc.edu

Abstract. Dramatic increases in the number of transistors that can be integrated on a chip make processors more susceptible to radiation-induced transient errors. For commodity chips which are cost- and energy-constrained, we need a flexible and inexpensive technology for fault detection. Software approaches can play a major role for this sector of the market because they need little hardware modifications and can be tailored to fit different requirements of reliability and performance. However, software approaches add a significant overhead.

In this paper we propose two novel techniques that reduce the overhead of software error checking approaches. The first technique uses boolean logic to identify code patterns that correspond to outcome tolerant branches. We develop a compiler algorithm that finds those patterns and removes the unnecessary replicas. In the second technique we evaluate the performance benefit obtained by removing address checks before load and stores. In addition, we evaluate the overheads that can be removed when the register file is protected in hardware.

Our experimental results show that the first technique improves performance by an average 7% for three of the SPEC benchmarks. The second technique can reduce overhead by up-to 50% when the most aggressive optimization is applied.

1 Introduction

Dramatic increases in the number of transistors that can be integrated on a chip will deliver great performance gains. However, it will also expose a major roadblock, namely the poor reliability of the hardware. Indeed, in the near-future environment of low power, low voltage, relatively high frequency, and very small feature size, processors will be more susceptible to transient errors. Transient faults, also known as soft errors are due to impacts from high-energy particles that change the logic values of latches or logic structures [1,2,3,4].

In this new environment, we believe that a Software Checking System has a fundamental role in providing fault detection and recovery. It is possible that high-end architectures will include several hardware-intensive fault-tolerant techniques that are currently supported by IBM mainframes [5], HP NonStop [6] or mission-critical computers [7]. However, commodity multicore chips will likely be

⋆ This material is based upon work supported by the National Science Foundation under the CSR-AES program Award No. 0615273.

too cost- and energy-constrained to include such hardware. Instead, we believe that they will likely include only relatively simple hardware primitives, such as parity for certain processor buses and structures, error correction codes (ECC) and scrubbing in the memory hierarchy [8] and low-cost support for memory checkpointing and rollback (e.g., ReVive [9] or SafetyNet [10]). Then they will rely on flexible and inexpensive software technology for error protection.

Current software approaches address the problem by replicating the instructions and adding checking instructions to compare the results, but they add a significant overhead. In this paper we propose two novel techniques to reduce the overhead of the software error checking approaches. The first technique is based on the fact that programs already have redundancy, and if the compiler can determine the programs sections where such redundancy exists, it can avoid the replication and later checking. We use boolean logic to identify a code pattern that corresponds to outcome tolerant branches and develop a compiler algorithm that automatically finds those patterns and removes the unnecessary replicas. The second technique is based on the observation that faults that corrupt the application tend to quickly generate other noisy errors such as segmentation faults [11]. Thus, we can reduce replication of the instructions that tend to generate these type of errors, trading reliability for performance. In this paper we remove the checks of the memory addresses and discuss situations where removing these checks affect little to the fault coverage. This occurs when a check of a variable is covered by a later check to the same variable, and thus errors in the first check will be detected by the later checks; and in pointer-chasing, when the data loaded by a load is used immediately by another load. Finally, We also consider the situation where the register file is protected with parity or ECC, such as Intel Itanium [12], Sun UltraSPARC [13] and IBM Power4-6 [14]. We call them register safe platforms.

We have implemented the baseline replication and the proposed techniques using the LLVM Compiler Infrastructure [15] and run experiments on a Pentium 4 using Spec benchmarks. Our results show that the boolean logic technique achieves 7% performance speedup on three benchmarks, and 1.6% on average. If we do not check load addresses, the performance is improved by 20.2%. If we do not check addresses of both load and store, the performance is improved by 24.8%. On platforms where registers are protected in hardware, we can combine these techniques and obtain an average speedup of 35.2% and 40.8%, respectively, and decrease the software checking overhead by 44.9% and 50%, respectively. Our fault injection experiments show that removing address checks before loads only increases Silent Data Corruption (SDC) from 0.27% to 0.35%, and removing address checks for loads and stores raises SDC to 1.11%.

The rest of the paper is organized as follows. Section 2 presents the background and the baseline software checking; Section 3 describes the techniques to detect outcome tolerant branches; Section 4 describes the removal of address checks; Section 5 discusses the benefits of having a register file that is checked in hardware; Section 6 presents our experimental results; Section 7 presents related work, and finally Section 8 concludes the paper.

2 Background and Baseline Software Checking

The use of software approaches for fault tolerance has received significant attention in the research domain. Software techniques such as SWIFT [16] replicate the instructions of the original program and interleave the original instructions and their replicas in the same thread. Memory does not need to be replicated because the memory hierarchy is protected with ECC and scrubbing. Stores, branches, function calls and returns are considered "synchronization" points and checking instructions are inserted before these instructions to validate certain values. Before a store, checking instructions verify that the correct data is stored to the correct memory location. Before a branch, checking instructions verify that the branch takes the appropriate path. Before a function call checking instructions verify the input operands by comparing them against their replica. Before a function return, checking instructions verify the return value by comparing the return register and its replica.

Stores are executed only once, but loads are replicated because the loaded data can be corrupted. However, uncachable loads, such as those from external devices, and loads in a multithreaded program may return different values when executing two consecutive loads to the same memory address; so rather than replicating the load, checking instructions are also added before loads to verify that the address of the load matches its replica. After that verification, the loaded value can be copied to another register [16,17,18]. Thus, since loads are not replicated, they are also considered "synchronization" points. An example with the original and its corresponding replicated code is shown in Figure 1-(a) and (b), respectively. The replicated code contains additional instructions and uses additional registers marked with a `'`. The additional instructions are shown in bold and numbered. Instructions 1 and 2 check that the `load` is loading from the correct address, instruction 3 copies the value in `r3` to `r3'`, instruction 4 replicates the addition, and instruction 5-8 check that the store writes the correct data to the correct memory address.

```
                   cmp r6, r6'      (1)
                   jne faultDet     (2)
ld r3=[r6]         ld r3=[r6]                 ld r3=[r6]
                   mov r3'=r3       (3)
....               ....                       ....
add r4= r3,1       add r4= r3,1               add r4= r3,1
                   add r4'=r3',1    (4)       add r4'=r3,1    (4)
                   ....                       ....
                   cmp r4, r4'      (5)       cmp r4, r4'     (5)
                   jne faultDet     (6)       jne faultDet    (6)
                   cmp r6, r6'      (7)
                   jne faultDet     (8)
store [r6]=r4      store [r6]=r4              store [r6]=r4

(a) Original code  (b) Replicated code        (c) Safe registers
```

Fig. 1. Example of baseline software replication and checking

3 Use of Boolean Logic to Find Outcome Tolerant Branches

In this Section we explain how to use boolean logic to reduce the amount of replicated instructions. We first do an overview (Section 3.1) and then explain the compiler algorithm (Section 3.2).

3.1 Overview

Our technique is based on the fact that programs have redundancy. For instance, Wang et al. [19] performed fault injection experiments and found that about 40% of all the dynamic conditional branches are outcome tolerant. These are branches that, despite an error, converge to the correct point of execution. These branches are outcome-tolerant due to redundancies introduced by the compiler or the programmer. An example of outcome-tolerant branch appears in a structure such as `if (A || B || C) then X else Y`. In this case if `A` is erroneously computed to be true, but `B` or `C` are actually true, this branch is outcome tolerant, since the code converges to the correct path. The control flow graph of this structure is shown in Figure 2-(a).

The state-of-the-art approach to check for errors is to replicate branches as shown in Figure 2-(b), where the circles correspond to the branch replicas. However, we can reduce overheads by removing the comparison replica when the branch correctly branches to `X`. If the original comparison in `A` is true we need to execute the comparison replica to verify that the code correctly branches to `X`. However, if `A` is false, we can skip the execution of the `A` replica and move to check `B`. We will only need to execute the A replica if both `B` and `C` are also false. The resulting control flow graph is shown in Figure 2-(c). In situations where `A` and `B` are false, but `C` is true, we can save a few comparisons.

Outcome tolerant branches also appear in code structures such as `if (A & B & C) then X else Y`, and in general in all the code structures that contain one or more shortcut paths in the control flow graph. A basic *shortcut path* is `edge(A->X)` in Figure 3-(a), where both `A` and its child point to the same block. However, most shortcut paths are more complex. For instance, in Figure 3-(b), block `A` points to the same block pointed by its grandchild (not its direct

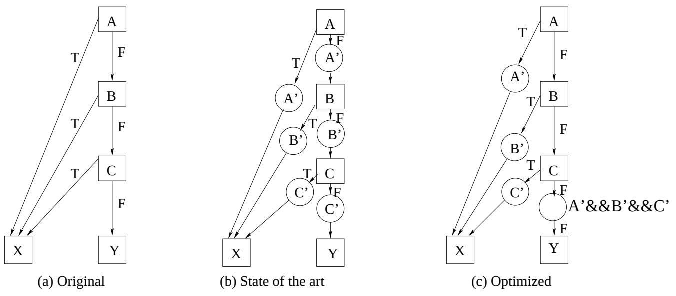


Fig. 2. Eliminating replicated predicate evaluation

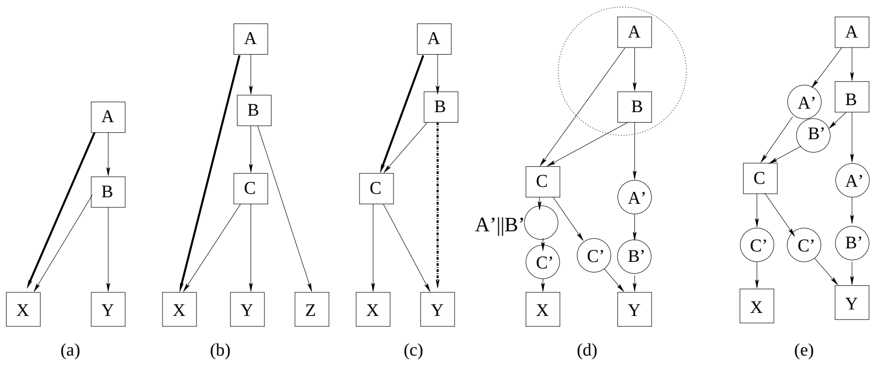


Fig. 3. Shortcut graphs and optimizations

child). Thus, the optimizer should move `A'` from `edge(A->B)` to `edge(B->Z)` and `edge(C->Y)`. The example in Figure 3-(c) can be optimized in two different ways. If `A` and `B` are considered as a whole unit, `edge(B->Y)` is the shortcut path, and the graph can be optimized as shown in Figure 3-(d); otherwise, it can be optimized as shown in Figure 3-(e).

Detecting the existence of a shortcut path is not sufficient to determine that there is an outcome tolerant branch. The reason is that one of the blocks involved in the shortcut can modify a variable that is later used by instructions outside the block. That block needs to be replicated or the error could propagate outside the block. Next we show two examples:

```
(a) if (*m > 0) && (m < N) then X else Y
(b) if (t=(*m > 0)) && (m < N) then X else Y
```

In the example in (a), if `(*m>0)` is mistakenly computed as True, but `(m<N)` is False, we can safely ignore the error on `(*m>0)` and take the `Y` path. However, if the error occurs to the example in (b), and `t` is used in `Y`, ignoring the error will result in a wrong value for `t` being propagated to `Y`, which may end up corrupting the system. To avoid this type of errors our compiler algorithm only considers blocks that are involved in a shortcut path and produce values that are only used by the block itself.

3.2 Compiler Algorithm

Our algorithm analyzes the control flow graph of the original program and extracts the shortcut paths and the related blocks. A *shortcut graph* always has a head node (block `A` in all the examples in Figure 3), one or more intermediate nodes (like `B` and `C`), two or more leaves (like `X` and `Y`), and one or more shortcut paths. Notice that in this paper we call a *block* to a single basic block or a list of basic blocks connected one by one with edges of unconditional branches.

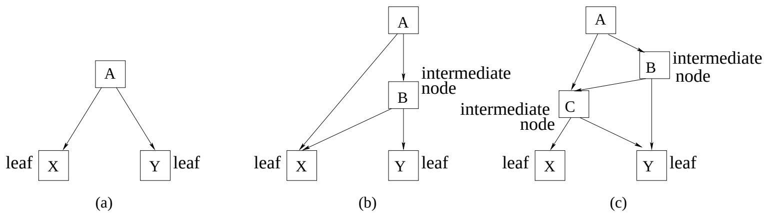


Fig. 4. Constructing potential shortcut graphs

Our algorithm has two phases: first a search of all potential shortcut graphs, and second, the optimization and appropriate placement of the replicas.

Shortcut Graphs Search. The searching process starts by classifying each block as an intermediate node or a leaf, and building an intermediate node set and a leaf set. A block is called "intermediate node" if it ends with a conditional branch and does not contain side effects (does not contain a function call, a memory write or generates a value used by another block). In addition, to avoid being trapped in loops, we require that none of the outgoing edges of an intermediate node is a loop backward edge. If the node does not classify as intermediate node, then it is considered a "leaf", meaning that this block can be at the most an ending node in a shortcut graph. At the same time we build the intermediate and leaf sets, we also build a separate head node set. A block is called "head node" if it ends with a conditional branch and none of the outgoing edges is backwards, no matter it has side effects or not. Thus the head node set contains all intermediate nodes and some of the leaves.

After building the intermediate node set, the leaf set, and the head node set the shortcut graphs are built from bottom to up by scanning the head node set repeatedly. We start by initializing an empty set "graph-head-set", which will contain temporary graph head nodes. For any node(`A`) in the head node set, we check its two children (see Figure 4):

1. If the two children are leaves, this node is added to the graph-head-set (Figure 4-(a)).
2. If one child is a leaf(`X`) and the other child is an intermediate node(`B`) and node(`B`) is already in the graph-head-set, node(`B`) is replaced by the current node(`A`) in the graph-head-set (Figure 4-(b)). We also check if the leaf(`X`) is a child or grandchild of node(`B`), in which case a shortcut path for node (`A`) is marked.
3. If the two children are both intermediate nodes((`B`) and (`C`)) and both are in the graph-head-set, nodes (`B`) and (`C`) are replaced by node(`A`) in the graph-head-set (Figure 4-(c)). We also check if (`A`) introduces new shortcut paths.

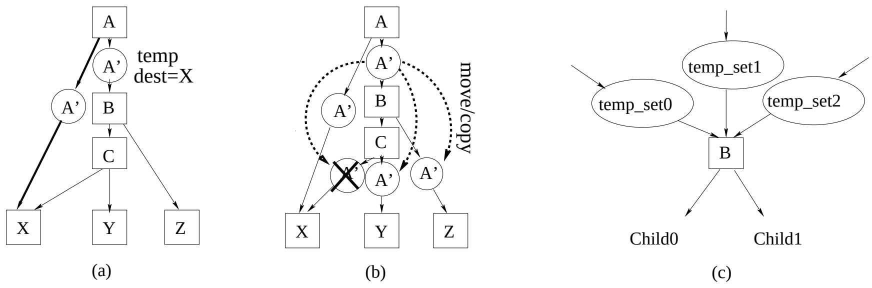


Fig. 5. Optimizing shortcut graphs

The scan continues until all the nodes in the head node set have been visited. Then, a node in the graph-head-set represents a graph led by this node together with the shortcut paths found. A final pass traverses the graph-head-set and removes those heads that do not contain any shortcut path.

Optimization. After the shortcut paths are found we start applying the optimization, but we first check when it is legal to perform it. In Figure 2-(b), our optimization will move the replica `A'` from `edge(A->B)` to `edge(C->Y)`. However, this is only legal if `A` dominates `C`. Otherwise `A'` may use undefined values in the new position. Thus to apply our optimization phase we first verify the domination relationship of all shortcut paths.

The goal of our optimization pass is to move replicas of the non-shortcut path down to the edge/s between the last child and the leaf/leaves. Next, we explain how this algorithm proceeds using the example in Figure 5. For each shortcut graph in the graph-head-set the algorithm finds all the shortcut paths (`edge(A->X)` in Figure 5-(a)), marks the replica (`A'`) on the other path as temporary (temp), and records the destination of the shortcut path (`X`). Next the optimization pass scans all the intermediate nodes in the shortcut graph in a top-down fashion, and moves temporary replicas from the incoming edges to all the outgoing ones, except to those where the recorded destination of the replica and the destination of the intermediate node that we are processing are the same (an example is shown in Figure 5-(b)). Notice that when an intermediate node has multiple incoming edges (as shown in Figure 5-(c)) we only move the replicas that appear on all the incoming edges. Also notice that this optimization pass processes nodes top-down, and it does not treat multiple nodes as a single unit. Thus, for the example in Figure 3-(c), the optimized version after this pass will be the one shown in Figure 3-(e).

Finally note that `A`, `B` and `C` can contain computations like `(s+1) == 5`. In this case, if the computations are only used to determine the outcome of the branch, the computation replicas are also eliminated when the branch replica does not need to execute.

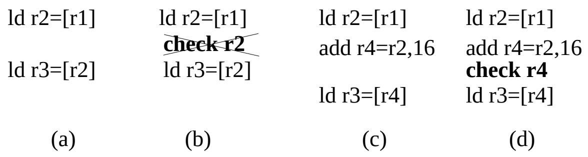
4 Removal of Address Checks

Recent experiments have shown that faults produce not only data corruption, but also events that are atypical of steady state operation and that can be used as a warning that something is wrong [11]. Thus, we can reduce the overhead of the software approaches and trade reliability for performance by reducing the replication, hoping that the error will manifest with these atypical events.

In this Section we consider the removal of address checks before load and store instructions. Errors in the registers containing memory addresses may manifest as segmentation faults. However, any fault-tolerant system must also include support for roll-back to a safe state and thus, on a segmentation fault we can roll-back and re-execute, and only communicate the error to the user if it appears again. However, by doing this the system will be vulnerable to errors, since some of these faulty addresses will access a legal space and the operating system will not be able to detect the error. Thus, this technique will decrease error coverage. Next, we discuss two techniques that the compiler can use to determine which load and store instructions are most suitable for address check removal.

Address checks can be removed when there are later checks checking the same variable. For example, in Figure 1-(b), checking instructions (1-2) and (7-8) are checking the register `r6`. This makes the first check (1-2) unnecessary, because if an error occurs to `r6` it will manifest as a segmentation fault or will be eventually detected by the checking instructions (7-8). We have observed many of these checks in the SPEC benchmarks due to the register indirect addressing mode, since the same register is used to access two fields of a structure, or because two array accesses share a common index. Removing these replicated checks can significantly reduce the software overhead.

Address checks can also be removed when the probability of error to the loaded value is small. This case appears in pointer chasing, where the data loaded from memory is used as the address for a subsequent load. An example is shown in Figure 6-(a) and (b). In this case, since the processor will issue the second load as soon as the first one completes, the probability of error is very small. In some cases, however, the value loaded by the first load is not exactly the one used by the next load, if not that it may be first modified by an `add` instruction. This occurs when accessing an element of a structure that is different from the first one. In this case, the probability of error is higher, and the checking instructions will also determine if an error occurred during the computation of the addition. An example is shown in Figure 6-(c) and (d).



(a)
```
ld r2=[r1]

ld r3=[r2]
```

(b)
```
ld r2=[r1]
~~check r2~~
ld r3=[r2]
```

(c)
```
ld r2=[r1]
add r4=r2,16

ld r3=[r4]
```

(d)
```
ld r2=[r1]
add r4=r2,16
check r4
ld r3=[r4]
```

Fig. 6. Address check removal for pointer chasing

In this paper we evaluate the removal of the address checks for only the loads, or for both loads and stores. Thus, our results are an upper bound on the performance benefit that we can obtain and the reliability that we can lose. In the future we plan to write a data flow analysis to identify the checks that are safe to remove, as explained above.

5 Register Safe Platforms

In this Section we consider the situation where the register file is hardware protected with parity or ECC, or other cost-effective mechanisms as the ones proposed by [20,21,22,23]. In fact, the register file of the Intel Itanium [12], Sun UltraSPARC [13] and IBM Power4-6 [14] are already protected by parity or ECC. However, the ALUs and other portions of the processor are not protected, so arithmetic and logic operations can return wrong results. Thus, all the instructions that imply ALU operations need to be replicated; however, memory operations such as load and stores are safe. As a result, a register that is defined by a load does not need to be replicated, saving the instruction to perform the copy and the additional register. An example is shown in Figure 1. The replicated code in Figure 1-(b) can be simplified as shown in Figure 1-(c). Register `r3'` is not necessary because registers and memory are safe, and instruction 4 can use directly the contents from register `r3`. Instructions 1, 2, 7 and 8 can be removed if we assume register `r6` has been defined by a load. Instructions 5 and 6 cannot be removed because register `r4` is defined by an addition, and we need to validate the results of the addition.

6 Evaluation

In this Section we evaluate our proposed techniques. We first discuss our environmental setup (Section 6.1), analyze our techniques statically (Section 6.2), evaluate performance (Section 6.3), and measure reliability (Section 6.4).

6.1 Environmental Setup

We use LLVM [15] as our compiler infrastructure to generate redundant codes. Replicated and checking instructions are added at the intermediate level, right after all the static optimizations have been done. We replicate all the integer and floating point instructions. Previous implementations have replicated instructions at the backend, right before register allocation [16,24] or via dynamic binary translation [25]. However, the advantages of working at the intermediate level are: i) the redundant code can be easily ported to other platforms, ii) we do not need to fully understand the assembly code for that platform, and iii) at the intermediate level we see a simple memory access model rather than complex one of the x86 ISA. To prevent optimizations done by the backend generator such as common subexpression elimination and instruction combination, we tag the replicated instructions, and the backend optimizations are applied separately to the tag and the untag instructions.

For the evaluation we use SPEC CINT2000 and the C codes from SPEC CFP 2000, running with the ref inputs. Experiments are done on a 3.6GHz INTEL Pentium 4 with 2GB of RAM running RedHat9 Linux.

6.2 Static Analysis

In this Section we characterize load addresses depending on whether the register is checked by a later checking instruction (Covered), or if the register used by the load was just loaded from memory (Loaded), as in the pointer chasing example of Section 4. All the remaining load addresses are classified as (Other). The breakdown is shown in Figure 7. On average more than 40% load addresses have nearby later checks on the same value. About 20% of the loads use registers whose contents where just loaded from memory. As we have discussed in Section 4, the probability of error of any of these addresses is very small, because the processor will likely issue the second load as soon as the first one completes. Also, if we assume a register safe platform these checks are unnecessary. For the remaining 40% of the addresses, an error in the most significant bits will be detected as a form of segmentation faults, but an error in the least significant ones can cause a silent error.

6.3 Performance

Figure 8 shows the performance speedup obtained when using boolean logic to eliminate replication and checks on outcome tolerant branches (Section 3). Three benchmarks (gzip, vpr, and perlbmk) achieve 7% performance gains, though the average speedup is 1.6% through all tested benchmarks. Notice that there is also a negative impact on vortex, where we observe more load/store instructions after the optimization, meaning that this optimization introduces additional register spills that hurt the benefit of less dynamic instructions.

Figure 9 evaluates the performance benefit of our second technique (Section 4): baseline Fully Replicated(FullRep), No checks for Address of Loads(NAL), No checks for Address of Load and Store(NALS), and No checks when the Register file is safe (R). The Fully Replicated code(FullRep) is on average 2.38 times

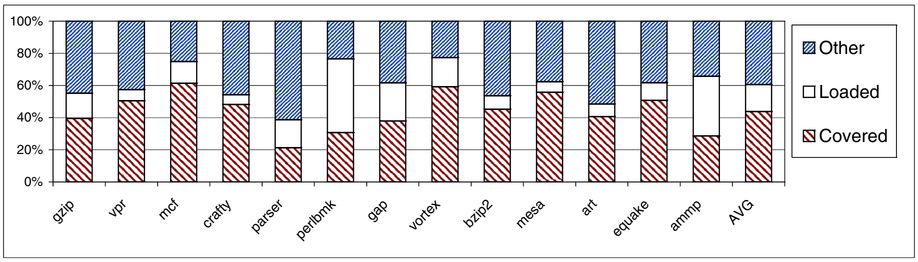


Fig. 7. Characterization of load addresses

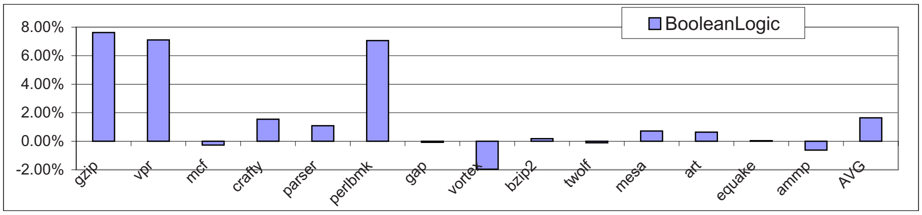


Fig. 8. Performance speedup with boolean logic optimization compared to baseline replication

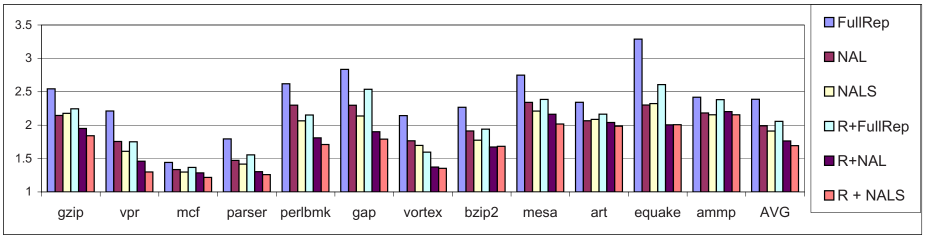


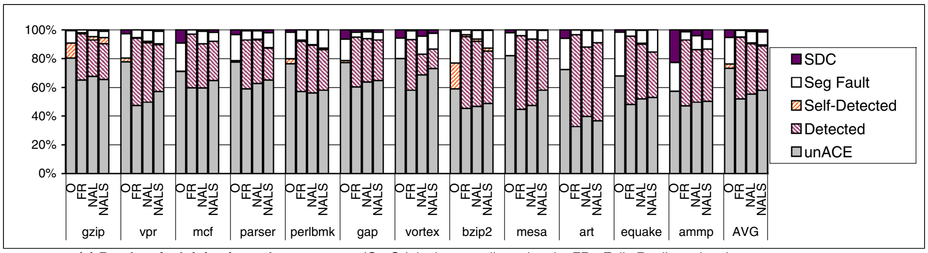
Fig. 9. Performance of the different optimizations normalized against the original non-replicated code

slower than the original code. This large overhead is due to high register pressure and additional instructions. On average, register safe optimization (R) runs 16.0% faster than the (FullRep).

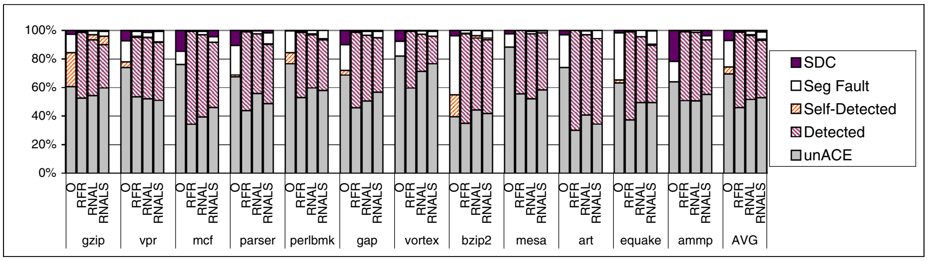
After we remove checks for address of loads (NAL), we get an average 20.2% speedup over the baseline Fully Prelicated (FullRep). If we further remove checks for address of stores (NALS), we improve 4.6% more. And if the register is protected in hardware and we combine (NAL) or (NALS) with (R), we can obtain an average speedup of 35.2% and 40.8% respectively, what will reduce the the software checking overhead by 44.9% and 50%, respectively. Notice that with (NALS) all address checks before loads and stores are removed, so the performance benefit of (R+NALS) versus (NALS) is due to the reduced register pressure (the register of the load does not need to be replicated) and the removal of a few additional checks before the data being stored.

6.4 Reliability

Our first technique is very conservative and should not affect the fault coverage. But for the second technique, since we remove all the checks for memory addresses, memory can be corrupted. In order to evaluate the loss of fault coverage, we use Pin [26] and inject faults to the binary file (excluding system libraries). We assume a Single Event Upset(SEU) fault model, so only one bit fault is injected during the execution of the program. In total 300 faults are injected for each program. Although both integer and floating point registers



(a) Random fault injection scheme (O - Original non-replicated code, FR - Fully Replicated code, NAL - No address check for load, NALS - No address check for load, store)



(b) Safe register fault injection scheme (O - Original non-replicated code, RFR - Fully Replicated code with Register Safe OPT, RNAL - No address check for load with Register Safe OPT, RNALS - No address check for load, store with Resiter Safe OPT)

Fig. 10. Fault-detection rates break down

can be corrupted, in order to magnify the impact of the errors we only inject fault to the 8 32-bit integer registers and the status flags EFLAGS. When we consider that the register file is not protected in hardware we mimic the fault distribution by randomly selecting a dynamic instruction and flipping a random bit in a random register (we call this scheme "random fault injection"). When the register file is protected in hardware, we do the same, but flip the random bit from its "output". The output can be in a register or in memory if it has been spilled. In this scheme, memory load instructions are avoided (we call this scheme "safe register fault injection").

After injecting an error into the binary, the program is run to completion (unless it aborts) and its output is compared to a correct output. Depending on the result the error will be categorized as: (unACE), the bit is unnecessary for Architectural Correct Execution [27]; (Detected), the error is detected by our checking code; (Self-Detected), the error is detected by the program assertions; (Seg Fault), the error manifests as an exception or a segmentation fault; (SDC), Silent Data Corruption, when the program finishes normally but the produced output is incorrect. (SDC) is the first type of errors we want to prevent. Then, we also want to minimize (self-Detected) errors and (Seg Fault), because it is usually hard to determine if the error is due to a program bug or a soft error. But with proper support, if we can roll-back and re-execute, these faults can be recovered, so they are less harmful.

Figure 10-(a) and (b) show the experimental results for random fault injection and safe register fault injection, respectively. The fault detection rates for these two schemes are very similar. Notice that the original program (O) has on average 75% (unACE) and less than 10% (SDC), which means that the software itself has a certain fault maskability. With the safe register scheme more faults result in SDC than with the random scheme (8.5% over 5%) and less Seg Fault (15.6% over 17%). The reason is that the random scheme is more likely to pick up a dynamic dead register or a register that holds the index for addresses.

After the program is replicated (FR), most (Seg Fault), (Self-Detected) and (SDC) go to the (Detected) category. (SDC) errors appear because some faults are injected before the value is used but after is checked. If we remove checks for addresses, reliability does not drop much. Under random injection scheme, if we remove checks for load addresses (NAL), comparing to (FR), (SDC) increases from 0.36% to 1.08%, (Seg Fault) increases from 4.47% to 8.05%. If we also remove checks for store addresses (NALS), (SDC) rises to 1.44%, and (Seg Fault) rises to 9.02%. Under safe register injection scheme, removing checks for load addresses increases (SDC) from 0.27% to 0.38%, increase (Seg Fault) from 2.66% to 4.99%. Removing checks for store addresses further results in (SDC) of 1.11%, and (Seg Fault) of 4.99%. In other words, when normalized to the original program, under the safe register scheme removing checks for addresses of load only incurs an extra 1.3% (SDC), while removing checks for all addresses incurs 9.8%(SDC). Given that we almost decrease the performance overhead by half, this loss of fault coverage seems acceptable.

7 Related Work

Previous work on compiler instrumentation for fault tolerance focuses on replication and checking. There have been previous works on software checking optimization. For example, SWIFT [16] merges checks before branches into control flow signature checks, and removes checks for blocks that do not have stores. In this paper, we propose a new area for optimization: when the code structure itself can mask errors and the compiler can determine those programs sections, replication and later checking can be avoided.

Some previous works provide ways to trade reliability for performance. For example, the work by Oh and McCluskey [28] selectively duplicates procedure calls instead of replicating instructions inside them. This way error detection latency is sacrificed for less power consumption. But for each procedure, either all the instructions in the procedure or the call needs to be replicated. PROFiT [29] and Spot [25] divide program into regions and pick up only important regions to do software replication and checking. Spot provides very flexible selection granularity, ranging from a few blocks to a whole procedure. However, in Spot making a good selection requires knowledge of fault mask probability and replication overhead for each region. Jonathan Chang et. al [20] propose to protect a portion of the register file based on a profile of register life time and usage. For different platforms or different programs, the protected portion may be different. In this paper, we provide a fine and simple leverage control: we choose to remove checks for addresses of load or stores. With static compiler analysis, this technique can be applied independently of the target platform. Furthermore, we can combine this technique with previous ones to trade fault coverage with performance.

Previous works on compiler instrumentation for fault-tolerance implement their techniques at the source level [30], compiler backend [16,24,31,29,32], or runtime binary level [25]. However, our techniques are implemented at the intermediate level, which makes it portable across platforms and friendly to users who are not expert on the target ISA.

8 Conclusion

This paper makes several contributions. First, we identify a code pattern that corresponds to outcome tolerant branches, and develop a compiler algorithm that finds these patterns, avoiding unnecessary replication and checking. Second, we evaluate the removal of address checks for loads and stores, and analyze situations where these checks can be removed with little loss of fault coverage. We also identify the check and replicated registers that can be removed on a register safe platform.

Optimizing outcome tolerant branches obtains 7% performance speedup for 3 benchmarks, and an average of 1.6% for all, while keeping the same level of reliability. We also find that on register safe platforms removing the checks for the addresses of load reduce the replication overhead by 44.9%, and only increases SDC (Silent Data Corruption) rate from 0.27% to 0.38%. Also, if 1.11% SDC rate is acceptable, we can furthermore reduce the replication overhead by 50% by also removing checks for the store addresses.

References

1. Constantinescu, C.: Impact of Deep Submicron Technology on Dependability of VLSI Circuits. In: Proc. of the International Conf. on Dependable Systems and Networks, pp. 205–209 (2002)
2. Hazucha, P., Karnik, T., Walstra, S., Bloechel, B., Tschanz, J.W., Maiz, J., Soumyanath, K., Dermer, G., Narendra, S., De, V., Borkar, S.: Measurements and Analysis of SER-tolerant Latch in a 90-nm dual-V/sub T/ CMOS Process. IEEE Journal of Solid-State Circuits 39(9), 1536–1543 (2004)
3. Karnik, T., Hazucha, P.: Characterization of Soft Errors Caused by Single Event Upsets in CMOS Processes. IEEE Transactions on Dependable and Secure Computing 1(2), 128–143 (2004)
4. Shivakumar, P., Kistler, M., Keckler, S., Burger, D., Alvisi, L.: Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. In: Proc. of the International Conf. on Dependable Systems and Networks, pp. 289–398 (2002)
5. Slegel, T., Averill, R., Check, M., Giamei, B., Krumm, B., Krygowski, C., Li, W., Liptay, J., MacDougall, J., McPherson, T., Navarro, J., Schwarz, E., Shum, K., Webb, C.: IBM's S/390 G5 Microprocessor Design. IEEE Micro 19(2), 12–23 (1999)
6. McEvoy, D.: The architecture of tandem's nonstop system. In: ACM 1981: Proceedings of the ACM 1981 conference, p. 245. ACM Press, New York (1981)

7. Yeh, Y.: Triple-triple Redundant 777 Primary Flight Computer. In: Proc. of the IEEE Aerospace Applications Conference, pp. 293–307 (1996)
8. Mukherjee, S., Emer, J., Fossum, T., Reinhardt, S.: Cache Scrubbing in Microprocessors: Myth or Necessity? In: Proc. of the Pacific RIM International Symposium on Dependable Computing, pp. 37–42 (2004)
9. Prvulovic, M., Zhang, Z., Torrellas, J.: ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In: Proc. of the International Symposium on Computer Architecture (ISCA) (2002)
10. Sorin, D., Martin, M., Hill, M., Wood, D.: SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In: Proc. of the International Symposium on Computer Architecture (ISCA) (2002)
11. Wang, N.J., Patel, S.J.: ReStore: Symptom Based Soft Error Detection in Microprocessors. In: Proc. of the International Conference on Dependable Systems and Network (DSN), pp. 30–39 (2005)
12. McNairy, C., Bhatia, R.: Montecito: A Dual-core, Dual-thread Itanium Processor. IEEE Micro 25(2), 10–20 (2005)
13. Kongetira, P., Aingaran, K., Olukotun, K.: Niagara: A 32-way multithreaded sparc processor. IEEE Micro 25(2), 21–29 (2005)
14. Bossen, D., Tendler, J., Reick, K.: Power4 system design for high reliability. IEEE Micro 22(2), 16–24 (2002)
15. Lattner, C., Adve, V.: The LLVM Compiler Framework and Infrastructure Tutorial. In: Eigenmann, R., Li, Z., Midkiff, S.P. (eds.) LCPC 2004. LNCS, vol. 3602. Springer, Heidelberg (2005)
16. Reis, G.A., Chang, J., Vachharajani, N., Rangan, R., August, D.I.: SWIFT: Software Implemented Fault Tolerance. In: Proc. of the International Symposium on Code Generation and Optimization (CGO) (2005)
17. Mukherjee, S.S., Kontz, M., Reinhardt, S.K.: Detailed Design and Evaluation of Redundant Multithreading Alternatives. In: Proc. of International Symposium on Computer Architecture, Washington, DC, USA, pp. 99–110. IEEE Computer Society, Los Alamitos (2002)
18. Reinhardt, S.K., Mukherjee, S.S.: Transient Fault Detection via Simultaneous Multithreading. In: Proc. of International Symposium on Computer Architecture, pp. 25–36. ACM Press, New York (2000)
19. Wang, N., Fertig, M., Patel, S.: Y-Branches: When You Come to a Fork in the Road, Take It. In: Proc. of the International Conference on Parallel Architectures and Compilation Techniques (PACT) (2003)
20. Chang, J., Reis, G.A., Vachharajani, N., Rangan, R., August, D.: Non-uniform fault tolerance. In: Proceedings of the 2nd Workshop on Architectural Reliability (WAR) (2006)
21. Gaisler, J.: Evaluation of a 32-bit microprocessor with built-in concurrent error detection. In: FTCS 1997: Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS 1997), Washington, DC, USA, p. 42. IEEE Computer Society, Los Alamitos (1997)
22. Montesinos, P., Liu, W., Torrellas, J.: Shield: Cost-Effective Soft-Error Protection for Register Files. In: Third IBM TJ Watson Conference on Interaction between Architecture, Circuits and Compilers (PAC 2006) (2006)
23. Hu, J., Wang, S., Ziavras, S.G.: In-register duplication: Exploiting narrow-width value for improving register file reliability. In: DSN 2006: Proceedings of the International Conference on Dependable Systems and Networks (DSN 2006), Washington, DC, USA, pp. 281–290. IEEE Computer Society, Los Alamitos (2006)

24. Reis, G.A., Chang, J., Vachharajani, N., Rangan, R., August, D.I., Mukherjee, S.S.: Design and Evaluation of Hybrid Fault-Detection Systems. In: Proc. of the International International Symposium on Computer Architecture (ISCA) (2005)
25. Reis, G.A., Chang, J., August, D.I., Cohn, R., Mukherjee, S.S.: Configurable Transient Fault Detection via Dynamic Binary Translation. In: Proceedings of the 2nd Workshop on Architectural Reliability (WAR) (2006)
26. Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In: Proc. of the International Conference on Programming Language Design and Implementation (PLDI) (2005)
27. Mukherjee, S.S., Weaver, C., Emer, J., Reinhardt, S.K., Austin, T.: A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In: MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture, Washington, DC, USA, p. 29. IEEE Computer Society, Los Alamitos (2003)
28. Oh, N., McCluskey, E.J.: Low Energy Error Detection Technique Using Procedure Call Duplication. In: Proc. of the International Conference on Dependable Systems and Network (DSN) (2001)
29. Reis, G.A., Chang, J., Vachharajani, N., Rangan, R., August, D.I., Mukherjee, S.S.: Software-controlled fault tolerance. ACM Trans. Archit. Code Optim. 2(4), 366–396 (2005)
30. Rebaudengo, M., Reorda, M.S., Violante, M., Torchiano, M.: A Source-to-Source Compiler for Generating Dependable Software. In: IEEE International Workshop on Source Code Analysis and Manipulation (SCAM), pp. 35–44 (2001)
31. Oh, N., Shirvani, P., McCluskey, E.J.: Error Detection by Duplicated Instructions in Super-scalar Processors. IEEE Transactions on Reliability 51(1), 63–75 (2002)
32. Chang, J., Reis, G.A., August, D.I.: Automatic Instruction-Level Software-Only Recovery. In: DSN 2006: Proceedings of the International Conference on Dependable Systems and Networks (DSN 2006), Washington, DC, USA, pp. 83–92. IEEE Computer Society, Los Alamitos (2006)

Revisiting SIMD Programming

Anton Lokhmotov[1,⋆], Benedict R. Gaster[2],
Alan Mycroft[1], Neil Hickey[2], and David Stuttard[2]

[1] Computer Laboratory, University of Cambridge
15 JJ Thomson Avenue, Cambridge, CB3 0FD, UK
[2] ClearSpeed Technology
3110 Great Western Court, Bristol, BS34 8HP, UK

Abstract. Massively parallel SIMD array architectures are making their way into embedded processors. In these architectures, a number of identical processing elements having small private storage and using asynchronous I/O for accessing large shared memory executes the same instruction in lockstep.

In this paper, we outline a simple extension to the C language, called C^n, used for programming a commercial SIMD array architecture. The design of C^n is based on the concept of the SIMD array type architecture and revisits first principles of designing efficient and portable parallel programming languages. C^n has a low level of abstraction and can also be seen as an intermediate language in the compilation from higher level parallel languages to machine code.

1 Introduction

Massively parallel SIMD array architectures are no longer merely the province of large supercomputer systems but are making their way into embedded processors. What seemed a cautious extrapolation a decade ago ("even a parallel SIMD coprocessor embedded in a single-user workstation may not be such a far-fetched idea" [1]) has now become a reality.

In the 40 years since the design of Illiac IV [2], the first large-scale array computer consisting of 64 processing elements (PEs), the progress of VLSI technology allows to pack even more PEs into a single-chip microprocessor that in most respects can be considered "embedded". For example, ClearSpeed's CSX600 array processor consisting of 96 PEs has excellent performance per unit power by delivering more than 3 GFLOPS per watt [3].

The CSX is a SIMD array architecture with a control unit and a number of processing elements, with each PE having relatively small private storage and using asynchronous I/O mechanisms to access large shared memory.

In this paper, we describe a data-parallel extension to the C language, called C^n, for programming the CSX architecture. In C^n, parallelism is mainly expressed at the type level rather than at the code level. Essentially, C^n introduces a new multiplicity type qualifier **`poly`** which implies that each PE has its own copy of a value of that type. For example, the definition **`poly int`** `X;` implies that, on the CSX600 with 96 PEs, there

⋆ This author gratefully acknowledges the financial support by a TNK-BP Cambridge Kapitza Scholarship and by an Overseas Research Students Award.

exist 96 copies of integer variable `X`, each having the same address within its PE's local storage.

The multiplicity is also manifested in conditional statements. For example, the following code assigns zero to (a copy of) `X` on every even PE (the runtime function `get_penum()` returns the ordinal number of a PE):

```
if(get_penum()%2 == 0) X = 0;
```

On every odd PE, the assignment is not executed (this is equivalent to issuing a NOP instruction, as the SIMD array operates in lock-step).

We describe designing C^n as an efficient *and* portable language. Efficiency and portability often conflict with each other (especially in parallel languages [4]). We argue that the CSX architecture (§2) is representative of its class and thus can be considered as a *SIMD array type architecture* (§3). Core C^n operations (§4) can be thought of as cheap (in the spirit of C), while more expensive operations are relegated to the standard library. We compare and contrast C^n design decisions with similar approaches (§5), and then argue that C^n can be seen as an *intermediate* language in the compilation from higher level parallel languages (§6). We briefly discuss the C^n compiler implementation (§7) and outline future work in conclusion (§8).

2 CSX Architecture

This section outlines ClearSpeed's CSX architecture [3], which largely honours the classical SIMD array organisation pioneered by the Solomon/Illiac IV designs [5], albeit embodied in a single chip.

2.1 CSX Family

The CSX architecture is a family of processors based on ClearSpeed's multi-threaded array processor (MTAP) core. The architecture has been developed for high rate processing. CSX processors can be used as application accelerators, alongside general-purpose processors such as those from Intel and AMD.

The MTAP consists of execution units and a control unit. One part of the processor forms the *mono* execution unit, dedicated to processing scalar (or mono) data. Another part forms the *poly* execution unit, which processes parallel (or poly) data, and may consist of tens, hundreds or even thousands of identical processing element (PE) cores. This array of PE cores operates in a synchronous, Single Instruction Multiple Data (SIMD) manner, where every enabled PE core executes the same instruction on its local data.

The control unit fetches instructions from a *single* instruction stream, decodes and dispatches them to the execution units or I/O controllers. Instructions for the mono and poly execution units are handled similarly, except for conditional execution. The mono unit uses conditional jumps to branch around code like a standard RISC architecture. This affects both mono and poly operations. The poly unit uses an *enable register* to control execution of each PE. If one or more of the bits of that PE enable register is zero, then the PE core is *disabled* and most instructions it receives will be ignored. The enable register is a stack, and a new bit, specifying the result of a test, can be pushed onto the top of the stack allowing nested predicated execution. The bit can later be popped from the top of the stack to remove the effect of that condition. This makes handling nested conditions and loops efficient.

In order to provide fast access to the data being processed, each PE core has its own local memory and register file. Each PE core can directly access only its own storage. (Instructions for the poly execution unit having a mono register operand indirectly access the mono register file, as a mono value gets broadcast to each PE.) Data is transferred between PE (poly) memory and the poly register file via load/store instructions. The mono unit has direct access to main (mono) memory. It also uses load/store instructions to transfer data between mono memory and the mono register file. Programmed I/O (PIO) extends the load/store model: it is used for transfers of data between mono memory and poly memory.

2.2 CSX600 Processor

The CSX600 is the first product in the CSX family. The processor is optimised for intensive double-precision floating-point computations, providing sustained 33 GFLOPS of performance on DGEMM (double precision matrix multiply), while dissipating an average of 10 watts. The poly execution unit is a linear array of 96 PE cores, with 6KB SRAM and a superscalar 64-bit FPU on each PE core. The PE cores are able to communicate with each other via what is known as *swazzle path* that connects each PE with its left and right neighbours. Further details can be found in white papers [3].

2.3 Acceleration Example

A C^n implementation of Monte-Carlo simulations for computing European option pricing with double precision performs 100,000 Monte-Carlo simulations at the rate of 206.5M samples per second on a ClearSpeed Advance board having two CSX600 chips. In comparison, an optimised C program using the Intel Math Kernel Library (MKL) and compiled with the Intel C Compiler achieves the rate of 40.5M samples per second on a 2.33GHz dual-core Intel Xeon (Woodcrest, HP DL380 G5 system). Combining both the Intel processor and the ClearSpeed board achieves the rate of 240M samples per second, which is almost 6 times the performance of the host processor alone.

3 C^n Design Goals and Choices

The key design goals of C^n were efficiency and portability. We first discuss these goals in a broader context of programming languages for sequential and parallel computers (§3.1), and then discuss how they affected the design choices of C^n (§3.2).

3.1 Efficiency and Portability

Program efficiency and portability are two common tenets of high-level programming languages. Efficiency means that it is possible to write a compiler generating code that is “almost” as fast as code written in assembly. Portability means that software can be adapted to run on different target systems.

Languages like C have been successful largely because of their efficiency and portability on von Neumann machines, having a single processor and uniform random access memory. Efficiency comes from the programmer's clear understanding of the performance implications of algorithm selection and coding style. Portability is achieved because languages like C hide most features of physical machines, such as instruction set, addressing modes, register file, *etc.* Moreover, the hidden features apparently have a negligible effect on performance, so porting often maintains efficiency [4].

Efficiency and portability are even more desired when programming parallel systems. Performance is the most compelling argument for parallel computing; and given the amount of human effort required to develop an efficient parallel program, the resulting program should better have a long useful life [6].

In the world of parallelism, however, no single model accurately abstracts the variability of parallel systems. While it is possible, for example, to program distributed memory machines using a shared memory model, programmers having limited control over data distribution and communication are unlikely to write efficient programs. So in this world, sadly, efficiency and portability are no longer close friends.

Snyder introduced [7] the notion of a *type architecture*—a machine model abstracting the performance-important features of a family of physical machines, in the same way as the RAM model abstracts von Neumann machines. He proposed the Candidate Type Architecture (CTA) which effectively abstracts MIMD machines (multicomputers) with unspecified interconnect topology. The key CTA abstraction is that accessing another processor's memory is significantly more expensive than accessing local memory (typically by 2–5 orders of magnitude) [4].

3.2 C^n as a Language for the SIMD Array Type Architecture

Our key observation is that the CSX architecture can be considered a *SIMD array type architecture* (SATA), as it respects classical organisation and has typical costs of communicating data between main (mono) and local (poly) memory. Designing a core language based on the type-architecture facilities should provide both efficiency and portability [7]. To match the programmer's intuition, core language operations are cheap, while operations relegated to the standard library are more expensive. For example, arithmetic operations are cheap, while reduction operations (using the inter-PE communication) are somewhat more expensive, albeit still cheaper than data transfer operations between mono and poly memories.

Efficiency mandates only providing language constructs that can be reliably implemented on the SATA using standard compiler technology. History shows that languages that are difficult to implement are also rarely successful (HPF is a dismal example [8]).

Portability is important because the CSX architecture family (§2.1) does not fix the PE array size. Also, as the number of PE cores increases, other (than linear) interconnect topologies could be introduced.

Designing C^n as a C extension provides a known starting point for a large community of C programmers. In addition, software development tools can be written by making (small) modifications to existing ones, rather than from scratch.

4 C^n Outline

In this section we outline the most salient features of C^n. We give further rationale behind some design decisions in §5.

4.1 Types

C^n extends C with two additional keywords that can be used as part of the declaration qualifier for declarators, *i.e.* logically amend the type system [9]. These keywords are *multiplicity qualifiers* and allow the programmer to specify a memory space in which a declared object resides. The new keywords are:

- **mono**: for declaring an object in the mono domain (*i.e.* one copy exists in main memory);
- **poly**: for declaring an object in the poly domain (*i.e.* one copy per PE in its local memory).

The default multiplicity is **mono**. Wherever a multiplicity qualifier may be used, an implicit **mono** is assumed, unless an explicit **poly** is provided. A consequence of this is that all C programs are valid C^n programs, with the same semantics. Thus, C^n is a superset of C (but see §5.2).

Basic types. C^n supports the same basic types as C. They can be used together with a multiplicity qualifier to produce declarations (only one of the qualifiers can be used in a basic type declaration). Some example declarations follow:

```
poly int i; // multiple copies in PE local (poly) memory
mono unsigned cu; // a single copy in main (mono) memory
unsigned long cs; // a single copy in main (mono) memory
```

Pointer types. The pointer types in C^n follow similar rules to those in C. Pointer declarations consist of a base type and a pointer. The declaration on the left of the asterisk represents the base type (the type of the object that the pointer points to). The declaration on the right of the asterisk represents the pointer object itself. It is possible to specify the multiplicity of either of these entities in the same way as **const** and **volatile** work in C. For example:

```
poly int * poly sam; // poly pointer to poly int
poly int * frodo;    // mono pointer to poly int
int * poly bilbo;    // poly pointer to mono int
```

Thus, there are four different ways of declaring pointers with multiplicity qualifiers:

- mono pointer to mono object (*e.g.* **mono int * mono**);
- mono pointer to poly object (*e.g.* **poly int * mono**);
- poly pointer to mono data (*e.g.* **mono int * poly**);
- poly pointer to poly data (*e.g.* **poly int * poly**).

Note that in the case of a poly pointer, multiple copies of the pointer exist, potentially pointing to different locations.

As in C, pointers are used to access memory. The compiler allocates named poly objects to the same address within each PE local memory. Thus, taking the address of a named object (whether mono or poly) always yields a mono pointer.

Array types. The syntax for array declaration in C^n is similar to that in C. It is possible to use a multiplicity qualifier in an array declaration. Consider the declaration `poly int A[42];`. The multiplicity qualifier applies only to the base type of the array (*i.e.* to the type of array elements). This declaration will reserve a poly space for 42 integers at the same location on each PE. Similar to C, we can say that the array name is coerced to the address of its first element, which is a mono pointer to poly object (*e.g.* `poly int * mono`).

There are some additional restrictions when dealing with arrays, specifically with array subscripting, discussed in §4.3.

Aggregate types. Similar to C, C^n distinguishes between declaration and definition of objects. A declaration is where an object type is specified but no object of that type is created, *e.g.* a struct type declaration. A definition is where an object of a particular type is created, *e.g.* a variable definition. Structs and unions in C^n match their C equivalents, the only difference being the type of fields one can specify inside a struct or union type declaration.

Standard C allows essentially any declaration as a field, with the exception of storage class qualifiers. C^n allows the same with the additional restriction that fields cannot have multiplicity qualifiers. This is because a struct type declaration just specifies the structure of memory. Memory is not allocated until an instance of that struct type is created. Thus, putting a poly field in a struct declaration and then defining an instance of that struct in the mono domain would result in having contradictory multiplicity specifications. (Similarly for a mono field in a struct instance defined in the poly domain.) For example:

```
// legal struct declaration
struct _A {
    int a;
    float b;
};
poly struct _A kaiser;

// illegal struct declaration
struct _B {
    poly int a;   // illegal use of multiplicity
    mono float b; // illegal use of multiplicity
};
mono struct _B king; // where should king.a go?
poly struct _B tsar; // where should tsar.b go?
```

Multiplicity qualifiers, however, can be used on the base type of pointers (otherwise pointers to poly data could not be declared as fields). For example:

```
union _C { // define a union
    poly int * a;
    mono int * poly * b;
};
poly union _C fred;
mono union _C barney;
```

In the declaration of `fred`, the field `a` is created as a poly pointer to a poly int (**`poly int * poly`**) and `b` is created as a poly pointer to a poly pointer to a mono int (**`mono int * poly * poly`**). In the declaration of `barney`, `a` is created as a mono pointer to a poly int (**`poly int * mono`**) and `b` is created as a mono pointer to a poly pointer to a mono int (**`mono int * poly * mono`**).

4.2 Expressions

C^n supports all the operators of C. Mono and poly objects can usually be used interchangeably in expressions (but see §4.3 for exceptions).

Note that the result of any expression involving a mono and a poly object invariably has a poly type. Thus, mono objects are promoted to the poly domain in mixed expressions. In the following example,

```
poly int x; int y; x = x + y;
```

where `y` is promoted to **`poly int`** before being added to (every copy of) `x`. (Technically, the promotion is done by broadcasting values from the mono register file to the poly execution unit.)

4.3 Assignment Statements

Assignment within a domain (*i.e.* poly to poly, or mono to mono) is always legal and has the obvious behaviour.

A mono value can also be assigned to a poly variable. In this case the same value is assigned to every copy of the variable on each PE. For example, **`poly int`** `x = 1;` results in (every copy of) `x` having the value of 1. (Again, the architecture supports such assignments by broadcasting the mono value to the poly unit.)

It is not obvious, however, what should happen when assigning a poly to a mono, *i.e.* when taking data from multiple copies of the poly variable and storing it in the single mono variable. Therefore direct assignment from the poly domain to the mono domain is disallowed in C^n.

Note that certain C^n expressions are disallowed, as otherwise they would require an implicit data transfer from mono to poly memory. One such expression is dereferencing of a poly pointer to mono data (*e.g.* **`mono int * poly`**). Attempting to dereference such a pointer would result in poly data, but the data is stored in the mono domain and would therefore need to be copied to poly memory. Since broadcasting can only send a single mono value to the poly unit at a time, such a copy would involve expensive programmed I/O mechanisms. Therefore, allowing such dereferencing would conflict with the design goal that the core language operations should be cheap. Thus, the compiler reports dereferencing a poly pointer to mono data as an error.

Since, following C, C^n treats `x[i]` as equivalent to `*(x + i)`, it follows that indexing a mono array with a poly expression is also forbidden, because it would implicitly dereference a poly pointer to mono data.

In all the cases when data transfers between mono and poly memories are required, the programmer has to use memory copy routines from the standard C^n library.

4.4 Reduction Operations

Many parallel algorithms require reducing a vector of values to a single scalar value. In addition to the core operations defined above, the C^n standard library provides `sum`, `times`, and bit-wise operations defined for basic types for reducing a poly value into a mono value. For example, on an array of n PEs,

```
poly float x; mono float y; ...
y = reduce_mono_sum(x);
```

means

```
y = x(0) + x(1) + ... + x(n-1);
```

where `x(i)` refers to the value of `x` on ith PE.

For some algorithms another form of reduction is useful: logically, a poly value is reduced to an intermediate mono value which is then broadcast into a result poly value. Thus,

```
poly float x, z; ...
z = reduce_poly_sum(x);
```

means

```
z(0) = ... = z(n-1) = x(0) + x(1) + ... + x(n-1);
```

Both forms can be efficiently implemented on the CSX using the inter-PE swazzle path. The order in which the result is evaluated is unspecified.

4.5 Control Statements

The basic control flow constructs in C^n are the same as in C. Conditional expressions, however, can be of mono or poly domain. Consider the **`if`** statement:

```
if(expression) { statement-list }
```

A mono expression for the condition affects both the mono and poly execution units. If the expression is false, the statement list will be skipped entirely, and execution will continue after the **`if`** statement.

A poly expression for the condition can be true on some PEs and false on others. This is where the PE enable state (described in §2.1) comes in: all the PEs for which the condition is false are disabled for the duration of executing the statement list. The statement list is executed (even if all PEs are disabled by the condition), but any poly statements (*e.g.* assignments to poly variables) have an affect only on the enabled PEs. Mono statements inside a poly **`if`**, however, get executed irrespective of the conditions. Consider the following example:

```
poly int foo = 0; mono int bar = 1;
if(get_penum()%2 == 0) { // disable all odd PEs
  foo = 1; // foo is 1 on even and 0 on odd PEs
  bar = 2; // bar is 2
}
```

Effectively, a poly condition is invisible to any mono code inside that **`if`** statement.[1] This language design choice may seem counterintuitive and is indeed a controversial point in the design of SIMD languages, to which we return in §5.2.

A poly condition in an **`if..else`** statement implies that for the **`if`**-clause all the PEs on which the condition evaluates to true are enabled, and the others are disabled. Then, for the **`else`**-clause, the enable state of all the PEs is inverted: those PEs that were enabled by the condition are disabled and vice-versa.

Conditional statements can be nested just as in C. Poly statements are only executed on PEs when all the nested conditional expressions evaluate to true.

These rules, of course, dictate different compilation of conditional statements. Mono conditional statements result in generating branch instructions, while poly conditional statements result in generating poly instructions enabling and disabling PEs (enable stack operations on the CSX; see §7 for more details).

Similar principles apply to loop constructs **`for`**, **`while`** and **`do..while`**. A loop with a poly control expression executes until the loop condition is false on every PE. Note that a poly loop can iterate zero times, so in that case, unlike the **`if`** statement, even mono statements in its body will not be executed.

4.6 Functions

Multiplicity qualifiers can be specified for the return type of a function, as well as the types of any arguments. We refer to a function as being mono or poly according to whether its return type is mono or poly.

A return statement from a mono function behaves exactly as expected by transferring control to the point after the call statement. A poly function does not actually return control until the end of the function body. A return from a poly function works by disabling the PEs which execute it. Other PEs execute the rest of the function code. When the function returns, all PEs have their enable state restored to what it was on entry. Note that all mono code in the function is always executed (unless branched over by mono conditional statements).

5 C^n Design Rationale

5.1 Low-Level Abstraction

Early SIMD programming languages for the Illiac IV computer included Glypnir [10], with syntax based on Algol 60, and CFD [11], based on Fortran. The main design goal of these languages was “to produce a useful, reliable, and efficient programming

[1] It way be worth noting that the block structure of code is still relevant. So, for example, any mono declarations within the body of a poly conditional statement are local to that body.

tool with a high probability of success" [10]. Given the state of compiler technology in the early 1970s, the languages could not *both* be machine independent *and* satisfy this goal.[2] In addition to explicit means for specifying storage allocation and program control as in C^n, these languages even provided means for accessing subwords and registers.

Many vector processing languages have appeared since then, providing higher levels of abstraction for array-based operations (*e.g.* Actus [12], Vector C [13], High Performance Fortran [8], *etc.*). Unfortunately, such languages present greater complexity for the implementors because of data alignment, storage allocation and communication issues on SIMD machines (*e.g.* see [14,15]).

C^n builds on the C strength of "solving" most implementation efficiency problems by leaving them to the programmers. While not the most desirable solution, it relieves the programmers from solving the same problems using assembly language. In §6 we argue that C^n can be seen as an intermediate representation for compiling from higher level parallel languages.

5.2 C^n and Other SIMD Dialects of C

C^n is by no means the first C dialect for SIMD programming. Notable examples include C* [16] for programming the Connection Machine models, MPL [17] for the MasPar models, and 1DC (one-dimensional C) [18] for the IMAP models.

All these languages intentionally reflect their respective architectures. The unifying theme is the use of a keyword to specify multiple instance variables (**`poly`** in C*, **`plural`** in MPL, **`sep`** in 1DC). The language differences stem from the differences between the architectures and design goals.

Communication. While the **`poly`** keyword implies the physical data distribution, C* enshrines the viewpoint that no parallel version of C can abandon the uniform address space model without giving up its claim to be a superset of C. Uniform memory means that a PE can have a pointer `p` into the memory of another PE. For example, the statement `*p = x;` means "send message `x` to a location pointed to by `p`". Thus, C* relies on pointers for interprocessor communication.

This C* feature is underpinned by Connection Machine's key capability allowing any PE to establish a connection with any other PE in the machine (via the global routing mechanism or local meshes). Still, the programmer relies on the compiler writer's ability to implement pointer-based communication efficiently. Perhaps, this is the reason why, in contrast, MasPar's MPL provides explicit communication operators, although the MasPar has connection capabilities similar to the Connection Machine.

The IMAP architectures have a linear nearest neighbour interconnect, as does the CSX. 1DC supports special operators for nearest neighbour communication, while C^n relegates communication functions to the C^n standard library.

Dereferencing pointers. Even rejecting the pointer-based communication in C* does not mean that the compiler will be able to optimise poorly written code; in particular,

[2] Indeed, the compiler for Tranquil—the first Illiac IV language—was abandoned because of implementation difficulties and lack of resources [10].

code that makes a heavy use of costly DMA transfers between mono and poly memories. The C^n ban on dereferencing poly pointers to mono data (which would transfer only several bytes at a time) aims to shut the door in front of the abusing programmers. In contrast, MPL does not sacrifice convenience for efficiency and allows all pointer operations.

Poly conditional statements. In §4.5, we discussed the behaviour of poly **`if`** statements in C^n: even if the poly condition is false on every PE, the statement list is executed regardless, including all mono statements in the list. This model of execution is easy to implement on the CSX by inserting operations on the hardware enable stack (see §7). The same model was used in MPL [19].

The designers of C* followed the other route by adopting the "Rule of Local Support" [16, §6.2]: if the poly condition is false on every PE, then the statement list is not executed at all. The Rule of Local Support required extra implementation trouble but preserved the usual model of a **`while`** loop in terms of **`if`** and **`goto`** statements. The C* designers, nevertheless, admitted that their rule occasionally also caused inconvenience and confusion.

Deciding on whether to preserve or to change the semantics of familiar statements when extending a sequential language for parallelism is hard, and may even drive the designers to a thought that designing a language from scratch is a better idea (*e.g.* see the history of ZPL [4]). In the case of **`if`** statements, the solution does not need to be that radical and could merely come as using different keywords in a more general language supporting both execution models (for example, **`ifp`** for the C^n/MPL model and **`ifm`** for the C* model).

6 C^n as Intermediate Language

SIMD array supercomputers went out of fashion in the early 1990s when clusters of commodity computers proved to be more cost effective. Ironically, vector-style instructions reemerged in commodity processors under the name of SIMD (or multimedia) extensions, such as Intel MMX/SSE and PowerPC AltiVec.

The principal difference between "real" (array) SIMD and vector extensions is that the latter operate on data in long registers and do not have local memory attached to each PE as in SIMD arrays. In terms of the C^n language, this means that it would be inefficient (if not impossible) to use pointers to poly data when programming in C^n for vector extensions. This is because it would not be straightforward to compile C^n programs expressing memory operations that are not supported by the hardware.

We argue, however, that the C^n language can be regarded as a "portable assembly" language for both SIMD and vector architectures. Given an architecture description, high-level languages providing more abstract array-based operations can be translated to C^n and then efficiently mapped to machine code by the C^n compiler.

For example, consider a Fortran 90 style statement `B[0:959] = A[0:959]+42;` where `A[]` and `B[]` are arrays of floats. This statement loads 960 values from `A`, adds 42 to each, and stores 960 results to `B`. Suppose that the target architecture is a vector machine with 96 elements per vector register, hence the statement parallelism needs to be folded 10 times to fit the hardware by *strip mining* [20,15].

For a vector machine, the declaration **poly float** va, vb; can be thought of as defining two vector registers in a similar way as **vector float** vc; can be used to define a (4-element) vector in the AltiVec API [21]. The vector statement can then be strip-mined *horizontally* [15], resulting in:

```
const poly int me = get_penum(); // get PE number
mono float * poly pa = A + me;
mono float * poly pb = B + me;
for(int i = 0; i < 10; i++, pa += 96, pb += 96) {
  va  = *pa; // load 96-element vector
  vb  = va + 42;
  *pb = vb; // store 96-element vector
}
```

(Note that here we have lifted the C^n ban on dereferencing poly pointers to mono data, which is the artefact of distributed memory SIMD arrays.)

The same strategy works on a SIMD machine having 96 PEs: 10 vector indirect loads (one on each iteration) are replaced with 10 DMA transfers from mono to poly memory, 10 vector stores with 10 DMA transfers from poly to mono memory; each transfer moves a single float to/from a PE. It is more efficient, however, to strip-mine *vertically* [15], which in pseudo-vector notation can be written as:

```
poly float pA[10], pB[10];
pA[0:9] = A[10*me:10*me+9]; // DMA mono to poly
pB[0:9] = pA[0:9] + 42;
B[10*me:10*me+9] = pB[0:9]; // DMA poly to mono
```

This requires only two DMA transfers of 10 floats each. Given high DMA start-up costs, vertical strip-mining is several times more efficient. Hence, the compiler should favour the second form on SIMD machines. (This is a trivial example of the decisions we referred to in §5.1 that the compiler needs to make to efficiently compile high-level vector abstractions).

To summarise, C^n code can be seen as an intermediate language, from which target code can be generated. Once a C^n compiler is written and well-tuned, the quality of target code should principally depend on the ability of the (machine-description driven) front-end to generate efficient intermediate code from a higher level parallel language. Even if the front-end generates suboptimal intermediate code, the performance-conscious programmer still has a fallback to the intermediate language, rather than to assembly.

C^n has already been targeted from a subset of OpenMP [22]. We believe that C^n can also be targeted from high-level data parallel libraries such as MSR Accelerator [23] and memory hierarchy oriented languages such as Stanford Sequoia [24].

7 C^n Compiler Implementation

ClearSpeed has developed a C^n optimising compiler for the CSX architecture using the CoSy compiler development framework [25]. Small modifications to the front-end were needed to support the multiplicity qualifiers in the source code. Each type is supplemented with a flag indicating whether it is mono or poly. A special phase was written to recognise poly conditional statements and transform them into a form of predicated execution. For example, assuming that both `x` and `y` are in the poly domain,

```
if (y > 0) {
  x = y;
}
```

becomes

```
enablestate = push(y > 0, enablestate);
x ?= y;
enablestate = pop(enablestate);
```

where the predicated assignment operator `?=` assigns its *rhs* expression to the *lhs* location only on those PEs where all the bits of the hardware enable stack (represented by the variable `enablestate`) are 1.

Predicated execution requires careful rethinking of standard optimisations. For example, the standard register allocation algorithm via coloring has to recognise that liveness of a virtual poly register is not simply implied by its def-use chain but is also a function of the enable state. This is not, however, a new problem (*e.g.* see [26]).

7.1 C^n Compiler Performance

ClearSpeed has developed a number of interesting applications in C^n, including functions from the molecular dynamics simulation package AMBER, Monte-Carlo option pricing simulations, implementations of 2D and 3D FFTs, and others.

The C^n design choice of only including features that can be efficiently implemented using standard compiler technology pays off handsomely, since the implementors can concentrate their efforts on optimisations that the programmer expects the compiler to get right. For example, Fig. 1 shows on 20 benchmarks the performance improvement achieved by the version 3.0 of the C^n compiler over the version 2.51.

Much of the improvement comes from the work on the register allocator and other optimisations becoming aware of poly variables. Particular optimisations, *e.g.* loop-invariant code motion, have proved to benefit from the target processor's ability to issue *forced* poly instructions which are executed regardless of the enable state.

Code for the `amber` benchmark generated by the current version of the compiler performs within 20% of hand-coded assembly. (No other similar data is available for comparison, because it makes little sense to re-implement assembly programs in C^n, other than to improve portability.)

8 Future Work and Conclusion

Programming in C^n tends to require the programmer to restructure programs to expose the parallelism lost when writing in a sequential language. Essentially, the C^n programmer indicates a parallel loop. We believe it is possible to ease the programmer's task of annotating sequential C programs (but expressing parallel algorithms) with the **`poly`** qualifier, if the programmer can assert certain properties.

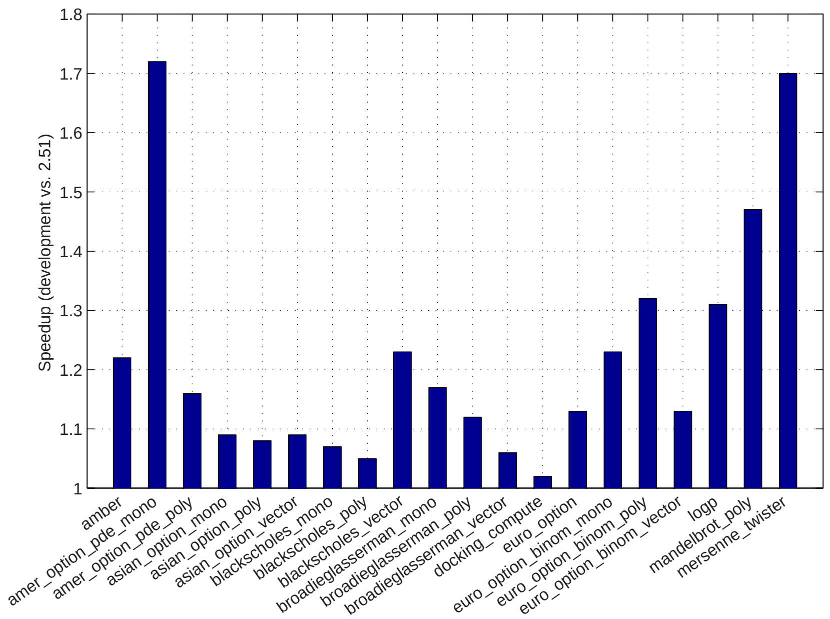


Fig. 1. Performance improvement of code generated by the compiler version 3.0 over code generated by the compiler version 2.51

We are developing an auto-parallelising compiler module converting C code to valid C^n code using programmer's annotations based on the concept of delayed writes [27]. The resulting C^n code is then fed into the optimisation and code generation phases of the C^n compiler.

ClearSpeed has developed a set of production quality tools targeting the CSX array architecture, including an optimising compiler, assembler, linker and debugger, that make software development in C^n a viable alternative to hand-coding in assembly as loss in performance is far outweighed by the advantages of high-level language programming.

References

1. Parhami, B.: SIMD machines: do they have a significant future? SIGARCH Comput. Archit. News 23(4), 19–22 (1995)
2. Barnes, G.H., Brown, R.M., Kato, M., Kuck, D.J., Slotnick, D.L., Stokes, R.A.: The Illiac IV computer. IEEE Trans. Computers C-17(8), 746–757 (1968)
3. ClearSpeed Technology: The CSX architecture, `http://www.clearspeed.com/`
4. Snyder, L.: The design and development of ZPL. In: Proc. of the third ACM SIGPLAN conference on History of programming languages (HOPL III), pp. 8–1–8–37. ACM Press, New York (2007)
5. Slotnick, D.: The conception and development of parallel processors—a personal memoir. IEEE Annals of the History of Computing 4(1), 20–30 (1982)

6. Wilkes, M.V.: The lure of parallelism and its problems. In: Computer Perspectives. Morgan Kaufmann, San Francisco (1995)
7. Snyder, L.: Type architecture, shared memory and the corollary of modest potential. Annual Review of Computer Science 1, 289–317 (1986)
8. Kennedy, K., Koelbel, C., Zima, H.: The rise and fall of High Performance Fortran: an historical object lesson. In: Proc. of the third ACM SIGPLAN conference on History of programming languages (HOPL III), pp. 7–1–7–22. ACM Press, New York (2007)
9. American National Standards Institute: ANSI/ISO/IEC 9899-1999: Programming Languages – C (1999)
10. Lawrie, D.H., Layman, T., Baer, D., Randal, J.M.: Glypnir—a programming language for Illiac IV. Commun. ACM 18(3), 157–164 (1975)
11. Stevens Jr., K.: CFD—a Fortran-like language for the Illiac IV. SIGPLAN Not. 10(3), 72–76 (1975)
12. Perrott, R.H.: A language for array and vector processors. ACM Trans. Program. Lang. Syst. 1(2), 177–195 (1979)
13. Li, K.C., Schwetman, H.: Vector C: a vector processing language. Journal of Parallel and Distributed Computing 2(2), 132–169 (1985)
14. Knobe, K., Lukas, J.D., Steele Jr., G.L.: Data optimization: allocation of arrays to reduce communication on SIMD machines. J. Parallel Distrib. Comput. 8(2), 102–118 (1990)
15. Weiss, M.: Strip mining on SIMD architectures. In: Proc. of the 5th International Conference on Supercomputing (ICS), pp. 234–243. ACM Press, New York (1991)
16. Rose, J.R., Steele Jr., G.L.: C*: An extended C language for data parallel programming. In: Proc. of the 2nd International Conference on Supercomputing (ICS), vol. 2, pp. 2–16 (1987)
17. MasPar Computer Corporation: MasPar Programming Language (ANSI C compatible MPL) Reference Manual (1992)
18. Kyo, S., Okazaki, S., Arai, T.: An integrated memory array processor for embedded image recognition systems. IEEE Trans. Computers 56(5), 622–634 (2007)
19. Christy, P.: Software to support massively parallel computing on the MasPar MP-1. In: Proc. of the 35th IEEE Computer Society International Conference (Compcon Spring), pp. 29–33 (1990)
20. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures. Morgan Kaufmann, San Francisco (2002)
21. Freescale Semiconductor: AltiVec technology programming interface manual (1999)
22. Bradley, C., Gaster, B.R.: Exploiting loop-level parallelism for SIMD arrays using OpenMP. In: Proc. of the 3rd International Workshop on OpenMP (IWOPM) (2007)
23. Tarditi, D., Puri, S., Oglesby, J.: Accelerator: using data parallelism to program GPUs for general-purpose uses. In: Proc. of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII), pp. 325–335. ACM Press, New York (2006)
24. Fatahalian, K., Horn, D.R., Knight, T.J., Leem, L., Houston, M., Park, J.Y., Erez, M., Ren, M., Aiken, A., Dally, W.J., Hanrahan, P.: Sequoia: programming the memory hierarchy. In: Proc. of the 2006 ACM/IEEE Conference on Supercomputing (SC), pp. 83–92. ACM Press, New York (2006)
25. ACE Associated Compiler Experts: The CoSy compiler development system, `http://www.ace.nl/`
26. Kim, H.: Region-based register allocation for EPIC architectures. PhD thesis, Department of Computer Science, New York University (2001)
27. Lokhmotov, A., Mycroft, A., Richards, A.: Delayed side-effects ease multi-core programming. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par 2007. LNCS, vol. 4641. Springer, Heidelberg (2007)

Multidimensional Blocking in UPC

Christopher Barton[1], Călin Caşcaval[2], George Almasi[2], Rahul Garg[1], José Nelson Amaral[1], and Montse Farreras[3]

[1] University of Alberta, Edmonton, Canada
[2] IBM T.J. Watson Research Center
[3] Universitat Politècnica de Catalunya
Barcelona Supercomputing Center

Abstract. Partitioned Global Address Space (PGAS) languages offer an attractive, high-productivity programming model for programming large-scale parallel machines. PGAS languages, such as Unified Parallel C (UPC), combine the simplicity of shared-memory programming with the efficiency of the message-passing paradigm by allowing users control over the data layout. PGAS languages distinguish between private, shared-local, and shared-remote memory, with shared-remote accesses typically much more expensive than shared-local and private accesses, especially on distributed memory machines where shared-remote access implies communication over a network.

In this paper we present a simple extension to the UPC language that allows the programmer to block shared arrays in multiple dimensions. We claim that this extension allows for better control of locality, and therefore performance, in the language.

We describe an analysis that allows the compiler to distinguish between local shared array accesses and remote shared array accesses. Local shared array accesses are then transformed into direct memory accesses by the compiler, saving the overhead of a locality check at runtime. We present results to show that locality analysis is able to significantly reduce the number of shared accesses.

1 Introduction

Partitioned Global Address Space (PGAS) languages, such as UPC [14], Co-Array Fortran [10], and Titanium [16], extend existing languages (C, Fortran and Java, respectively) with constructs to express parallelism and data distributions. They are based on languages that have a large user base and therefore there is a small learning curve to move codes to these new languages.

We have implemented several parallel algorithms — stencil computation and linear algebra operations such as matrix-vector and Cholesky factorization — in the UPC programming language. During this effort we identified several issues with the current language definition, such as: rudimentary support for data distributions (shared arrays can be distributed only block cyclic), flat threading model (no ability to support subsets of threads), and shortcomings in the collective definition (no collectives on subsets of threads, no shared data allowed as target for collective operations, no concurrent participation of a thread in multiple collectives). In addition, while implementing a compiler and runtime system we found that naively translating all shared accesses to runtime calls is prohibitively expensive. While the language supports block transfers and cast operations that could alleviate some of the performance issues, it is more convenient to address these problems through compiler optimizations.

Tackling some of these issues, this paper makes the following contributions:

- propose a new data distribution directive, called multidimensional blocking, that allows the programmer to specify n-dimensional tiles for shared data (see Section 2);
- describe a compile-time algorithm to determine the locality of shared array elements and replace references that can be proven to be locally owned by the executing thread with direct memory accesses. This optimization reduces the overhead of shared memory accesses and thus brings single thread performance relatively close to serial implementations, thereby allowing the use of a scalable, heavier, runtime implementation that supports large clusters of SMP machines (see Section 3);
- present several benchmarks that demonstrate the benefits of the multidimensional blocking features and the performance results of the locality analysis; these performance results were obtained on a cluster of SMP machines, which demonstrates that the flat threading model can be mitigated through knowledge in the compiler of the machine architecture (Section 5).

2 Multidimensional Blocking of UPC Arrays

In this section we propose an extension to the UPC language syntax to provide additional control over data distribution: tiled (or *multiblocked*) arrays. Tiled data structures are used to enhance locality (and therefore performance) in a wide range of HPC applications [2]. Multiblocked arrays can help UPC programmers to better express these types of applications, allowing the language to fulfill its promise of allowing both high productivity and high performance. Also, having this data structure available in UPC facilitates using library routines, such as BLAS [4], in C or Fortran that already make use of tiled data structures.

Consider a simple stencil computation on a 2 dimensional array that calculates the average of the four immediate neighbors of each element.

```
shared double A[M][N];
...
for (i=1..M-2,j=1..N-2)
  B[i][j] = 0.25*(A[i-1][j]+A[i+1][j]+A[i][j-1]+A[i][j+1]);
```

Since it has no data dependencies, this loop can be executed in parallel. However, the naive declaration of A above yields suboptimal execution, because e.g. A[i-1][j] will likely not be on the same UPC thread as A[i][j] and may require inter-node communication to get to. A somewhat better solution allowed by UPC is a striped 2D array distribution:

```
shared double [M*b] A[M][N];
```

$M \times b$ is the *blocking factor* of the array; that is, the array is allocated in contiguous blocks of this size. This however, limits parallelism to $\frac{N}{b}$ processors and causes $O(\frac{1}{b})$ remote array accesses. By contrast, a tiled layout provides $\frac{M \times N}{b^2}$ parallelism and $O(\frac{1}{b^2})$ of the accesses are remote. Typical MPI implementations of stencil computation tile the array and exchange "border regions" between neighbors before each iteration. This approach is also possible in UPC:

```
struct block { double tile[b][b]; };
shared block A[M/b][N/b];
```

However, the declaration above complicates the source code because two levels of indexing are needed for each access. We cannot pretend that A is a simple array anymore. We propose a language extension that can declare a tiled layout for a shared array, as follows:

```
shared <type> [b0][b1]...[bn] A[d0][d1] ... [dn];
```

Array A is an n-dimensional tiled (or "multi-blocked") array with each tile being an array of dimensions $[b0][b1]...[bn]$. Tiles are understood to be contiguous in memory.

2.1 UPC Array Layout

To describe the layout of multiblocked arrays in UPC, we first need to discuss conventional shared arrays. A UPC array declared as below:

```
shared [b] <type> A[d0][d1]...[dn];
```

is distributed in memory in a block-cyclic manner with blocking factor b. Given an array index $\mathbf{v} = v_0, v_1, ...v_{n-1}$, to locate element $A[\mathbf{v}]$ we first calculate the linearized row-major index (as we would in C):

$$L(\mathbf{v}) = v_0 \times \prod_{j=1}^{n-1} d_j + v_1 \times \prod_{j=2}^{n-1} d_j + ... + v_{n-1} \tag{1}$$

Block-cyclic layout is based on this linearized index. We calculate the UPC *thread* on which array element $A[\mathbf{v}]$ resides. Within the local storage of this thread the array is kept as a collection of blocks. The *course* of an array location is the block number in which the element resides; the *phase* is its location within the block.

$$\begin{cases} thread(A, \mathbf{v}) & ::= \left\lfloor \frac{L(\mathbf{v})}{b} \right\rfloor \bmod \mathcal{T} \\ phase(A, \mathbf{v}) & ::= L(\mathbf{v}) \bmod b \\ course(A, \mathbf{v}) & ::= \left\lfloor \frac{L(\mathbf{v})}{b \times \mathcal{T}} \right\rfloor \end{cases}$$

Multiblocked arrays: The goal is to extend UPC syntax to declare tiled arrays while minimizing the impact on language semantics. The internal representation of multiblocked arrays should not differ too much from that of standard UPC arrays. Consider a multiblocked array A with dimensions $D = \{d_0, d_1, ...d_n\}$ and blocking factors $B = \{b_0, b_1, ...b_n\}$. This array would be allocated in $k = \prod_{i=0}^{n-1} \left\lceil \frac{d_i}{b_i} \right\rceil$ blocks (or tiles) of $b = \prod_{i=0}^{n-1} b_i$ elements. We continue to use the concepts of *thread*, *course* and *phase* to find array elements. However, for multiblocked arrays two linearized indices must be computed: one to find the block and another to find an element's location within a block. Note the similarity of Equations 2 and 3 to Equation 1:

$$L_{in-block}(\mathbf{v}) = \sum_{k=0}^{n-1}((v_k \bmod b_k) \times \prod_{j=k+1}^{n-1} b_j) \tag{2}$$

$$L(\mathbf{v}) = \sum_{k=0}^{n-1}(\left\lfloor \frac{v_k}{b_k} \right\rfloor \times \prod_{j=k+1}^{n-1} \left\lceil \frac{d_j}{b_j} \right\rceil) \tag{3}$$

The *phase* of a multiblocked array element is its linearized in-block index. The *course* and *thread* are calculated with a cyclic distribution of the block index, as in the case of regular UPC arrays.

$$\begin{cases} thread(A, \mathbf{v}) & ::= \left\lfloor \frac{L(\mathbf{v})}{\prod_{i=0}^{n-1} b_i} \right\rfloor \bmod \mathcal{T} \\ phase(A, \mathbf{v}) & ::= L_{in-block}(\mathbf{v}) \\ course(A, \mathbf{v}) & ::= \left\lfloor \frac{L(\mathbf{v})}{\prod_{i=0}^{n-1} b_i \times \mathcal{T}} \right\rfloor \end{cases} \tag{4}$$

Array sizes that are non-multiples of blocking factors: The blocking factors of multiblocked arrays are not required to divide their respective dimensions, just as blocking factors of regular UPC arrays are not required to divide the array's dimension(s). Such arrays are padded in every dimension to allow for correct index calculation.

2.2 Multiblocked Arrays and UPC Pointer Arithmetic

The address of any UPC array element (even remote ones) can be taken with the `upc_addressof` function or with the familiar & operator. The result is called a *pointer-to-shared*, and it is a reference to a memory location somewhere within the space of the running UPC application. In our implementation a pointer-to-shared identifies the base array as well as the thread, course and phase of an element in that array.

UPC pointers-to-shared behave much like pointers in C. They can be incremented, dereferenced, compared etc. The familiar pointer operators (`*`, &, `++`) are available. A series of increments on a pointer-to-shared will cause it to traverse a UPC shared array in row-major order.

Pointers-to-shared can also used to point to multiblocked arrays. Users can expect pointer arithmetic and operators to work on multiblocked arrays just like on regular UPC shared arrays.

Affinity, casting and dynamic allocation of multiblocked arrays: Multiblocked arrays can support affinity tests (similar to the `upc_threadof` function) and type casts the same way regular UPC arrays do.

Dynamic allocation of UPC shared arrays can also be extended to multiblocked arrays. UPC primitives like `upc_all_alloc` always return shared variables of type `shared void *`; multiblocked arrays can be allocated with such primitives as long as they are cast to the proper type.

2.3 Implementation Issues

Pointers and dynamic allocation of arrays: Our current implementation supports only statically allocated multiblocked arrays. Dynamically allocated multiblocked arrays could be obtained by casting dynamically allocated data to a shared multiblocked type, making dynamic multiblocked arrays a function of correct casting and multiblocked pointer arithmetic. While correct multiblocked pointer arithmetic is not conceptually difficult, implementation is not simple: to traverse a multiblocked array correctly, a pointer-to-shared will have to have access to all blocking factors of the shared type.

Processor tiling: Another limitation of the current implementation is related to the cyclic distribution of blocks over UPC threads. An alternative would be to specify a processor grid to distribute blocks over. Equation 3 would have to be suitably modified to take thread distribution into consideration. We have not implemented this yet in the UPC runtime system, although performance results presented later in the paper clearly show the need for it.

Hybrid memory layout: Our UPC runtime implementation is capable of running in mixed multithreaded/multinode environments. In such an environment locality is interpreted on a per-node basis, but array layouts have to be on a per-UPC-thread basis to be compatible with the specification. This is true both for regular and multiblocked arrays.

3 Locality Analysis for Multi-dimensional Blocking Factors

This section describes a compile-time analysis for multi-dimensional blocking factors in UPC shared arrays. The analysis considers loop nests that contain accesses to UPC shared arrays and finds shared array references that are provably local (on the same UPC thread) or shared local (on the same node in shared memory, but on different UPC threads). All other shared array references are potentially remote (reachable only via inter-node communication).

The analysis enables the compiler to refactor the loop nest to separate local and remote accesses. Local and shared local accesses cause the compiler to generate simple memory references; remote variable accesses are resolved through the runtime with a significant remote access overhead. We consider locality analysis crucial to obtaining good performance with UPC.

In Figure 1 we present a loop nest that will be used as an example for our analysis. In this form the shared array element in the affinity test — the last parameter in the `upc_forall` statement — is formed by the current loop-nest index, while the single element referenced in the loop body has a displacement, with respect to the affinity expression, specified by the distance vector $\mathbf{k} = [k_0, k_1, \ldots, k_{n-1}]$. Any loop nest in which the index for each dimension, both in the affinity test and in the array reference, is an affine expression containing only the index in the corresponding dimension can be transformed to this cannonical form.[1] Table 1 summarizes the notation used throughout

[1] An example of a loop nest that cannot be transformed to this cannonical form is a two-level nest accessing a two-dimensional array in which either the affinity test or the reference contains an expression such as `A[`$v_0 + v_1$`][`v_1`]`.

```
shared [$b_0$][$b_1$]$\cdots$[$b_{k-1}$] int A[$d_0$][$d_1$]$\cdots$[$d_{k-1}$];
for($v_0$=0 ; $v_0 < d_0 - k_0$ ; $v_0$++)
    for($v_1$=0 ; $v_1 < d_1 - k_1$ ; $v_1$++){
        ...
        upc_forall($v_{n-1}$=0 ; $v_{n-1} < d_{n-1} - k_{n-1}$ ; $v_{n-1}$++ ; &A[$v_0$][$v_1$]...[$v_{n-1}$])
                A[$v_0 + k_0$][$v_1 + k_1$]...[$v_{n-1} + k_{n-1}$] = $v_0 * v_1 * \ldots * v_{n-1}$;
    }
```

Fig. 1. Multi-level loop nest that accesses a multi-dimensional array in UPC

Table 1. Expressions used to compute the node ID that each element $A[\mathbf{v}]$ of array A belongs to

Ref	Expression	Description
1	n	number of dimensions
2	b_i	blocking factor in dimension i
3	d_i	array size in dimension i
4	v_i	position index in dimension i
5	$\mathbf{v} = [v_0, v_1, \ldots, v_{n-1}]$	Index of an array element
6	$\mathcal{T}$	number of threads
7	t	number of threads per node
8	$\mathcal{B}_i = \left\lfloor \frac{v_i}{b_i} \right\rfloor$	Block index in dimension i
9	$L(\mathbf{v}) = \sum_{i=0}^{n-1} \mathcal{B}_i \times \prod_{j=i+1}^{n-1} \left\lceil \frac{d_j}{b_j} \right\rceil$	Linearized block index
10	$L'(\mathbf{v}) = L(\mathbf{v}) \% \mathcal{T}$	Normalized linearized block index
11	$\mathcal{N}(\mathbf{v}) = \left\lfloor \frac{L'(\mathbf{v})}{t} \right\rfloor$	Node ID
12	$\mathcal{O}(\mathbf{v}) = L(\mathbf{v}) \% t$	Block offset within a node

this section and the expressions used by the locality analysis to compute the locality of array elements. The goal of the locality analysis is to compute symbolically the node ID of each shared reference in the loop and compare it to the node ID of the affinity expression. All references having a node ID equal to the affinity expression's node ID are local.

Locality analysis is done on the n-dimensional blocks of the multiblocked arrays present in the loop. For conventional UPC shared arrays declared with a blocking factor b, the analysis uses blocking factors of 1 in all dimensions except the last dimension, where b is used. The insight of the analysis is that a block shifted by a displacement vector $\mathbf{k}$ can span at most two threads along each dimension. Therefore locality can only change in one place in this dimension. We call this place the cut.

Once the cut is determined, our analysis tests the locality of the elements at the 2^n corners of the block. If a corner is found to be local, all the elements in the region from the corner up to the cuts in each dimension are also local.

Definition 1. *The value of a cut in dimension i, Cut_i, is the distance, measured in number of elements, between the corner of a block and the first transition between nodes on that dimension.*

Consider the two-level loop nest that accesses a two-dimensional blocked array shown in Figure 2. The layout of the array used in the loop is shown to the right of the code.

```
/* 8 threads and 2 threads/node */

shared [2][3] int A[8][8];

for(v0=0; v0<7 ; v0++){
   upc_forall(v1=0; v1<6 ; v1++; &A[v0][v1]){
      A[v0+1][v1+2] = v0*v1;
   }
}
```

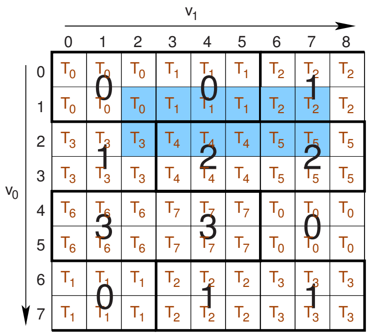

v1
v0
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7
T0 T0 T0 T1 T1 T1 T2 T2 T2
T3 T3 T3 T4 T4 T4 T5 T5 T5
T6 T6 T6 T7 T7 T7 T0 T0 T0
T1 T1 T1 T2 T2 T2 T3 T3 T3
0 0 1
1 2 2
3 3 0
0 1 1

Fig. 2. A two-dimensional array node example

Thin lines separate the elements of the array. Large bold numbers inside each block of 2×3 elements denote the node ID to which the block is mapped. Thick lines separate nodes from each other. The grey area in the array represents all elements that are referenced by iterations of the `forall` loop that are affine with `&A[0][0]`; cuts in this iteration space are determined by the thick lines (node boundaries).

Finding the cuts: In general, for dimensions $i = 0$ to $n - 2$ the value of the cut in that dimension is given by the following expression.

$$\mathrm{Cut}_i = b_i - k_i \% \, b_i \tag{5}$$

Thus in the example $\mathrm{Cut}_0 = 1$, which means that the only possible change of locality value happens between the first and second row of the block being accessed.

The cuts in the last dimension of the array are not necessarily the same for each corner. In Figure 2, for the top corners the cut is 4 but for the bottom corners the cut is 1. This happens when there are multiple colocated UPC threads in a node (in a hybrid setup); because the blocks in a node may "wrap around" the rows in the array.

Thus the analysis has to compute two separate values for the last cut: one for the upper corners and a second one for the lower corners. Upper and Lower refers to the last dimension in a multi-dimensional array. Let $\mathbf{k}' = [k_0 + b_0 - 1, k_1, \ldots, k_{n-1}]$. The expression for the last cut in the upper corner is as follows:

$$\mathrm{Cut}_{n-1}^{\mathrm{Upper}} = (t - \mathcal{O}(\mathbf{k})) \times b_{n-1} - k_{n-1} \% \, b_{n-1} \tag{6}$$

$$\mathrm{Cut}_{n-1}^{\mathrm{Lower}} = (t - \mathcal{O}(\mathbf{k}')) \times b_{n-1} - k_{n-1} \% \, b_{n-1} \tag{7}$$

where t is the number of threads per node.

When there is a single thread per node (*i.e.* $t = 1$), the normalized linearized block index is zero, and thus equations 6 and 7 simplify to equation 5.

Axiom 3.0.1. *Given an* `upc_forall` *loop with affinity test* $\mathrm{AffTest} = A(\mathbf{v})$ *and a shared array reference* $\mathrm{Ref} = A(\mathbf{v} + \mathbf{k})$, *this reference is local if and only if* $\mathcal{N}(\mathrm{AffTest}) = \mathcal{N}(\mathrm{Ref})$

Theorem 1. *Let A be an n-dimensional shared array with dimensions $d_0, d_1, \ldots, d_{n-1}$ and with blocking dimensions $b_0, b_1, \ldots, b_{n-1}$. Let $\mathbf{w} = v_0, v_1, \ldots, v_p, \ldots, v_{n-1}$ and $\mathbf{y} = v_0, v_1, \ldots, v_p + 1, \ldots v_{n-1}$ be two vectors such that $A(\mathbf{w})$ and $A(\mathbf{y})$ are elements of A. Let $B_{\text{Off}} = \mathcal{O}(v_0, v_1, \ldots, 0)$ be the block offset for the first element in the block in dimension $n-1$. Let*

$$v_i' = \begin{cases} v_i \% \, b_i - k_i \% \, b_i & \text{if} \quad i \neq n-1 \\ (v_i + B_{\text{Off}} \times b_i) \% \, (b_i \times t) & \text{Otherwise.} \end{cases} \tag{8}$$

if $v_p' \neq \text{Cut}_p - 1$ then $\mathcal{N}(\mathbf{w}) = \mathcal{N}(\mathbf{y})$.

Proof. We only present the proof for the case $p \neq n-1$ here. The proof for the case $p = n-1$ follows a similar reasoning but is more involved because it has to take into account the block offset for the first element in dimension $n-1$.

From the expressions in Table 1 the expression for the node id of elements $\mathbf{w}$ and $\mathbf{y}$ are given by:

$$\mathcal{N}(\mathbf{w}) = \left\lfloor \frac{L(\mathbf{w}) \% \, \mathcal{T}}{t} \right\rfloor \text{ and } \mathcal{N}(\mathbf{y}) = \left\lfloor \frac{L(\mathbf{y}) \% \, \mathcal{T}}{t} \right\rfloor \tag{9}$$

The linearized block index for $\mathbf{w}$ and $\mathbf{y}$ can be written as:

$$L(\mathbf{w}) = \sum_{i=0}^{n-1} \left\lfloor \frac{v_i}{b_i} \right\rfloor \times \prod_{j=i+1}^{n-1} \left\lceil \frac{d_j}{b_j} \right\rceil \tag{10}$$

$$L(\mathbf{y}) = L(\mathbf{w}) + \left(\left\lfloor \frac{v_p + 1}{b_p} \right\rfloor - \left\lfloor \frac{v_p}{b_p} \right\rfloor \right) \times \prod_{j=p+1}^{n-1} \left\lceil \frac{d_j}{b_j} \right\rceil \tag{11}$$

From equations 5 and 8:

$$v_p' = \text{Cut}_p - 1 \tag{12}$$

$$v_p \% \; b_p - k_p \% \; b_p = b_p - k_p \% \; b_p - 1 \tag{13}$$

From equation 13, the condition $v_p' \neq \text{Cut}_p - 1$ implies that $v_p \% \; b_p \neq b_p - 1$, which implies that $v_p \% \; b_p \leq b_p - 2$. Therefore:

$$\left\lfloor \frac{v_p + 1}{b_p} \right\rfloor = \left\lfloor \frac{v_p}{b_p} \right\rfloor \tag{14}$$

Substituting this result in equation 11 results that $L(\mathbf{y}) = L(\mathbf{w})$ and therefore $N(\mathbf{w}) = N(\mathbf{y})$.

Theorem 1 is the theoretical foundation of locality analysis based on corners and cuts. It establishes that the only place within a block where the node ID may change is at the cut. The key is that the elements $A(\mathbf{w})$ and $A(\mathbf{y})$ are adjacent elements of A.

4 Identifying Local Shared Accesses

In this section we present an algorithm that splits a loop nest into a number of smaller regions in the iteration space, such that in each region, each shared reference is known to be local or known to be remote. In a region, if a shared reference is determined to be local then the reference is privatized otherwise a call to the runtime is inserted.

To determine such regions, our analysis reasons about the positions of various shared references occuring in the loop nest relative to the affinity test expression. For each region, we keep track of a *position* relative to the affinity test shared reference. For each shared reference in the region, we also keep track of position of each reference relative to the region.

We start with the original loop nest as a single region. This region is analyzed and the cuts are computed. The region is then split according to the cuts generated. The new generated regions are again analyzed and split recursively until no more cuts are required. When all of the regions have been generated, we use the position of the region, and the position of the shared reference within the region to determine if it is local or remote. All shared references that are local are privatized. Figure 3 provides a sample

```
1 shared [5][5] int A[20][20];
2 int main() {
3   int i,j;
4   for (i=0; i < 19; i++)
5     upc_forall(j=0; j < 20; j++; &A[i][j]) {
6       A[i+1][j] = MYTHREAD;
7     }
8 }
```

Fig. 3. Example `upc_forall` loop containing a shared reference

loop nest containing a `upc_forall` loop and a shared array access. We will assume the example is compiled for a machine containing 2 nodes and will run with 8 UPC threads, creating a thread group size of 4. In this scenario, the shared array access on Line 6 will be local for the first four rows of every block owned by a thread T and remote for the remaining row. The LocalityAnalysis algorithm in Figure 4 begins by collecting all top-level loop nests that contain a candidate `upc_forall` loop. To be a candidate for locality analysis, a `upc_forall` loop must be normalized (lower bound begins at 0 and the increment is 1) and must use a pointer-to-shared argument for the affinity test. The algorithm then proceeds to analyze each loop nest independently (Step 2).

Phase 1 of the per-loopnest analysis algorithm finds and collects the `upc_forall` loop l_{forall}. The affinity statement used in l_{forall}, A_{stmt} is also obtained. Finally the CollectSharedReferences procedure collects all candidate shared references in the specified `upc_forall` loop. In order to be a candidate for locality analysis, a shared reference must have the same blocking factor as the shared reference used in the affinity test. The compiler must also be able to compute the *displacement vector* $k = ref_{shared} - affinityStatement$ for the shared reference, the vectorized difference between the indices of the reference and of the affinity statement.

In the example in Figure 3 the loop nest on Line 4 is collected as a candidate for locality analysis. The shared reference on Line 6 is collected as a candidate for locality analysis; the computed displacement vector is [1,0].

Phase 2 of the algorithm restructures the loop nest by splitting the iteration space of each loop into *regions* where the locality of shared references is known. Each region has

```
LocalityAnalysis($Procedure p$)
1. $NestSet \leftarrow$ GatherForallLoopNests($p$)
2. foreach loop nest $L$ in $NestSet$
Phase 1 - Gather Candidate Shared References
3.     $l_{forall} \leftarrow$ upc_forall loop found in loop nest $L$
4.     $nestDepth \leftarrow$ depth of $L$
5.     $A_{stmt} \leftarrow$ Affinity statement used in $l_{forall}$
6.     $SharedRefList \leftarrow$ CollectSharedReferences($l_{forall}, A_{stmt}$)
Phase 2 - Restructure Loop Nest
7.     $FirstRegion \leftarrow$ InitializeRegion($L$)
8.     $\mathcal{L}_R \leftarrow FirstRegion$
9.     while $\mathcal{L}_R$ not empty
10.        $R \leftarrow$ Pop head of $\mathcal{L}_{\mathcal{R}}$
11.        $CutList \leftarrow$ GenerateCutList($R, SharedRefList$)
12.        $nestLevel \leftarrow R.nestLevel$
13.        if $nestLevel < nestDepth - 1$
14.          $\mathcal{L}_R \leftarrow \mathcal{L}_R \cup$ GenerateNewRegions($R, CutList$)
15.        else
16.          $\mathcal{L}_R^{final} \leftarrow \mathcal{L}_R^{final} \cup$ GenerateNewRegions($R, CutList$)
17.        endif
18.    end while
Phase 3 - Identify Local Accesses and Privatize
19.    foreach $R$ in $\mathcal{L}_R^{final}$
20.        foreach $ref_{shared}$ in $SharedRefList$
21.            $refPosition \leftarrow$ComputePosition($ref_{shared}, R$)
22.            $nodeId \leftarrow$ ComputeNodeId($ref_{shared}, refPosition$)
23.            if $nodeId = 0$
24.              PrivatizeSharedReference($ref_{shared}$)
25. endfor
```

Fig. 4. Locality analysis for UPC shared references

a *statement list* associated with it, i.e. the lexicographically ordered list of statements as they appear in the program. Each region is also associated with a *position* in the iteration space of the loops containing the region.

In the example in Figure 3 the first region, R_0 contains the statements on Lines 5 to 7. The position of R_0 is 0, since the iteration space of the outermost loop contains the location 0. Once initialized, the region is placed into a list of regions, $\mathcal{L}_R$ (Step 8).

The algorithm iterates through all regions in $\mathcal{L}_R$. For each region, a list of cuts is computed based on the shared references collected in Phase 1. The cut represents the transition between a local access and a remote access in the given region. The GenerateCutList algorithm first determines the loop-variant induction variable iv in R that is used in ref_{shared}. The use of iv identifies the dimension in which to obtain the blocking factor and displacement when computing the cut. Depending on the dimension of the induction variable, either Equation 5 or Equations 6 and 7 are used to compute the cuts.

GENERATECUTLIST sorts all cuts in ascending order. Duplicate cuts and cuts outside the iteration space of the region ($Cut = 0$ or $Cut \geq b$) are discarded. Finally, the current region is cut into multiple iteration ranges, based on the cut list, using the GENERATENEWREGION algorithm. Newly created regions are separated by an if statement containing a *cut expression* of the form $iv\%b < Cut$ (the modulo is necessary since a cut is always in the middle of a block).

Step 13 determines if the region R is located in the innermost loop in the current loop nest (*i.e.* there are no other loops inside of R). If R contains innermost statements the regions generated by GENERATENEWREGIONS are placed in a separate list of final regions. $\mathcal{L}_R^{final}$. This ensures that at the end of Phase 2, the loop nest has been refactored into several iteration ranges and final statement lists (representing the innermost loops) are collected for use in Phase 3.

```
1  shared [5][5] int A[20][20];
2
3  int main() {
4    int i,j;
5    for (i=0; i < 19; i++)
6      if ((i % 5) < 4) {
7        upc_forall(j=0; j < 20; j++;
8                  &A[i][j]) {
9          A[i+1][j] = MYTHREAD;
10       }
11     }
12     else {
13       upc_forall(j=0; j < 20; j++;
14                 &A[i][j]) {
15         A[i+1][j] = MYTHREAD;
16       }
17     }
18 }
```

Fig. 5. Example after first cut

```
1  shared [5][5] int A[20][20];
2  int main() {
3    int i,j;
4    for (i=0; i < 19; i++)
5      if ((i % 5) < 4) {
6        upc_forall(j=0; j < 20; j++;
7                  &A[i][j]) {
8          offset = ComputeOffset(i,j);
9          base_A+offset = MYTHREAD;
10       }
11     }
12     else {
13       upc_forall(j=0; j < 20; j++;
14                 &A[i][j]) {
15         A[i+1][j] = MYTHREAD;
16       }
17     }
18 }
```

Fig. 6. Example after final code generation

The second phase iterates through the example in Figure 3 three times. The first region, R_0 and the $CutList = 4$, calculated by GENERATECUTLIST are passed in and the intermediate code shown in Figure 5 is generated. GENERATENEWREGIONS inserts the `if ((i \% 5) < 4)` branch and replicates the statements in region R_0. Two new regions, R_1 containing statements between lines 8 to 10 and R_2, containing lines 14 to 16, are created and added to the $NewList$. The respective positions associated with R_1 and R_2 are [0] and [4], respectively.

The new regions, R_1 and R_2 are popped off of the region list $\mathcal{L}_R$ in order. Neither region requires any cuts. GENERATENEWREGIONS copies R_1 and R_2 into R_3 and R_4 respectively. Since R_1 and R_2 represent the innermost loops in the nest, the new regions R_3 and R_4 will be placed into the final regions list (Step 16 in Figure 4). The position of region R_3 is [0,0] and the position of region R_4 is [4,0].

Phase 3 of the algorithm uses the position information stored in each of the final regions to compute the position of each shared reference in that region (Step 21). This information is then used to compute the node ID of the shared reference using the equations presented in Section 3 (Step 22). All shared references with a node ID of 0 are local and are privatized (Step 24). The shared reference $ref^{R_3}_{shared}$ located in R_3 is computed to have a position of $[1, 0]$ based on the position of R_3,$[0, 0]$, and the displacement vector of ref_{shared}, $[1, 0]$. The node ID for this position is 0 and thus $ref^{R_3}_{shared}$ is local. The shared reference $ref^{R_4}_{shared}$ is computed to have a position of $[5, 0]$ using the position for region R_4, $[4, 0]$. The node ID for this position is 1, and thus this reference is remote. Figure 6 shows the final code that is generated.

5 Experimental Evaluation

In this section we propose to evaluate the claims we have made in the paper: namely the usefulness of multiblocking and locality analysis. For our evaluation platform we used 4 nodes of an IBM Squadron^{TM}cluster. Each node has 8 SMP Power5 processors running at 1.9 GHz and 16 GBytes of memory.

Cholesky factorization and Matrix multiply: Cholesky factorization was written to showcase multi-blocked arrays. The tiled layout allows our implementation to take direct advantage of the ESSL [5] library. The code is patterned after the LAPACK [4] `dpotrf` implementation and adds up to 53 lines of text. To illustrate the compactness of the code, we reproduce one of the two subroutines used, distributed symmetric rank-k update, below.

```
void update_mb (shared double [B][B] A[N][N], int col0, int col1) {
  double a_local[B*B], b_local[B*B];
  upc_forall (int ii=col1; ii<N; ii+=B; continue)
    upc_forall (int jj=col1; jj<ii+B; jj+=B; &A[ii][jj]) {
      upc_memget (a_local, &A[ii][col0], sizeof(double)*B*B);
      upc_memget (b_local, &A[jj][col0], sizeof(double)*B*B);
      dgemm ("T", "N", &n, &m, &p, &alpha, b_local, &B, a_local,
              &B, &beta, (void *)&A[ii][jj], &B);
    }
}
```

The matrix multiply benchmark is written in a very similar fashion. It amounts to little more than a (serial) `k` loop around the `update` function above with slightly different loop bounds and three shared array arguments `A`, `B` and `C` instead of only one. It amounts to 20 lines of code. Without question, multiblocking allows compact code representation. The benchmark numbers presented in Figures 7 show mediocre scaling and performance “hiccups”, which we attribute to communication overhead and poor communication patterns. Clearly, multiblocking syntax needs to be extended with a distribution directive. Also, the UPC language could use better collective communication primitives; but that is in the scope of a future paper.

Dense matrix-vector multiplication: This benchmark multiplies a two-dimensional shared matrix with a one-dimensional shared vector and places the result in a one-dimensional shared vector. The objective of this benchmark is to measure the speed difference between compiler-privatized and unprivatized accesses.

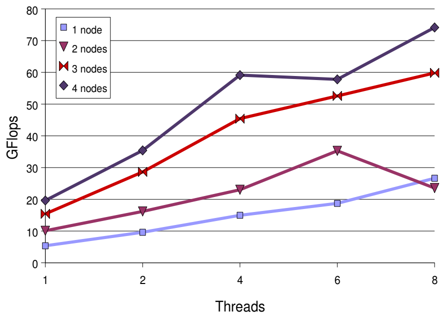
The matrix, declared of size 14400×14400, the vector as well the result vector are all blocked using single dimensional blocking. The blocking factors are equivalent to the [*] declarations. Since the vector is shared, the entire vector is first copied into a

Cholesky Performance (GFlops)

	1 node	2 nodes	3 nodes	4 nodes
1 TPN	5.37	10.11	15.43	19.63
2 TPN	9.62	16.19	28.64	35.41
4 TPN	14.98	23.03	45.43	59.14
6 TPN	18.73	35.29	52.57	57.8
8 TPN	26.65	23.55	59.83	74.14

Matrix Multiply Performance (GFlops)

	1 node	2 nodes	3 nodes	4 nodes
1 TPN	5.94	11.30	16.17	22.24
2 TPN	11.76	21.41	29.82	42.20
4 TPN	23.24	39.18	51.05	73.44
6 TPN	31.19	54.51	66.17	89.55
8 TPN	44.20	63.24	79.00	99.71



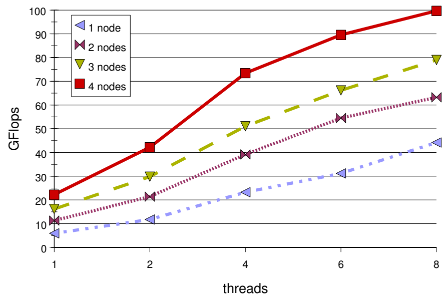


Fig. 7. Performance of multiblocked Cholesky and matrix multiply as a function of participating nodes and threads per node (TPN). Theoretical peak: 6.9 $GFlops \times threads \times nodes$.

Matrix-vector multiply

Naive	1 node	2 nodes	3 nodes	4 nodes
1 TPN	27.55	16.57	14.13	9.21
2 TPN	16.57	8.59	7.22	4.32
4 TPN	8.57	4.3	3.63	2.18
6 TPN	7.2	3.62	2.43	1.89
8 TPN	4.33	2.2	1.96	1.28

Opt.	1 node	2 nodes	3 nodes	4 nodes
1 TPN	2.08	1.22	0.78	0.6
2 TPN	1.7	0.85	0.63	0.43
4 TPN	0.85	0.44	0.33	0.23
6 TPN	0.65	0.35	0.25	0.19
8 TPN	0.44	0.23	0.22	0.17

Stencil benchmark

Naive	1 node	2 nodes	3 nodes	4 nodes
1 thread	35.64	24.59	19.04	13.41
2 threads	18.85	13.56	9.82	7.9
4 threads	9.8	13.64	5.58	8.9
6 threads	10.85	8.98	7.53	6.12
8 threads	4.9	5.58	9.52	3.66

Opt.	1 node	2 nodes	3 nodes	4 nodes
1 thread	0.30	1.10	1.41	0.74
2 threads	0.73	0.72	0.75	1.06
4 threads	0.44	1.19	0.39	0.84
6 threads	0.32	0.30	1.11	0.75
8 threads	0.22	0.63	1.07	1.02

Fig. 8. Runtime in seconds for the matrix-vector multiplication benchmark (left) and for the stencil benchmark (right). The tables on the top show naive execution times; the tables on the bottom reflect compiler-optimized runtimes.

local buffer using `upc_memget`. The matrix-vector multiplication itself is a simple 2 level nest with the outer loop being `upc_forall` . The address of the result vector element is used as the affinity test expression.

Results presented in Figure 8 (left side) confirm that compiler-privatized accesses are about an order of magnitude faster than unprivatized accesses.

5-point Stencil: This benchmark computes the average of a 4 immediate neighbors and the point itself at every point in a 2 dimensional matrix and stores the result in a different matrix of same size. The benchmark requires one original data matrix and one result matrix. 2-d blocking was used to maximize the locality. The matrix size used for the experiments was 5760×5760. Results, presented in Figure 8 (right side), show that in this case, too, run time is substantially reduced by privatization.

6 Related Work

There is a significant body of work on data distributions in the context of High Performance Fortran (HPF) and other data parallel languages. Numerous researchers have tackled the issue of optimizing communication on distributed memory architectures by either finding an appropriate distribution onto processors [1,9] or by determining a computation schedule that minimizes the number of message transfers [7,12]. By contrast to these works, we do not try to optimize the communication, but rather allow the programmer to specify at very high level an appropriate distribution and then eliminate the need for communication all together using compiler analysis. We do not attempt to restructure or improve the data placement of threads to processors in order to minimize communication. While these optimizations are certainly possible in our compiler, we leave them as future work.

The locality analysis presented in this paper is also similar to array privatization [13,11]. However, array privatization relies on the compiler to provide local copies and/or copy-in and copy-out semantics for all privatized elements. In our approach, once ownership is determined, private elements are directly accessed. In future work we will determine if there is sufficient reuse in UPC programs to overcome the cost of copying array elements into private memory.

Tiled and block distributions are useful for many linear algebra and scientific codes [2]. HPF-1 provided the ability to choose a data distribution independently in each dimesion if desired. Beside HPF, several other languages, such as ZPL [3] and X10 [15] provide them as standard distributions supported by the language. In addition, libraries such as the Hierarichical Tiled Arrays library [2] provide tiled distributions for data decomposition. ScaLAPACK [6], a widely used parallel library provides a 2 dimensional block-cyclic distribution for matrices which allows the placement of blocks over a 2-dimensional processor grid. The distribution used by ScaLAPACK is therefore more general than the distribution presented in the this paper.

7 Conclusions and Future Work

In this paper we presented a language extension for UPC shared arrays that provides fine control over array data layout. This extensions allows the programmer to obtain better performance while simplifying the expression of computations, in particular matrix computations. An added benefit is the ability to integrate existing libraries written in C and Fortran, which require specific memory layouts. We also presented a compile-time analysis and optimization of shared memory accesses. Using this analysis, the compiler is able to reduce the overheads introduced by the runtime system.

A number of issues still remain to be resolved, both in the UPC language and more importantly in our implementation. For multiblocked arrays, we believe that adding processor tiling will increase the programmer's ability to write codes that scale to large numbers of processors. Defining a set of collectives that are optimized for the UPC programming model will also address several scalability issues, such as the ones occuring in the LU Factorization and the High Performance Linpack kernel [8].

Our current compiler implementation suffers from several shortcomings. In particular, several loop optimizations are disabled in the presence of upc_forall loops. These limitations are reflected in the results presented in this paper, where the baseline C compiler offers a higher single thread performance compared to the UPC compiler.

Acknowledgements

This material is based upon work supported in part by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002. This work has also been supported by the Ministry of Education of Spain under contract TIN2007-60625. We also want to thank Philip Luk and Ettore Tiotto for their help with the IBM xlUPC compiler.

References

1. Ayguade, E., Garcia, J., Girones, M., Labarta, J., Torres, J., Valero, M.: Detecting and using affinity in an automatic data distribution tool. In: Languages and Compilers for Parallel Computing, pp. 61–75 (1994)
2. Bikshandi, G., Guo, J., Hoeflinger, D., Almási, G., Fraguela, B.B., Garzarán, M.J., Padua, D.A., von Praun, C.: Programming for parallelism and locality with hierarchically tiled arrays. In: PPOPP, pp. 48–57 (2006)
3. Chamberlain, B.L., Choi, S.-E., Lewis, E.C., Lin, C., Snyder, L., Weathersby, D.: ZPL: A machine independent programming language for parallel computers. Software Engineering 26(3), 197–211 (2000)
4. Dongarra, J.J., Du Croz, J., Hammarling, S., Hanson, R.J.: An extended set of FORTRAN Basic Linear Algebra Subprograms. ACM Transactions on Mathematical Software 14(1), 1–17 (1988)
5. ESSL User Guide,
   `http://www-03.ibm.com/systems/p/software/essl.html`
6. Blackford, L.S., et al.: ScaLAPACK: a linear algebra library for message-passing computers. In: Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing (Minneapolis, MN, 1997) (electronic), Philadelphia, PA, USA, p. 15. Society for Industrial and Applied Mathematics (1997)
7. Gupta, M., Schonberg, E., Srinivasan, H.: A unified framework for optimizing communication in data-parallel programs. IEEE Transactions on Parallel and Distributed Systems 7(7), 689–704 (1996)
8. HPL Algorithm description,
   `http://www.netlib.org/benchmark/hpl/algorithm.html`
9. Kremer, U.: Automatic data layout for distributed memory machines. Technical Report TR96-261, 14 (1996)

10. Numrich, R.W., Reid, J.: Co-array fortran for parallel programming. ACM Fortran Forum 17(2), 1–31 (1998)
11. Paek, Y., Navarro, A.G., Zapata, E.L., Padua, D.A.: Parallelization of benchmarks for scalable shared-memory multiprocessors. In: IEEE PACT, p. 401 (1998)
12. Ponnusamy, R., Saltz, J.H., Choudhary, A.N., Hwang, Y.-S., Fox, G.: Runtime support and compilation methods for user-specified irregular data distributions. IEEE Transactions on Parallel and Distributed Systems 6(8), 815–831 (1995)
13. Tu, P., Padua, D.A.: Automatic array privatization. In: Compiler Optimizations for Scalable Parallel Systems Languages, pp. 247–284 (2001)
14. UPC Language Specification, V1.2 (May 2005)
15. The X10 programming language (2004), `http://x10.sourceforge.net`
16. Yelick, K., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., Aiken, A.: Titanium: A high-performance java dialect. Concurrency: Practice and Experience 10(11-13) (September-November 1998)

An Experimental Evaluation of the New OpenMP Tasking Model

Eduard Ayguadé[1], Alejandro Duran[1], Jay Hoeflinger[2], Federico Massaioli[3], and Xavier Teruel[1]

[1] BSC-UPC
[2] Intel
[3] CASPUR

Abstract. The OpenMP standard was conceived to parallelize dense array-based applications, and it has achieved much success with that. Recently, a novel tasking proposal to handle unstructured parallelism in OpenMP has been submitted to the OpenMP 3.0 Language Committee. We tested its expressiveness and flexibility, using it to parallelize a number of examples from a variety of different application areas. Furthermore, we checked whether the model can be implemented efficiently, evaluating the performance of an experimental implementation of the tasking proposal on an SGI Altix 4700, and comparing it to the performance achieved with Intel's Workqueueing model and other worksharing alternatives currently available in OpenMP 2.5. We conclude that the new OpenMP tasks allow the expression of parallelism for a broad range of applications and that they will not hamper application performance.

1 Introduction

OpenMP grew out of the need to standardize the directive languages of several vendors in the 1990s. It was structured around parallel loops and was meant to handle dense numerical applications. The simplicity of its original interface, the use of a shared memory model, and the fact that the parallelism of a program is expressed in directives that are loosely-coupled to the code, all have helped OpenMP become well-accepted today. However, the sophistication of parallel programmers has grown in the last 10 years since OpenMP was introduced, and the complexity of their applications is increasing. Therefore, OpenMP is in the process of adding a tasking model to address this new programming landscape. The new directives allow the user to identify units of independent work, leaving the decisions of how and when to execute them to the runtime system.

In this paper, we have attempted to evaluate this new tasking model. We wanted to know how the new tasking model compared to traditional OpenMP worksharing and the existing Intel workqueueing model, both in terms of expressivity and performance. In order to evaluate expressivity, we have parallelized a number of problems across a wide range of application domains, using the tasking proposal. Performance evaluation has been done on a prototype implementation

```
#pragma omp parallel private (p)
{
  #pragma omp for
  for (i=0; i< n_lists; i++) {
    p = listheads[i];
    while(p) {
      #pragma omp task
        process(p)
      p=next(p);
    }
  }
}
```

Fig. 1. Parallel pointer chasing on multiple lists using `task`

```
void traverse(node *p, bool post)
{
  if (p->left)
    #pragma omp task
      traverse(p->left, post);
  if (p->right)
    #pragma omp task
      traverse(p->right, post);
  if (post) { /* postorder! */
    #pragma omp taskwait
  }
  process(p);
}
```

Fig. 2. Parallel depth-first tree traversal

of the tasking model. Performance results must be treated as preliminary, although we have validated the performance of our implementation against the performance of the commercial Intel workqueueing model implementation[1].

2 Motivation and Related Work

The task parallelism proposal under consideration by the OpenMP Language committee [2] gives programmers a way to express patterns of concurrency that do not match the worksharing constructs defined in the current OpenMP 2.5 specification.The proposal addresses common operations like complex, possibly recursive, data structure traversal, and situations which could easily cause load imbalance. The efficient parallelization of these algorithms using the 2.5 OpenMP standard is not impossible, but requires extensive program changes, such as run-time data structure transformations. This implies significant hand coding and run-time overhead, reducing the productivity that is typical of OpenMP programming[3].

Figure 1 illustrates the use of the new `omp task`[1] construct from the proposal. It creates a new flow of execution, corresponding to the construct's structured block. This flow of execution is concurrent to the rest of the work in the parallel region, but its execution can be performed only by a thread from the current team. Notice that this behavior is different from that of worksharing constructs, which are cooperatively executed by the existing team of threads. Execution of the task region does not necessarily start immediately, but can be deferred until the runtime schedules it.

The `p` pointer variable used inside the tasks in Figure 1 is implicitly determined *firstprivate*, i.e. copy constructed at task creation from the original copies used by each thread to iterate through the lists. This default was adopted in the proposal to balance performance, safety of use, and convenience for the programmer. It can be altered using the standard OpenMP data scoping clauses.

[1] This paper will express all code in C/C++, but the tasking proposal includes the equivalent directives in Fortran.

The new `#pragma omp taskwait` construct used in Figure 1 suspends the current execution flow until all tasks it generated have been completed. The semantics of the existing `barrier` construct is extended to synchronize for completion of all generated tasks in the team.

For a programming language extension to be successful, it has to be useful, and must be checked for expressiveness and productivity. Are the directives able to describe explicit concurrency in the problem? Do data scoping rules, defaults and clauses match the real programmers' needs? Do common use cases exist that the extension does not fulfill, forcing the programmer to add lines of code to fill the gap? The two examples above, while illustrative, involve very basic algorithms. They cannot be considered representative of a real application kernel.

In principle, the more concurrency that can be expressed in the source code, the more the compiler is able to deliver parallelism. However, factors like subtle side effects of data scoping, or even missing features, could hamper the actual level of parallelism which can be achieved at run-time. Moreover, parallelism *per se* does not automatically imply good performance. The semantics of a directive or clause can have unforeseen impact on object code or runtime overheads. In a language extension process, this aspect should also be checked thoroughly, with respect to the existing standard and to competing models.

The suitability of the current OpenMP standard to express irregular forms of parallelism was already investigated in the fields of dense linear algebra [4,5], adaptive mesh refinement [6], and agent-based models [7].

The Intel *workqueueing* model [8] was the first attempt to add dynamic task generation to OpenMP. The model, available as a proprietary extension in Intel compilers, allows hierarchical generation of tasks by the nesting of `taskq` constructs. Synchronization of descendant tasks is controlled by means of the default barrier at the end of `taskq` constructs. The implementation exhibits some overhead problems [7] and other performance issues [9].

In our choice of the application kernels to test drive the OpenMP tasking proposal, we were also inspired by the classification of different application domains proposed in [10], which addresses a much broader range of computations than traditional in the HPC field.

3 Programming with OpenMP Tasks

In this section we describe all the problems we have parallelized with the new task proposal. We have worked on applications across a wide range of domains (linear algebra, sparse algebra, servers, branch and bound, etc) to test the expressiveness of the proposal. Some of the applications (*multisort*, *fft* and *queens*) are originally from the Cilk project[11], some others (*pairwise alignment*, *connected components* and *floorplan*) come from the Application Kernel Matrix project from Cray[12] and two (*sparseLU* and *user interface*) have been developed by us. These kernels were not chosen because they were the best representatives of their class but because they represented a challenge for the current 2.5 OpenMP standard and were publicly available.

We have divided them into three categories. First were those applications that could already be easily parallelized with current OpenMP worksharing but where the use of tasks allows the expression of additional parallelism. Second were those applications which require the use of nested parallelism to be parallelized by the current standard. Nested parallelism is an optional feature and it is not always well supported. Third were those applications which would require a great amount of effort by the programmer to parallelize with OpenMP 2.5 (e.g. by programming their own tasks).

3.1 Worksharing Versus Tasking

SparseLU. The sparseLU kernel computes an LU matrix factorization. The matrix is organized in blocks that may not be allocated. Due to the sparseness of the matrix, a lot of imbalance exists. This is particularly true for the the *bmod phase* (see Figure 3). SparseLU can be parallelized with the current worksharing directives (using an OpenMP *for* with dynamic scheduling for loops on lines 10, 15 and 21 or 23). For the *bmod phase* we have two options: parallelize the outer (line 21) or the inner loop (line 23). If the outer loop is parallelized, the overhead is lower but the imbalance is greater. On the other hand, if the inner loop is parallelized the iterations are smaller which allows a dynamic schedule to have better balance but the overhead of the worksharing is much higher.

```
 1 int sparseLU() {
 2    int ii, jj, kk;
 3 #pragma omp parallel
 4 #pragma omp single nowait
 5    for (kk=0; kk<NB; kk++) {
 6       lu0(A[kk][kk]);
 7       /* fwd phase */
 8       for (jj=kk+1; jj<NB; jj++)
 9          if (A[kk][jj] != NULL)
10             #pragma omp task
11                fwd(A[kk][kk], A[kk][jj]);
12       /* bdiv phase */
13       for (ii=kk+1; ii<NB; ii++)
14          if (A[ii][kk] != NULL)
15             #pragma omp task
16                bdiv (A[kk][kk], A[ii][kk]);
17       #pragma omp taskwait
18       /* bmod phase */
19       for (ii=kk+1; ii<NB; ii++)
20          if (A[ii][kk] != NULL)
21             for (jj=kk+1; jj<NB; jj++)
22                if (A[kk][jj] != NULL)
23                #pragma omp task
24                {
25                   if (A[ii][jj]==NULL) A[ii][jj]=allocate_clean_block();
26                   bmod(A[ii][kk], A[kk][jj], A[ii][jj]);
27                }
28       #pragma omp taskwait
29    }
30 }
```

Fig. 3. Main code of SparseLU with OpenMP tasks

Using tasks, first we only create work for non-empty matrix blocks. We also create smaller units of work in the bmod phase with an overhead similar to the outer loop parallelization. This reduces the load imbalance problems.

It is interesting to note that, if the proposed extension included mechanisms to express dependencies among tasks, it would be possible to express additional parallelism that exists between tasks created in lines 12 and 17 and tasks created in line 25. Also it would be possible to express the parallelism that exists across consecutive iterations of the *kk* loop.

Protein pairwise alignment. This application aligns all protein sequences from an input file against every other sequence. The alignments are scored and the best score for each pair is output as a result. The scoring method is a full dynamic programming algorithm. It uses a weight matrix to score mismatches, and assigns penalties for opening and extending gaps. It uses the recursive Myers and Miller algorithm to align sequences.

The outermost loop can be parallelized, but the loop is heavily unbalanced, although this can be partially mitigated with dynamic scheduling. Another problem is that the number of iterations is too small to generate enough work when the number of threads is large. Also, the loops of the different passes (forward pass, reverse pass, diff and tracepath) can also be parallelized but this parallelization is much finer so it has higher overhead.

We used OpenMP tasks to exploit the inner loop in conjunction with the outer loop. Note that the tasks are nested inside an OpenMP *for* worksharing construct. This breaks iterations into smaller pieces, thus increasing the amount of parallel work but at lower cost than an inner loop parallelization because they can be excuted inmediately.

```
#pragma omp for
for (si = 0; si < nseqs; si++) {
   len1 = compute_sequence_length(si+1);

   /* compare to the other sequences */
   for (sj = si + 1; sj < nseqs; sj++) {
      #pragma omp task
      {
         len2 = compute_sequence_length(sj+1);
         compute_score_penalties(...);
         forward_pass(...);
         reverse_pass(...);
         diff(...);
         mm_score = tracepath(...);
         if (len1 == 0 || len2 == 0) mm_score  = 0.0;
         else                        mm_score /= (double) MIN(len1,len2);

         #pragma omp critical
        print_score();
      }
   }
}
```

Fig. 4. Main code of the pairwise aligment with tasks

3.2 Nested Parallelism Versus Tasking

Floorplan. The Floorplan kernel computes the optimal floorplan distribution of a number of cells. The algorithm is a recursive branch and bound algorithm. The parallelization is straight forward (see figure 5). We hierarchically generate tasks for each branch of the solution space. But this parallelization has one caveat. In these kind of algorithms (and others as well) the programmer needs to copy the partial solution up to the moment to the new parallel branches (i.e. tasks). Due to the nature of C arrays and pointers, the size of it becomes unknown across function calls and the data scoping clauses are unable to perform a copy on their own. To ensure that the original state does not disappear before it is copied, a task barrier is added at the end of the function. Other possible solutions would be to copy the array into the parent task stack and then capture its value or allocate it in heap memory and free it at the end of the child task. In all these solutions, the programmer must take special care.

Multisort, FFT and Strassen. Multisort is a variation of the ordinary mergesort. It sorts a random permutation of n 32-bit numbers with a fast parallel sorting algorithm by dividing an array of elements in half, sorting each half

```
void add_cell(int id, coor FOOTPRINT, ibrd BOARD, struct cell *CELLS) {
  int  i, j, nn, area;  ibrd board;  coor footprint, NWS[DMAX];

  for (i = 0; i < CELLS[id].n; i++) {
       nn = compute_possible_locations(id, i, NWS, CELLS);
/* for all possible locations */
       for (j = 0; j < nn; j++) {
#pragma omp task private(board, footprint,area) \
          shared(FOOTPRINT,BOARD,CELLS)
{           /* copy parent state*/
            struct cell cells[N+1];
            memcpy(cells,CELLS,sizeof(struct cell)*(N+1));
       memcpy(board, BOARD, sizeof(ibrd));

       compute_cell_extent(cells,id,NWS,j);

            /* if the cell cannot be layed down, prune search */
            if (! lay_down(id, board, cells)) {
                goto _end;
            }
       area = compute_new_footprint(footprint,FOOTPRINT,cells[id]);

       /* if last cell */
            if (cells[id].next == 0) {
                 if (area < MIN_AREA)
                 #pragma omp critical
                      if (area < MIN_AREA)  save_best_solution();
            } else if (area < MIN_AREA)
       /* only continue if area is smaller to best area, otherwise prune */
                 add_cell(cells[id].next, footprint, board,cells);
_end:;
}
       }
  }
  #pragma omp taskwait
}
```

Fig. 5. C code for the Floorplan kernel with OpenMP tasks

recursively, and then merging the sorted halves with a parallel divide-and-conquer method rather than the conventional serial merge. When the array is too small, a serial quicksort is used so the task granularity is not too small. To avoid the overhead of quicksort, an insertion sort is used for arrays below a threshold of 20 elements.

The parallelization with tasks is straight forward and makes use of a few `task` and `taskgroup` directives (see figure 6), the latter being the structured form of the `taskwait` construct introduced in section 2.

FFT computes the one-dimensional Fast Fourier Transform of a vector of n complex values using the Cooley-Tukey algorithm. Strassen's algorithm for multiplication of large dense matrices uses hierarchical decomposition of a matrix. The structure of the parallelization of these two kernels is almost identical to the one used in multisort, so we will omit it.

N Queens problem. This program, which uses a backtracking search algorithm, computes all solutions of the n-queens problem, whose objective is to find a placement for n queens on an n x n chessboard such that none of the queens attacks any other.

In this application, tasks are nested dynamically inside each other. As in the case of floorplan, the state needs to be copied into the newly created tasks so we need to introduce additional synchronizations (in the form of `taskgroup`) in order for the original state to be alive when the tasks start so they can copy it.

```
void sort(ELM *low, ELM *tmp, long size) {
    if (size < quick_size) {
     /* quicksort when reach size threshold */
     quicksort(low, low + size - 1);
     return;
    }
    quarter = size / 4;

    A = low; tmpA = tmp;
    B = A + quarter; tmpB = tmpA + quarter;
    C = B + quarter; tmpC = tmpB + quarter;
    D = C + quarter; tmpD = tmpC + quarter;

    #pragma omp taskgroup {
        #pragma omp task
      sort(A, tmpA, quarter);
        #pragma omp task
      sort(B, tmpB, quarter);
        #pragma omp task
      sort(C, tmpC, quarter);
        #pragma omp task
      sort(D, tmpD, size - 3 * quarter);
    }
    #pragma omp taskgroup {
        #pragma omp task
      merge(A, A+quarter-1, B, B+quarter-1, tmpA);
        #pragma omp task
      merge(C, C+quarter-1, D, low+size-1, tmpC);
    }
    merge(tmpA, tmpC-1, tmpC, tmpA+size-1, A);
}
```

Fig. 6. Sort function using OpenMP tasks

Another issue is the need to count all the solutions found by different tasks. One approach is to surround the accumulation with a critical directive but this would cause a lot of contention. To avoid it, we used `threadprivate` variables that are reduced within a `critical` directive to the global variable at the end of the parallel region.

Concom (Connected Components). The concom program finds all the connected components of a graph. It uses a depth first search starting from all the nodes of the graph. Every node visited is marked and not visited again.

The parallelization with tasks involves just four directives: a parallel directive, a single directive, a task directive and a critical directive. This is a clear example of how well tasks map into tree-like traversals.

3.3 Almost Impossible in OpenMP 2.5

Web server. We used tasks to parallelize a small web server called Boa. In this application, there is a lot of parallelism, as each client request to the server can be processed in parallel with minimal synchronizations (only update of log files and statistical counters). The unstructured nature of the requests makes it very difficult to parallelize without using tasks.

On the other hand, obtaining a parallel version with tasks requires just a handful of directives, as shown in figure 8. Basically, each time a request is ready, a new task is created for it.

```
void CC (int i, int cc) {
    int j, n;
    /* if node has not been visited */
    if (!visited[i]) {
        /* add node to current component */
        add_to_component(i, cc); /* omp critical inside */

        /* add each neighbor's subtree to the current component */
        for (j = 0; j < nodes[i].n; j++) {
       n = nodes[i].neighbor[j];
       #pragma omp task
           CC(n, cc);
        }
    }
}

void main () {
    init_graph();
    cc = 0;
   /* for all nodes ... unvisited nodes start a new component */
   for (i = 0; i < NN; i++)
      if (!visited[i]) {
    #pragma omp parallel
        #pragma omp single
            CC(i, cc);
         cc++;
      }

}
```

Fig. 7. Connected components code with OpenMP tasks

```
#pragma omp parallel
#pragma omp single nowait
while (!end) {
    process signals (if any)
    foreach request from the blocked queue {
        if ( request dependences are met ) {
            extract from the blocked queue
            #pragma omp task
               serve_request(request);
        }
    }
    if ( new connection ) {
        accept_it();
        #pragma omp task
            serve_request(new connection);
    }
    select();
}
```

Fig. 8. Boa webserver main loop with OpenMp tasks

The important performance metric for this application is response time. In the proposed OpenMP tasking model, threads are allowed to switch from the current task to a different one. This task switching is needed to avoid starvation, and prevent overload of internal runtime data structures when the number of generated tasks overwhelms the number of threads in the current team. The implementation is allowed to insert implicit switching points in a task region, wherever it finds appropriate. The `taskyield` construct inserts an explicit switching point, giving programmers full control. The experimental implementation we used in our tests is not aggressive in inserting implicit switching points. To improve the performance of the Web server, we inserted a `taskyield` construct inside the serve_request function so that no request is starved.

User Interface. We developed a small kernel that simulates the behavior of user interfaces (UI). In this application, the objective of using parallelism is to obtain a lower response time rather than higher performance (although, of course, higher performance never hurts). Our UI has three possible operations, which are common to most user interfaces: start some work unit, list current ongoing work units and their status, and cancel an existing work unit.

The work units map directly into tasks (as can be seen in Figure 9). The thread executing the `single` construct will keep executing it indefinitely. To be able to communicate between the interface and the work units, the programmer needs to add new data structures. We found it difficult to free these structures from within the task because it could easily lead to race conditions (e.g. free the structure while listing current work units). We decided to just mark them to be freed by the main thread when it knows that no tasks are using it. In practice, this might not always be possible and complex synchronizations may be needed.

We also used the `taskyield` directive to avoid starvation.

```
void Work::exec ( ) {
    while (!end) {
        //do some amount of work
        #pragma omp taskyield
    }
}

void start_work (...) {
    Work *work = new Work(...);
    list_of_works.push_back(work);
    #pragma omp task
    {
        work->exec();
        work->die();
    }
    gc();
}

void ui () {
    ...
    if ( user_input == START_WORK ) start_work(...);
}

void main ( int argc, char **argv ) {
     #pragma omp parallel
     #pragma omp single nowait
         ui();
}
```

Fig. 9. Simplified code for a user interface with OpenMP tasks

4 Evaluation

4.1 The Prototype Implementation

In order to test the proposal in terms of expressiveness and performance, we have developed our own implementation of the proposed tasking model. We developed the prototype on top of a research OpenMP compiler (source-to-source restructuring tool) and runtime infrastructure [13].

The implementation uses execution units, that are managed through different execution queues (usually one *global queue* and one *local queue* for each thread used by the application). The library offers different services (fork/join, synchronize, dependence control, environment queries, ...) that can provide the worksharing and structured parallelism expressed by the OpenMP 2.5 standard. We added several services to the library to give support to the task scheme. The most important change in the library was the offering of a new scope of execution that allows the execution of independent units of work that can be deferred, but still bound to the thread team (the concept of *task*, see section 2).

When the library finds a task directive, it is able to decide (according to internal parameters: *maximum depth level* in task hierarchy, *maximum number of tasks* or *maximum number of tasks by thread*) whether to execute it immediately or create a work unit that will be queued and managed through the runtime scheduler. This new feature is provided by adding a new set of queues:

team queues. The scheduler algorithm is modified in order to look for new work in the *local*, *team* and *global* queues respectively.

Once the task is first executed by a thread, and if the task has *suspend/resume* points, we can expect two different behaviors. First, the task could be bound to that thread (so, it can only be executed by that thread) and second, the task is not attached to any thread and can be executed by any other thread of the team. The library offers the possibility to move a task from the *team* queues to the *local* queues. This ability covers the requirements of the `untied` clause of the `task` construct, which allows a task suspended by one thread to be resumed by a different one.

The synchronization construct is provided through *task counters* that keep track of the number of tasks which were created in the current scope (the current scope can be a `task` or `taskgroup` construct). Each task has in its own structure with a *successor* field that points to the counter it must decrement.

4.2 Evaluation Methodology

We have already shown the flexibility of the new tasking proposal, but what about its performance? To determine this, we have evaluated the performance of the runtime prototype against other options.

We have run all the previous benchmarks but we do not include the results for the webserver (due to a lack of the proper network environment) and the simple-ui (because it has an interactive behavior). For each application we have tried each possible OpenMP version: a single level of parallelism (labeled OpenMP worksharing), multiple levels of parallelism (labeled OpenMP nested) and with OpenMP tasks. For those applications that could be parallelized with Intel's taskqueues, we also evaluated them with taskqueues.

Table 1 summarizes the different input parameters and the experiments run for each application.

Table 1. Input parameters for each application

Application	**Input parameters**	**Experiments**
strassen	Matrix size of 1280x1280	nested, tasks, taskqueues
multisort	Array of 32M of integers	nested, tasks, taskqueues
fft	Array of 32M of complex numbers	nested, tasks, taskqueues
queens	Size of the board is 14x14.	nested, tasks, taskqueues
alignment	100 sequences	worksharing, nested, tasks
floorplan	20 cells	nested, tasks, taskqueues
concom	500000 graph nodes, 100000 edges	nested, tasks, taskqueues
sparseLU	Sparse matrix of 50 blocks of 100x100	worksharing, nested, tasks, taskqueues

We compiled the codes with taskqueues and nested parellelism with Intel's icc compiler version 9.1 at the default optimization level. The versions using tasks use our OpenMP source-to-source compiler and runtime prototype implementation, using icc as the backend compiler. The speedup of all versions is computed, using as a baseline the serial version of each kernel. We used Intel's icc compiler to compile the serial version.

All the benchmarks have been evaluated on an SGI Altix 4700 with 128 processors, although they were run on a cpuset comprising a subset of the machine to avoid interference with other running applications.

4.3 Results

In figure 10 we show the speedup for all the kernels (except the *concom*) with the different evaluated versions: OpenMP worksharing, OpenMP nested, OpenMP tasks and Intel's taskqueues. We do not show the results of the *concom* kernel because the slowdowns prevented us from running the experiments due to time constraints. These slowdowns were not only affecting the OpenMP task version but also the OpenMP nested and Intel's taskqueues. The main reason behind the slowdown is granularity. The tasks (or parallel regions in the nested case) are so fine grained that it is impossible to scale without aggregrating them. That is something that currently none of the models supports.

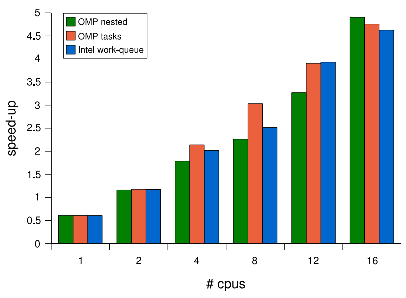
For a small number of threads (up to 4) we see that the versions using the new OpenMP tasks perform about the same as those using current OpenMP (worksharing and nested versions). But, as we increase the number of processors the task version scales much better, always improving over the other versions except for the *multisort* kernel, which has the same performance. These improvements are due to different factors, depending on the kernel: better load balance (*sparseLU*, *alignment*, *queens*, *fft*, *strassen* and *floorplan*), greater amount of parallel work (*alignment* and *sparseLU*) and less overhead (*alignment*). Overall, we can see that the new task proposal has the potential to benefit a wide range of application domains.

When we compare how the current prototype performs against a well established implementation of tasking, Intel's taskqueue, we can see that in most of the kernels the obtained speedup is almost the same and in a few cases (*sparseLU* and *floorplan*), even better. Only in two of them (*fft* and *strassen*) does taskqueue perform better, and even then, not by a large amount.

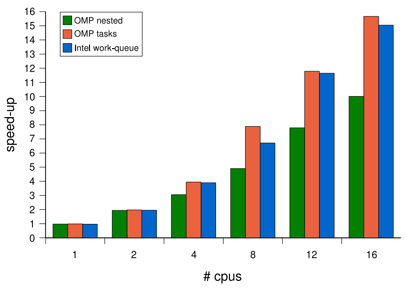
Taking into account that the prototype implementation has not been well tuned, we think that the results show that the new model will allow codes to obtain at least the performance of Intel's taskqueue and is even more flexible.

5 Suggestions for Future Work

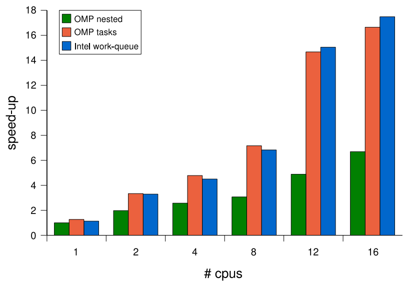
While the performance and flexibility of the new OpenMP tasking model seem good, there is still room for improvement. We offer these suggestions for ways to improve the usability and performance of the model, based on our experience with the applications described in this paper.



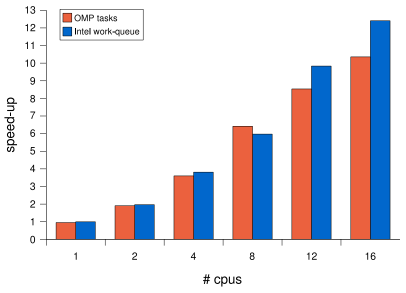
(a) Multisort evaluation



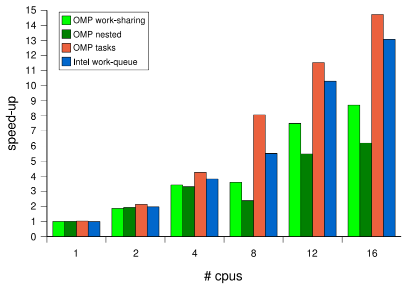
(b) N Queens evaluation



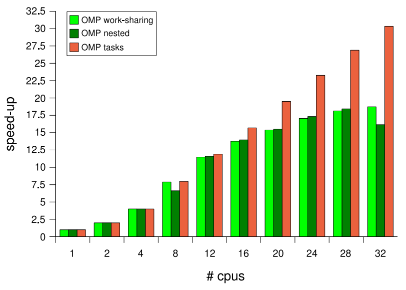
(c) FFT evaluation



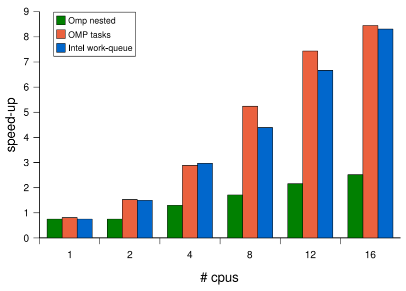
(d) Strassen evaluation



(e) SparseLU evaluation



(f) Alignment evaluation



(g) Floorplan evaluation

Fig. 10. Evaluation results for all the kernels. Speedups use serial version as baseline.

One problem we encountered consistently in our programming was the need to capture the value of a data structure when all we had was a pointer to it. If a pointer is used in a `firstprivate` directive, only the pointer is captured. In order to capture the data structure pointed-at, the user must program it by hand inside the task, including proper synchronization, to make sure that the data is not freed or popped off the stack before it is copied. Support for this in the language would improve the usability of the tasking model.

In the N Queens problem, we could have used a reduction operation for tasks. In other words, we could have used a way to automatically make tasks contribute values to a shared variable. It can be programmed explicitly using threadprivate variables, but a reduction clause would save programming effort.

The `taskgroup` and `taskwait` constructions provide useful task synchronization, but are cumbersome for programming some types of applications, such as a multi-stage pipeline. A pipeline could be implemented by giving names to tasks, and waiting for other tasks by name.

We anticipate much research in the area of improving the runtime library. One research direction that would surely yield improvements is working on the task scheduler, as it can significantly affect application performance. Another interesting idea would be to find the impact of granularity on application performance and develop ways, either explicitly or implicitly, to increase the granularity of the tasks (for example by aggregating them) so they could be applied to applications with finer parallelism (e.g. the connected components problem) or reduce the overhead in other applications.

Of course, we have not explored all possible application domains, so other issues may remain to be found. Therefore, it is important to continue the assesment of the proposal by looking at new applications and particularly at Fortran codes, where optimizations could be affected differently by the tasking model. Another interesting dimension to assess in the future is the point of view of novice programmers and their learning curve with the model.

6 Conclusions

This paper had two objectives: first, test the expressiveness of the new OpenMP tasks proposal. Second, verify that the model does not introduce hidden factors that hamper the actual level of parallelism which can be achieved at runtime.

We have shown that the new proposal allows the programmer to express the parallelism of a wide range of applications from very different domains (linear algebra, server applications, backtracking, etc). Furthermore, we have found different issues that OpenMP language designers may want to consider in the future to further improve the expressiveness of the language and simplify the programming effort in some scenarios.

Using these applications we have seen the new proposal matches other tasking proposals in terms of performance and that it surpasses alternative implementations with the current 2.5 OpenMP elements. While these results are not conclusive, as they certainly have not explored exhaustively all possibilities, they provide a strong indication that the model can be implemented without incurring significant overheads. We have also detected two areas where runtime improvements would benefit the applications (i.e. task scheduling and granularity).

In summary, we think that while the new OpenMP task proposal can be improved, it provides a solid basis for the development of applications containing irregular parallelism.

Acknowledgments

The Nanos group at BSC-UPC has been supported by the Ministry of Education of Spain under contract TIN2007-60625, and the European Commission in the context of the SARC integrated project #27648 (FP6).

References

1. Intel Corporation. Intel(R) C++ Compiler Documentation (May 2006)
2. Ayguadé, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Unnikrishnan, P., Zhang, G.: A Proposal for Task Parallelism in OpenMP. In: Chapman, B.M., Zheng, W., Gao, G.R., Sato, M., Ayguadé, E., Wang, D. (eds.) IWOMP 2007. LNCS, vol. 4935. Springer, Heidelberg (2008)
3. Hochstein, L., et al.: Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers. In: SuperComputing 2005 (November 2005)
4. Salvini, S.: Unlocking the Power of OpenMP. In: 5th European Workshop on OpenMP (EWOMP 2003) (September 2003) (Invited)
5. Kurzak, J., Dongarra, J.: Implementing Linear Algebra Routines on Multi-Core Processors with Pipelining and a Look Ahead. LAPACK Working Note 178, Dept. of Computer Science, University of Tennessee (September 2006)
6. Blikberg, R., Sørevik, T.: Load balancing and OpenMP implementation of nested parallelism. Parallel Computing 31(10-12), 984–998 (2005)
7. Massaioli, F., Castiglione, F., Bernaschi, M.: OpenMP parallelization of agent-based models. Parallel Computing 31(10-12), 1066–1081 (2005)
8. Shah, S., Haab, G., Petersen, P., Throop, J.: Flexible control structures for parallellism in OpenMP. In: 1st European Workshop on OpenMP (September 1999)
9. Van Zee, F.G., Bientinesi, P., Low, T.M., van de Geijn, R.A.: Scalable Parallelization of FLAME Code via the Workqueuing Model. ACM Trans. Math. Soft. (submitted, 2006)
10. Asanovic, K., et al.: The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Science Depts., University of California at Berkeley (December 2006)
11. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. In: PLDI 1998: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation, pp. 212–223. ACM Press, New York (1998)
12. Chamberlain, B., Feo, J., Lewis, J., Mizell, D.: An application kernel matrix for studying the productivity of parallel programming languages. In: W3S Workshop - 26th International Conference on Software Engineering, pp. 37–41 (May 2004)
13. Balart, J., Duran, A., Gonzàlez, M., Martorell, X., Ayguadé, E., Labarta, J.: Nanos mercurium: a research compiler for openmp. In: Proceedings of the European Workshop on OpenMP 2004 (October 2004)

Language Extensions in Support of Compiler Parallelization

Jun Shirako[1,2], Hironori Kasahara[1,3], and Vivek Sarkar[4]

[1] Dept. of Computer Science, Waseda University
[2] Japan Society for the Promotion of Science, Research Fellow
[3] Advanced Chip Multiprocessor Research Institute, Waseda University
[4] Department of Computer Science, Rice University
{shirako,kasahara}@oscar.elec.waseda.ac.jp, vsarkar@rice.edu

Abstract. In this paper, we propose an approach to automatic compiler parallelization based on language extensions that is applicable to a broader range of program structures and application domains than in past work. As a complement to ongoing work on high productivity languages for explicit parallelism, the basic idea in this paper is to make sequential languages more amenable to compiler parallelization by adding enforceable declarations and annotations. Specifically, we propose the addition of annotations and declarations related to multidimensional arrays, points, regions, array views, parameter intents, array and object privatization, pure methods, absence of exceptions, and gather/reduce computations. In many cases, these extensions are also motivated by best practices in software engineering, and can also contribute to performance improvements in sequential code. A detailed case study of the Java Grande Forum benchmark suite illustrates the obstacles to compiler parallelization in current object-oriented languages, and shows that the extensions proposed in this paper can be effective in enabling compiler parallelization. The results in this paper motivate future work on building an automatically parallelizing compiler for the language extensions proposed in this paper.

1 Introduction

It is now well established that parallel computing is moving into the mainstream with a rapid increase in the adoption of multicore processors. Unlike previous generations of mainstream hardware evolution, this shift will have a major impact on existing and future software. A highly desirable solution to the multicore software productivity problem is to automatically parallelize sequential programs. Past work on automatic parallelization has focused on Fortran and C programs with a large body of work on data dependence tests [1,25,18,9] and research compilers such as Polaris [7,19], SUIF [10], PTRAN [21] and the D System [12]. However, it is widely acknowledged that these techniques have limited effectiveness for programs written in modern object-oriented languages such as Java.

In this paper, we propose an approach to compiler parallelization based on language extensions that is applicable to a broader range of program structures and application domains than in past work. As a complement to ongoing work on high productivity languages for explicit parallelism, the basic idea in this paper is to make sequential languages more amenable to compiler parallelization by adding enforceable declarations and annotations. In many cases, these extensions are also motivated by best practices in software engineering, and can also contribute to performance improvements in sequential code.

A detailed case study of the Java Grande Forum benchmarks [22,13] confirms that the extensions proposed in this paper can be effective in enabling compiler parallelization. Experimental results were obtained on a 16-way Power6 SMP to compare the performance of four versions of each benchmark: 1) sequential Java, 2) sequential X10, 3) hand-parallelized X10, 4) parallel Java. Averaged over ten JGF Section 2 and 3 benchmarks, the parallel X10 version was 11.9× faster than the sequential X10 version, which in turn was 1.2× faster than the sequential Java version (Figure 1). An important side benefit of the annotations used for parallelization is that they can also speed up code due to elimination of runtime checks. For the eight benchmarks for which parallel Java versions were available, the parallel Java version was an average of 1.3× faster than the parallel X10 version (Figure 2). However, for two of the eight benchmarks, the parallel Java version used a different algorithm from the sequential Java version, and resulted in super-linear speedups. When the sequential and parallel X10 versions for the two benchmarks were modified to be consistent with the new algorithms, the parallel Java and X10 versions delivered the same performance on average (Figure 3).

The rest of the paper is organized as follows. Section 2 describes the language extensions (annotations and declarations) proposed in this paper. Section 3 summarizes the results of the case study including experimental results, and Section 5 contains our conclusions.

2 Language Extensions

While modern object-oriented languages such as Java have improved programming productivity and code reuse through extensive use of object encapsulation and exceptions, these same features have made it more challenging for automatically parallelizing compilers relative to Fortran programs where data structures and control flow are more statically predictable. In this section, we propose a set of declarations and annotations that enable compilers to perform automatic parallelization more effectively for these languages. Unlike annotations that explicitly manage parallelism as in OpenMP [6], our approach is geared toward enforceable declarations and annotations that can be expressed and understood in the context of sequential programs, and that should be useful from a software engineering viewpoint because of their ability to reduce common programming errors. Another difference from OpenMP is that the correctness of all our proposed annotations and declarations is enforced by the language system *i.e.,* they are all checked statically or dynamically, as outlined below.

2.1 Multidimensional Arrays, Regions, Points

Multidimensional arrays in Java are defined and implemented as nested unidimensional arrays. While this provides many conveniences for guaranteeing safety in a virtual machine environment (e.g., subarrays can be passed as parameters without exposing any unsafe pointer arithmetic), it also creates several obstacles to compiler optimization and parallelization. For example, a compiler cannot automatically conclude that `A[i][j]` and `A[i+1][j]` refer to distinct locations since the nested array model allows for the possibility that `A[i]` and `A[i+1]` point to the same subarray. Instead, we propose the use of object-oriented multidimensional arrays as in X10 [3], in which a compiler is guaranteed that `A[i,j]` and `A[i+1,j]` refer to distinct locations. *Array Views* (Section 2.2) make it possible to safely work with subarrays of multidimensional arrays without introducing unsafe pointer arithmetic.

A related issue is that induction variable analysis can be challenging in cases when an iterator is used or an integer variable is incremented by a step value that is not a compile-time constant as illustrated in the following common idiom from a DAXPY-like computation:

```
iy = 0; if (incy < 0) iy = (-n+1)*incy;
for (i = 0;i < n; i++) {
  dy[iy +dy_off] += . . .;  iy += incy;
}
```

In the above example, it is not easy for compilers to establish that `incy` $\neq 0$ and that there are no loop-carried dependences on the `dy` array.

To simplify analysis in such cases, we recommend the use of *regions* and *points* as proposed in ZPL [23] and X10, with extensions to support two kinds of region constructors based on triple notation, `[<start-expr> : <end-expr> : <step-expr>]` and `[<start-expr> ; <count-expr> ; <step-expr>]`, both of which are defined to throw a ZeroStepException if invoked with a zero-valued step expression. The use of high level regions and points distinguishes our approach from past work on annotations of arrays for safe parallelization [16].

A key property of regions and points is that they can be used to define both loops and arrays in a program. The above DAXPY-like example can then be rewritten as follows:

```
iy = 0; if (incy < 0) iy = (-n+1)*incy;
// Example of [<start-expr>;<count-expr>;<step-expr>] region
for (point p : [iy ; n ; incy] ) {
  dy[p] += . . .;
}
```

In this case, the compiler will know that `incy` $\neq 0$ when the loop is executed, and that all `dy[p]` accesses are distinct.

2.2 Array Views

As indicated in the previous section, it is easier for a compiler to parallelize code written with multidimensional arrays rather than nested arrays. However, this raises the need for the programmer to work with subarrays of multidimensional arrays without resorting to unsafe pointer arithmetic. Our solution is the use of *array views.* An array view can be created by invoking a standard library method, `view( <start-point-expr>, <region-expr>)`, on any array expression (which itself may be a view). Consider the following code fragment with array views:

```
// Allocate a two-dimensional M*N array
double[.] A = new double[[1:M,1:N]];
. . .
A[i,j] = 99;
. . .
// Allocate a one-dimensional view on A for row i
double[.] R = A.view([i,1], [1:N]);
. . .
temp = R[j]; // R[j] = 99, the value stored in A[i,j]
```

In the above example, `R` can be used like any one-dimensional array but accesses to `R` are aliased with accesses to `A` as specified by the region in the call to `A.view()`. A `ViewOutOfBoundsException` is thrown if a view cannot be created with the specified point and region. All accesses to `R` can only be performed with points (subscripts) that belong to the region specified when creating the view.

Views can also be created with an optional *intent* parameter that must have a value from a standard *enum*, {`In, Out, InOut`}. The default value is `InOut` which indicates that the view can be used to read and write array elements. `In` and `Out` intents are used to specify read-only and write-only constraints on the array views. Read-only views can be very helpful in simplifying compiler parallelization and optimization by identifying heap locations that are guaranteed to be immutable for some subset of the program's lifetime [17]. The runtime system guarantees that each array element has the same intent in all views containing the element. If an attempt is made to create a view that conflicts with the intent specified by a previous view, then a `ViewIntentException` is thrown.

2.3 Annotations on Method Parameters

We propose the use of a `disjoint` annotation to assert that all mutable (non-value) reference parameters in a method must be disjoint. (The *this* pointer is also treated as a parameter in the definition of the `disjoint` annotation.) If a disjoint method is called with two actual parameters that overlap, a `ParameterOverlap-Exception` is thrown at runtime. Declaring a method as disjoint can help optimization and parallelization of code within the method by assisting the compiler's alias analysis. This benefit comes at the cost of runtime tests that the compiler must insert on method entry, though the cost will be less in a strongly typed language like Java or X10 compared to a weakly typed language like C since runtime tests are not needed for parameters with non-compatible types in X10 but would be necessary in C due to its pointer addressing and cast operators. This is also why we expect it to be more effective for X10 than the `noalias` and `restricted` proposals that have been made in the past for `C`.

In addition to the `disjoint` annotation, we also propose the use of `in`, `out`, and `inout` intent annotations on method parameters as in Fortran. For object/array references, these annotations apply only to the object/array that is the immediate target of the reference.

2.4 Array and Object Privatization

It is well known that *privatization analysis* is a key enabling technique for compiler parallelization. For modern object-oriented languages with dynamically allocated objects and arrays, the effectiveness of privatization analysis is often bounded by the effectiveness of *escape analysis* [4]. We propose a `retained` type modifier[1] for declarations of local variables and parameters with reference types which asserts that the scope in which the local/parameter is declared will not cause any reference in a retained variable to escape. We also permit the `retained` modifier on declarations of methods with a non-value reference return type, in which case it ensures that the `this` pointer does not escape the method invocation.

The following loop from the MonteCarlo benchmark illustrates the use of the retained modifier to declare that each `ps` object is private to a single loop iteration.

```
results = new Vector(nRunsMC);
for( int iRun=0; iRun < nRunsMC; iRun++ ) {
    // ps object is local to a single loop iteration
    retained PriceStock ps = new PriceStock();
    // All methods invoked on ps must be declared as "retained"
    ps.setInitAllTasks((ToInitAllTasks) initAllTasks);
    ps.setTask((x10.lang.Object) tasks.elementAt(iRun));
    ps.run();
    results.addElement(ps.getResult());
} // for
```

To enable automatic parallelization, the compiler will also need information that indicates that results.addElement() is a reduction-style operator (associative and commutative). We discuss later in Section 2.7 how this information can be communicated using a `gather` clause.

2.5 Pure Annotation for Side-Effect-Free Methods

The return value (or exception value) and all parameters of a method annotated as `pure` must have *value types i.e.,* they must be immutable after initialization. Pure methods can call other pure methods and only allocate/read/write mutable heap locations whose lifetimes are contained within the method's lifetime (as defined with the `retained` type modifier). Therefore, if two calls are made to the same pure method with the same value parameters, they are guaranteed

[1] The `retained` name chosen because other candidates like “private” and “local” are overloaded with other meanings in Java.

to result in the same return value (or exception value). The only situation in which the two calls may not have the same outcome is if one of the calls triggers a nonfunctional error such as `OutOfMemoryError`. This definition of method purity is similar to the definition of "moderately pure" methods in [26]. The correctness of all `pure` annotations is enforced statically in our proposed approach, analogous to the static enforcement of immutability of value types in the X10 language [3].

2.6 Annotations Related to Exceptions

We propose the following set of declarations and annotations that can be used to establish the absence of runtime exceptions. All type declarations are assumed to be checked statically, but dynamic cast operations can be used to support type conversion with runtime checks. Some of the type declarations are based on the theory of *dependent types* (*e.g.,* see [11]) as embodied in version 1.01 of the X10 language [20].

- **Null Pointer exceptions:** A simple way to guarantee the absence of a NullPointerException for a specific operation is to declare the type of the underlying object/array reference to be *non-null*. As an example, the Java language permits null-valued references by default, with a proposal in JSR 305 [15] to introduce an @NonNull annotation to declare selected references as non-null. In contrast, the X10 language requires that all references be non-null by default and provides a special *nullable* type constructor that can be applied to any reference type. Though the results in our paper can be used with either default convention, we will use the X10 approach in all examples in this paper.
- **Array Index Out of Bounds exceptions:** A simple way to guarantee the absence of an `IndexOutOfBoundsExecption` for an array access is to ensure that the array access is performed in a loop that is defined to iterate over the array's region *e.g.,*

  ```
  for (point p : A.region) A[p] = ... ; //Iterate over A.region
  ```

  This idea can be extended by iterating over a region that is guaranteed to be a subset of the array's region, as in the following example (assuming `&&` represents region intersection):

  ```
  // Iterate over a subset of A.region
  for (point p : A.region && region2) A[p] = ... ;
  ```

  When working with multiple arrays, dependent types can be use to establish that multiple arrays have the same underlying region *e.g.,*

  ```
  final region R1 = ...;
  // A and B can only point to arrays with region = R1
  final double[:region=R1] A = ...;
  final double[:region=R1] B = ...;
  for (point p : R1 ) A[p] = F(B[p]) ; // F is a pure method
  ```

In the above example, the compiler knows from the dependent type declarations (and from the fact that the loop iterates over region `R1`) that array accesses `A[p]` and `B[p]` cannot throw an exception.

Dependent types can also be used on point declarations to ensure the absence of `IndexOutOfBoundsException`'s as in the access to `A[p]` in the following example:

```
final region R1 = ...;
final double[:region=R1] A = ...;
// p can only take values in region R1
point(:region=R1) p = ...;
double d = A[p];
```

– **Zero Divide/Step exceptions:** A simple way to guarantee the absence of a DivideByZeroException or a ZeroStepException for a specific operation is to declare the type of the underlying integer expression to be *nonzero* using dependent types as follows:

```
int(:nonzero) n = ...; // n's value must be nonzero
int q = m / n; // No DivideByZeroException
region R3 = [low : high : n]; // No ZeroStepException
```

– **ExceptionFree annotation:** A code region annotated as `ExceptionFree` is guaranteed to not throw any user-defined or runtime exception. As with `pure` methods, it is possible that a region of code annotated as `ExceptionFree` may encounter a nonfunctional error such as an `OutOfMemoryError`. The compiler checks all operations in the annotated code region to ensure that they are statically guaranteed to not throw an exception (by using the declarations and annotations outlined above).

2.7 Gather Computations and Reductions in Loops

A common requirement in parallel programs is the ability to either *gather* or *reduce* values generated in each loop iteration into (respectively) a collection or aggregate value. There has been a large body of past work on compiler analyses for automatically detecting gather and reduction idioms in loops and arrays *e.g.,* [14,8], but the presence of potentially aliased objects and large numbers of virtual calls render these techniques ineffective for object-oriented programs. Instead, we propose an extension to counted pointwise `for` loops that enables the programmer to specify the *gather* and *reduce* operators explicitly in a sequential program in a way that simplifies the compiler's task of automatic parallelization. Specifically, we extend the `for` loop with an optional `gather` clause as follows:

```
for ( ... ) { <body-stmts> gather <gather-stmt> }
```

A unique capability of the gather statement is that it is permitted to read private (`retained`) variables in the loop body that have primitive or value types, including the index/point variables that control the execution of the counted `for` loop. The design of `gather` clause is similar to the `inlet` feature in Cilk [24], which represents a post-processing of each parallel thread. Informally, the semantics of a `for` loop with a `gather` clause can be summarized as follows:

1. Identify `retained` variables in `<body-stmt>` with primitive or value types that are also accessed in `<gather-stmt>`. We refer to these as *gather* variables.
2. Execute all iterations of the `for` loop sequentially as usual, but store the values of *all* gather variables at the end of each iteration.
3. Execute `<gather-stmt>` once for each iteration of the `for` loop in a nondeterministic order (analogous to the nondeterminism inherent in iterating over an unordered collection in Java).
4. During execution of an instance of `<gather-stmt>` in Step 3, resolve read accesses to gather variables by returning the corresponding instances stored in Step 2.

We use the loop from the MonteCarlo benchmark discussed earlier to illustrate the use of the gather clause to specify the gather statement:

```
results = new Vector(nRunsMC);
for( point p[iRun] : [0 : nRunsMC-1] ) {
    // ps object is local to a single loop iteration
    retained PriceStock ps = new PriceStock();
    // All methods invoked on ps must be declared as "retained"
    ps.setInitAllTasks((ToInitAllTasks) initAllTasks);
    ps.setTask((x10.lang.Object) tasks.elementAt(iRun));
    ps.run();
    retained ToResult R = ps.getResult(); // must be a value type
    gather {
       // Invoked once for each iteration of the for loop
       results.addElement(R));
    }
}
```

3 Case Study: Java Grande Forum Benchmarks

In this section, we present the results of a case study that we undertook to validate the utility of the language annotations and extensions introduced in the previous section for compiler parallelization. The case study was undertaken on the Java Grande Forum (JGF) benchmark suite [22] because this suite includes both sequential and parallel (multithreaded) Java versions of the same benchmarks. We converted the sequential Java versions into sequential X10 versions, and then studied which annotations were necessary to enable automatic parallelization. A summary of the results can be found in Table 1, with discussion of the LUFact and Euler benchmarks in the following subsections. Performance results for sequential and parallel versions of the Java and X10 programs are presented later in Section 4.

3.1 LUFact

This benchmark solves a $N \times N$ linear system using LU factorization followed by a triangular solve. The kernel computation of the sequential Java version is as follows:

Table 1. Annotations required to enable parallelization of Java Grande Forum benchmarks

	Series	Sparse*	SOR	Crypt	LUFact	FFT	Euler	MolDyn	Ray*	Monte*
Multi-dim arrays	×		×		×		×			
Regions, Points	×	×	×		×		×	×		
Array views		×			×					
In/Out/InOut					×					
Disjoint		×		×			×			
Retained							×		×	×
Pure method	×					×				
NonNull	×	×	×	×	×	×	×	×	×	×
Region Dep-type		×		×	×	×			×	
Nonzero						×				
Exception free	×				×	×		×	×	×
Reduction		×						×	×	×

* Sparse: SparseMatmult, Ray: RayTracer, Monte: MonteCarlo

```
for (k = 0; k < nm1; k++) {
  col_k = a[k];
  l = idamax(n-k, col_k, k, 1) + k;
  ...
  for (j = kp1; j < n; j++) {
    col_j = a[j];
    t = col_j[l];
    if (l != k) {  col_j[l] = col_j[k];  col_j[k] = t;  }
    daxpy(n-(kp1), t, col_k, kp1, 1, col_j, kp1, 1);
  }
}
```

It is well known that all iterations of the inner j-loop can logically be executed in parallel, however there are numerous obstacles that make it challenging or intractable for a parallelizing compiler to discover this fact automatically. First, most compilers will have to conservatively assume that references to `col_j` from distinct iterations could potentially be aliased to the same subarray. Second, it will be hard for a compiler to statically establish that no array element access in the loop will throw an ArrayIndexOutOfBoundsException, especially the `col_j[l]` access with subscript `l` that is the return value of the call to function `idamax`. Third, a compiler will need to establish that the call to `daxpy` will not inhibit parallel execution of the `j` loop.

Now, consider the scenario in which the sequential code is written as follows using some of the language extensions proposed in this paper:

```
for (point k : [0:nm1-1] && a.region.rank(0) && a.region.rank(1)) {
  final double[.] col_k = a.view([k,0], [0:nm-1], IN);
  point (:region=a.region.rank(1)) l =
    (point (:region=a.region.rank(1))) idamax(n-k, col_k, k, 1) + k;
  ...
  for (point j : [kp1:n-1] && a.region.rank(0)) {
    final double[.] col_j = a.view([j,0], [0:nm-1], OUT);
    t = a[j, l];
    if (l != k) {  a[j, l] = a[j, k];  a[j, k] = t;  }
    daxpy(n-(kp1), t, col_k, kp1, 1, col_j, kp1, 1);
  }
}
```

As indicated in Table 1, the following annotations are sufficient to enable compiler parallelization for the LUFact benchmark:

- **Multi-dimensional arrays, Regions and Points, Array views:** In this example, array `a` is allocated as a two-dimensional array, and the use of multidimensional array views ensures that references to `col_j` from distinct iterations are guaranteed to point to distinct subarrays.
- **In/Out intents:** The use of an `IN` intent for `col_k` and an `OUT` intent for `col_j` ensures that accesses to the two subarrays will not inhibit parallelism.
- **NonNull:** Unlike Java, the default in X10 is that all object references are non-null by default, thereby ensuring that NullPointerException's cannot inhibit parallelism in this loop.
- **Region dependent types:** The use of a dependent type with a region constraint in the declaration of variable `l` ensures that all uses of `l` as a subscript in the second dimension (dimension 1) of array `a` must be in bounds — the cast operator effectively serves as a runtime check on the return value from function `idamax`.
- **Exception free:** Finally, an exception-free annotation on the `daxpy` method declaration (not shown above) assists the compiler in establishing that no exceptions can inhibit parallelization of the `j` loop.

With these extensions, it becomes entirely tractable for a compiler to automatically determine that iterations of the `j` loop can be executed in parallel.

3.2 Euler

The Euler benchmark solves a set of equations using a fourth order Runge Kutta method. It has many loops that can only be parallelized if the compiler knows that certain objects being accessed are private to each loop iteration. For example, consider the following `i` loop in method `calculateDummyCells`:

```
  private void calculateDummyCells(double localpg[][],
      double localtg[][], Statevector localug[][]) {  ...
    Vector2 tan = new Vector2();
    ...
    for (i = 1; i < imax; ++i) {
      tan.ihat = xnode[i][0] - xnode[i-1][0];
      tan.jhat = ynode[i][0] - ynode[i-1][0];
      ...  scrap = tan.magnitude();  ...
    }
    ...
  }
```

In the sequential version, a single instance of the `tan` object is allocated and reused across all iterations of the `i` loop. However, a closer examination reveals that each iteration could use a private copy of the `tan` object, thereby removing one of the obstacles to parallelization of the `i` loop.

We now consider the following alternate sequential version written using some of the language extensions proposed in this paper:

```
private disjoint void calculateDummyCells(double[.] localpg,
    double[.] localtg, Statevector[.] localug) {  ...
  for (point i : [1:imax-1] && xnode.region.rank(1) && ...) {
    retained Vector2 tan = new Vector2();
    tan.ihat = xnode[i, 0] - xnode[i-1, 0];
    tan.jhat = ynode[i, 0] - ynode[i-1, 0];
    ...  scrap = tan.magnitude();  ...
  }
   ...
}
```

As indicated in Table 1, the following annotations are sufficient to enable compiler parallelization for the Euler benchmark:

– **Multi-dimensional arrays, Regions and Points:** As with LUFact, the use of multidimensional arrays, regions and points enables a compiler to ensure that distinct iterations of the `i` loop are guaranteed to access distinct subarrays of `xnode` and `ynode` without ArrayIndexOutOfBoundsException.
– **Disjoint:** The `disjoint` annotation on method `calculateDummyCells` ensures that references `localpg` and `localtg` must point to distinct arrays.
– **Retained:** The `retained` annotation on the declaration of variable `tan` can be used by the compiler to determine that there are no loop-carried dependences on that variable.
– **NonNull:** As with LUFact, the fact that all object references are non-null by default ensures that NullPointerException's cannot inhibit parallelism in this loop.

4 Experimental Results

We then compared the performance of four versions of the Java Grande Forum (JGF) benchmarks:

1. **Sequential Java:** This set consists of six Section 2 benchmarks (Crypt, FFT, LUFact, Series, SOR, SparseMatmult) and four Section 3 benchmarks (Euler, MolDyn, MonteCarlo, RayTracer) taken from version v2.0 of the JGF benchmark release [13][2].
2. **Sequential X10:** Since the sequential subset of X10 overlaps significantly with the sequential subset of Java, this version is quite close to the Sequential

[2] Section 1 was excluded because it only contains microbenchmarks for low-level operations.

Java version in most cases. As in [2] we use a "lightweight" X10 version with regular Java arrays to avoid the large overheads incurred on X10 arrays in the current X10 implementation. However, all the other characteristics of X10 (*e.g.,* non-null used as the default type declaration, forbidden use of non-final static fields, etc.) are preserved faithfully in the Sequential X10 versions.

3. **Hand Parallelized X10:** This version emulates by hand the parallel versions that can be obtained by a compiler, assuming that annotations are added to the sequential X10 versions as outlined in Table 1.
4. **Parallel Java:** This is the threadv1.0 version of the JGF benchmarks [13], which contains multithreaded versions of five of the six Section 2 benchmarks and three of the four Section 3 benchmarks. The unimplemented benchmarks are FFT and Euler. Further, the threaded versions of two of the Section 2 benchmarks, SOR and SparseMatmult, were implemented using a different underlying algorithm from the sequential versions in v2.0.

All performance results were obtained using the following system settings:

– The target system is a p570 16-way Power6 4.7GHz SMP server with 186GB main memory running AIX5.3 J. In addition, each dual-core chip can access 32MB L3 cache per chip and 4MB L2 cache per core. The size of the L1 instruction cache is 64KB and data cache is 64KB.
– For all runs, SMT was turned off and a large page size of 16GB was used. The sequential Java and X10 versions used only 1 processor, where as the parallel Java and X10 versions used all 16 processors.
– The execution environment used for all Java runs is IBM's J9 VM (build 2.4, J2RE 1.6.0) with the following options, `-Xjit:count=0,optLevel=veryHot, ignoreIEEE -Xms1000M -Xmx1000M`.
– The execution environment used for all X10 runs was version 1.0.0. of the X10 compiler and runtime, combined with the same JVM as above, IBM's J9 VM (build 2.4, J2RE 1.6.0), but with additional options to skip null pointer and array bounds checks in X10 programs in accordance with the annotation in the X10 source program. The `INIT_THREADS_PER_PLACE` parameter was set to 1 and 16 for the sequential and parallel X10 runs respectively. (`MAX_NUMBER_OF_PLACES` was set to 1 in both cases.)
– The X10 runtime was also augmented with a special *one-way* synchronization mechanism to enable fine-grained producer-consumer implementations of X10's `finish` and `next` operations.
– For all runs, the main program was extended with a three-iteration loop within the same Java process, and the best of the three times was reported in each case. This configuration was deliberately chosen to reduce/eliminate the impact of JIT compilation time in the performance comparisons.

4.1 Sequential and Parallel Versions of X10

Figure 1 shows the speedup ratio of the serial and parallel X10 versions relative to the sequential Java version (JGF v2.0) for all ten benchmarks. An interesting

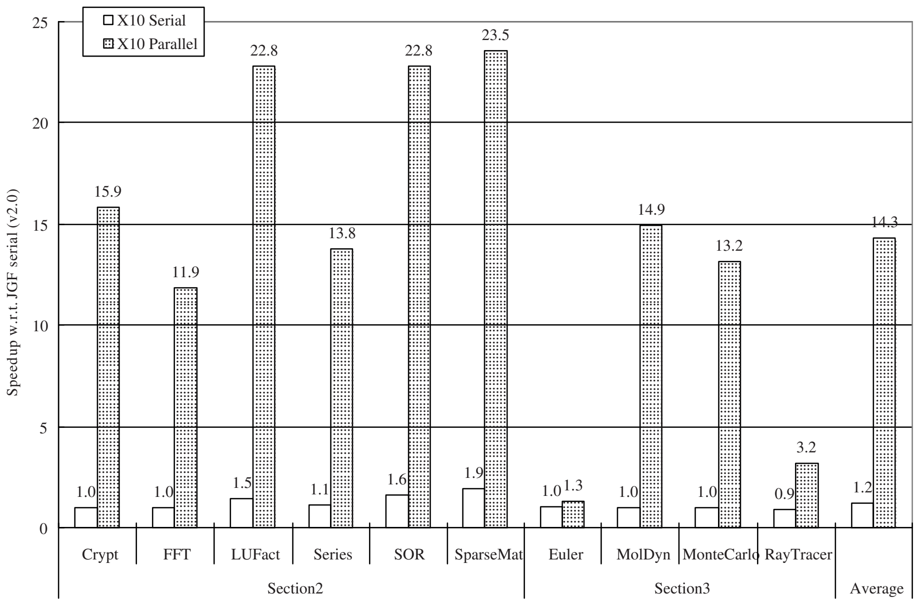


Fig. 1. Performance of Sequential and Parallel versions of X10 relative to Sequential Java

observation is that the sequential X10 version often runs faster than the sequential Java version. This is due to the annotations in the X10 program which enabled null checks and array bounds checks to be skipped. On average, the sequential X10 version was 1.2× faster than the sequential Java version, and the parallel X10 version was 11.9× faster than the sequential X10 version.

The sources of large speedups for SOR and LUFact were as follows. SOR's pipeline parallelism (faithful to the sequential version) was implemented using tightly-coupled one-way synchronizations which were added to the X10 runtime. The annotations for LUFact enabled a SPMD parallelization by following classical SPMDization techniques such as the approach outlined in [5].

The speedup was lowest for two benchmarks, *Euler* and *Raytracer*. The challenge in *Euler* is that it consists of a large number of small parallel loops which could probably benefit from more aggressive loop fusion and SPMD parallelization transformations than what was considered in our hand-parallelized experiments. The challenge in *Raytracer* is the classic trade-off between load balance (which prefers cyclic-style execution of the parallel loop) and locality (which prefers a block-style execution of the parallel loop).

4.2 Comparison with Parallel Java Versions

In this section, we extend results from the previous section by including results for Parallel Java executions as well. As mentioned earlier, Parallel Java versions (threadv1.0) are available for 8 of the 10 benchmarks. Results for these 8

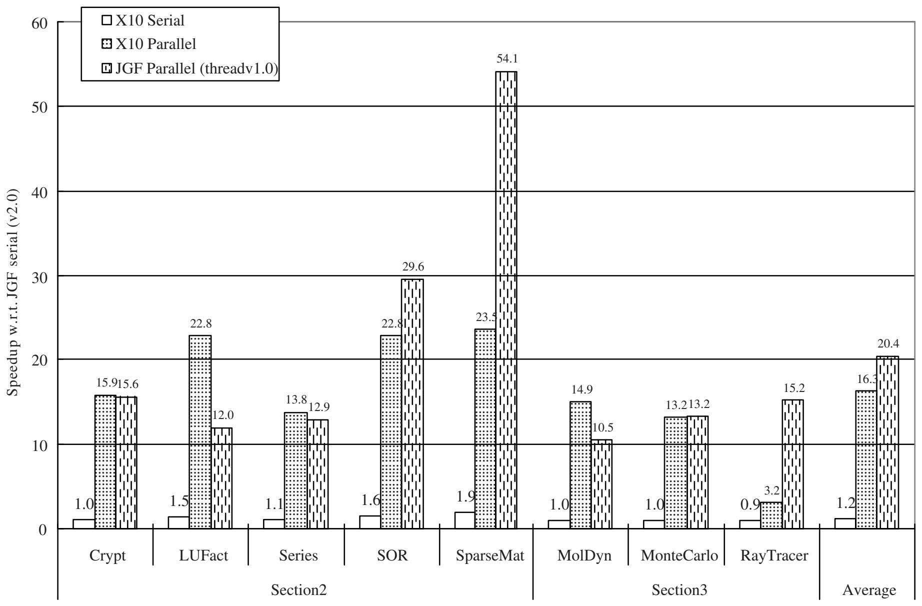


Fig. 2. Performance of Sequential and Parallel versions of X10 and Parallel Java relative to Sequential Java

benchmarks are shown in Figure 2. The two benchmarks for which the Parallel Java versions significantly out-performed the Parallel X10 versions were SOR and SparseMatmult. On closer inspection, we discovered that the underlying sequential algorithm was modified in both parallel versions (relative to the v2.0 sequential Java versions).

For SOR, the threadv1.0 parallel Java version uses a “red-black” scheduling of loop iteration to expose doall parallelism, even though this transformation results in different outputs compared to the sequential Java version. In contrast, the parallel X10 version contains pipeline parallelism that we expect can be automatically extracted from the sequential X10 version, and in fact returns the same output as the sequential X10 version.

For SparseMatmult, the thread v1.0 parallel Java version inserts an algorithmic step to sort non zero elements by their row value, so that the kernel computation can be executed as simple doall loop. Unfortunately, this additional step isn’t included in the execution time measurement for the Parallel Java case.

To take into account the algorithmic changes in the Parallel Java versions, Figure 3 show an alternate version of Figure 2 in which the algorithms used for the sequential and parallel X10 versions are modified to match the algorithm used in the parallel Java versions. With the algorithmic changes, we see that the performance of the parallel Java and X10 versions are now evenly matched in the average case.

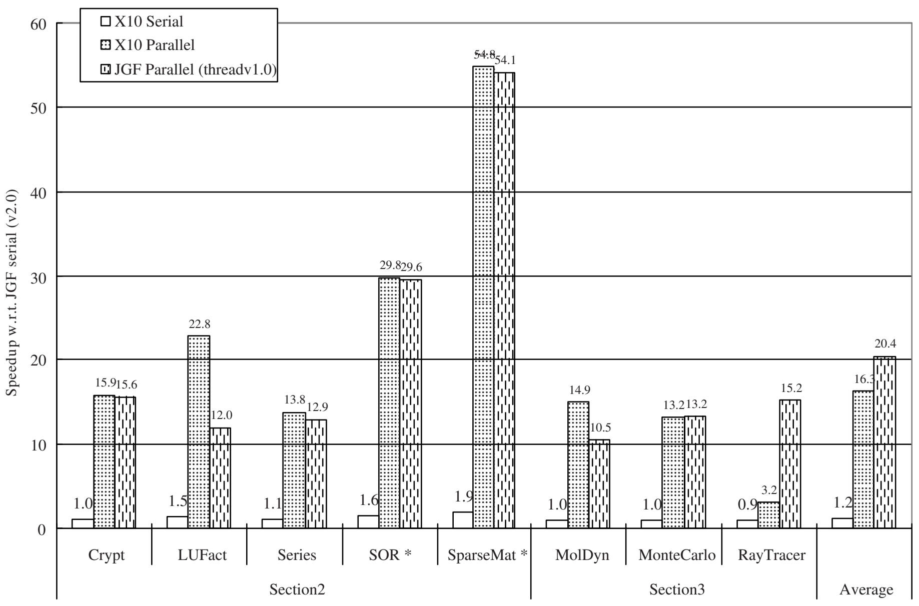


Fig. 3. Performance of Sequential and Parallel versions of X10 and Parallel Java relative to Sequential Java, with alternate Parallel X10 versions for SOR and SparseMatmult

5 Conclusions and Future Work

In this paper, we proposed a set of language extensions (enforced annotations and declarations) designed with a view to making modern object oriented languages more amenable to compiler parallelization. Many of the proposed extensions are motivated by best practices in software engineering for sequential programs. This is in contrast to the OpenMP approach where the annotations are geared towards explicit parallel programming and the correctness of user pragmas is not enforced by the language system.

We also performed a detailed case study of the Java Grande Forum benchmarks to confirm that the extensions proposed in this paper are effective in enabling compiler parallelization. Experimental results obtained on a 16-way Power6 SMP showed that the use of these language extensions can improve sequential execution time by 20% on average, and that a hand-simulation of automatically parallelized X10 programs can deliver speedup by matching the performance of the multithreaded Java versions of the JGF benchmarks.

The main topic for future work is to build an automatically parallelizing compiler which exploits the language extensions proposed in this paper. Another topic is to extend the definition of the language extensions to apply to explicitly parallel code *e.g.,* defining array views in the presence of distributions, and defining the semantics of in/out/inout intents for array views in the presence of concurrent array operations.

Acknowledgments

We are grateful to all X10 team members for their contributions to the X10 software used in this paper. We would like to especially acknowledge Vijay Saraswat's work on the design and implementation of dependent types in the current X10 implementation, and Chris Donawa and Allan Kielstra's implementation of experimental options for X10 in IBM's J9 virtual machine. While at IBM, Vivek Sarkar's work on X10 was supported in part by the Defense Advanced Research Projects Agency (DARPA) under its Agreement No. HR0011-07-9-0002. Finally we would like to thank Doug Lea, John Mellor-Crummey and Igor Peshansky for their feedback on this paper.

References

1. Allen, R., Kennedy, K.: Optimizaing Compilers for Modern Architectures. Morgan Kaufmann Publishers, San Francisco (2001)
2. Barik, R., Cave, V., Donawa, C., Kielstra, A., Peshansky, I., Sarkar, V.: Experiences with an smp implementation for x10 based on the java concurrency utilities. In: Workshop on Programming Models for Ubiquitous Parallelism (PMUP), held in conjunction with PACT 2006 (September 2006)
3. Charles, P., Donawa, C., Ebcioglu, K., Grothoff, C., Kielstra, A., von Praun, C., Saraswat, V., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. In: Proceedings of OOPSLA 2005, pp. 519–538. ACM Press, New York (2005)
4. Choi, J.-D., Gupta, M., Serrano, M.J., Sreedhar, V.C., Midkiff, S.P.: Stack allocation and synchronization optimizations for java using escape analysis. ACM Trans. Program. Lang. Syst. 25(6), 876–910 (2003)
5. Cytron, R., Lipkis, J., Schonberg, E.: A compiler-assisted approach to spmd execution. In: Supercomputing 1990: Proceedings of the 1990 ACM/IEEE conference on Supercomputing, Washington, DC, USA, pp. 398–406. IEEE Computer Society, Los Alamitos (1990)
6. Dagum, L., Menon, R.: OpenMP: An industry standard API for shared memory programming. IEEE Computational Science & Engineering (1998)
7. Eigenmann, R., Hoeflinger, J., Padua, D.: On the automatic parallelization of the perfect benchmarks. IEEE Trans. on parallel and distributed systems 9(1) (January 1998)
8. Gerlek, M.P., Stoltz, E., Wolfe, M.: Beyond induction variables: detecting and classifying sequences using a demand-driven ssa form. ACM Trans. Program. Lang. Syst. 17(1), 85–122 (1995)
9. Haghighat, M.R., Polychronopoulos, C.D.: Symbolic analysis for parallelizing compilers. Kluwer Academic Publishers, Dordrecht (1995)
10. Hall, M.W., Anderson, J.M., Amarasinghe, S.P., Murphy, B.R., Liao, S., Bugnion, E., Lam, M.S.: Maximizing multiprocessor performance with the SUIF compiler. IEEE Computer (1996)
11. Harper, R., Mitchell, J.C., Moggi, E.: Higher-order modules and the phase distinction. In: POPL 1990: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 341–354. ACM Press, New York (1990)

12. Hiranandani, S., Kennedy, K., Tseng, C.-W.: Preliminary experiences with the fortran d compiler. In: Proc. of Supercomputing 1993 (1993)
13. The Java Grande Forum benchmark suite, `http://www.epcc.ed.ac.uk/javagrande`
14. Jouvelot, P., Dehbonei, B.: A unified semantic approach for the vectorization and parallelization of generalized reductions. In: ICS 1989: Proceedings of the 3rd international conference on Supercomputing, pp. 186–194. ACM Press, New York (1989)
15. Jsr 305: Annotations for software defect detection, `http://jcp.org/en/jsr/detail?id=305`
16. Moreira, J.E., Midkiff, S.P., Gupta, M.: Supporting multidimensional arrays in java. Concurrency and Computation Practice & Experience (CCPE) 15(3:5), 317–340 (2003)
17. Pechtchanski, I., Sarkar, V.: Immutability Specification and its Applications. Concurrency and Computation Practice & Experience (CCPE) 17(5:6) (April 2005)
18. Pugh, W.: The omega test: A fast and practical integer programming algorithm for dependence analysis. In: Proc. of Super Computing 1991 (1991)
19. Rauchwerger, L., Amato, N.M., Padua, D.A.: Run-time methods for parallelizing partially parallel loops. In: Proceedings of the 9th ACM International Conference on Supercomputing, Barcelona, Spain, pp. 137–146 (July 1995)
20. Saraswat, V.: Report on the experimental language x10 version 1.01, `http://x10.sourceforge.net/docs/x10-101.pdf`
21. Sarkar, V.: The PTRAN Parallel Programming System. In: Szymanski, B. (ed.) Parallel Functional Programming Languages and Compilers. ACM Press Frontier Series, pp. 309–391. ACM Press, New York (1991)
22. Smith, L.A., Bull, J.M., Obdrzálek, J.: A parallel java grande benchmark suite. In: Supercomputing 2001: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM), p. 8. ACM Press, New York (2001)
23. Snyder, L.: The design and development of zpl. In: HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages, pp. 8–1–8–37. ACM Press, New York (2007)
24. MIT laboratory for computer science Supercomputing technologies group. Cilk 5.3.2 reference manual, `http://supertech.csail.mit.edu/cilk/manual-5.3.2.pdf`
25. Wolfe, M.: High Performance Compilers for Parallel Computing. Addison-Wesley Publishing Company, Reading (1996)
26. Xu, H., Pickett, C.J.F., Verbrugge, C.: Dynamic purity analysis for java programs. In: PASTE 2007: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, pp. 75–82. ACM Press, New York (2007)

Concurrency Analysis for Shared Memory Programs with Textually Unaligned Barriers

Yuan Zhang[1], Evelyn Duesterwald[2], and Guang R. Gao[1]

[1] University of Delaware, Newark, DE
{zhangy,ggao}@capsl.udel.edu
[2] IBM T.J.Watson Research Center, Hawthorne, NY
duester@us.ibm.com

Concurrency analysis is a static analysis technique that determines whether two statements or operations in a shared memory program may be executed by different threads concurrently. Concurrency relationships can be derived from the partial ordering among statements imposed by synchronization constructs. Thus, analyzing barrier synchronization is at the core of concurrency analyses for many parallel programming models. Previous concurrency analyses for programs with barriers commonly assumed that barriers are named or textually aligned. This assumption may not hold for popular parallel programming models, such as OpenMP, where barriers are unnamed and can be placed anywhere in a parallel region, i.e., they may be textually unaligned. We present in this paper the first interprocedural concurrency analysis that can handle OpenMP, and, in general, programs with unnamed and textually unaligned barriers. We have implemented our analysis for OpenMP programs written in C and have evaluated the analysis on programs from the NPB and SpecOMP2001 benchmark suites.

1 Introduction

Concurrency analysis is a static analysis technique that determines whether two statements or operations in a shared memory program may be executed by different threads concurrently. Concurrency analysis has various important applications, such as statically detecting data races [6,10], improving the accuracy of various data flow analyses [16], and improving program understanding. In general, precise interprocedural concurrency analysis in the presence of synchronization constraints is undecidable [15], and a precise intraprocedural concurrency analysis is NP-hard [18]. Therefore, a practical solution is to make a conservative estimate of all possible concurrency relationships, such that two statements that are not determined to be concurrent cannot execute in parallel in any execution of the program. If two statements are determined to be concurrent, they *may* execute concurrently.

In this paper we present a new interprocedural concurrency analysis that can handle parallel programming models with unnamed and textually unaligned barriers. We present our analysis in the context of the OpenMP programming model but our approach is also applicable to other SPMD (Single Program Multiple Data) parallel programming models.

OpenMP is a standardized set of language extensions (i.e., pragmas) and APIs for writing shared memory parallel applications in C/C++ and FORTRAN. Parallelism in an OpenMP program is expressed using the `parallel` construct. Program execution starts with a single thread called the *master thread*. When control reaches a `parallel` construct, a set of threads, called a *thread team*, is generated, and each thread in the team, including the master thread, executes a copy of the parallel region. At the end of the parallel region the thread team synchronizes and all threads except for the master thread terminate. The execution of the parallel region can be distributed within the thread team by work-sharing constructs (e.g., `for`, `sections` and `single`).

Synchronization is enforced mainly by global barriers and mutual exclusion (i.e., `critical` constructs and lock/unlock library calls). When a thread reaches a barrier it cannot proceed until all other threads have arrived at a barrier. In OpenMP, barriers are unnamed and they may be textually unaligned. Thus, threads may synchronize by executing a set of textually distinct barrier statements. Textually unaligned barriers make it difficult to reason about the synchronization structure in the program. Some parallel languages, therefore, require barriers to be textually aligned [19]. Textually unaligned barriers also hinder concurrency analysis because understanding which barrier statements form a common synchronization point is a prerequisite to analyzing the ordering constraints imposed by them. Our analysis is the first interprocedural concurrency analysis that can handle barriers in OpenMP and, in general, programs with unnamed and textually unaligned barriers. Figure 1 shows an OpenMP example program with a parallel region.

Barriers structure the execution of a parallel region into a series of synchronized execution phases, such that threads synchronize on barriers only at the beginning and at the end of each phase. Computing these execution phases for each parallel region provides the basic skeleton for ordering relationships among statements. Statements from different execution phases cannot execute concurrently. Thus, only statements within the same phase need to be examined for computing the concurrency relation.

To illustrate the concept of execution phases consider the sample program shown in Figure 1. The first execution phase, denoted as $(\mathit{begin}, \{b_1, b_2\})$, starts at the beginning of the parallel region and extends up to barriers b_1 and b_2. Note that barriers b_1 and b_2 establish a common synchronization point, i.e., they *match*. The next barrier synchronization point is at barriers $\{b_3, b_4\}$. Hence, the next execution phase is $(\{b_1, b_2\}, \{b_3, b_4\})$.

It is easy to see that statements from two different execution phases are ordered by barriers and thus cannot be concurrent. On the other hand, two statements from the same execution phase may be concurrent, such as S_1 and S_3 in Figure 1. However, barriers are not the only constructs that need to be considered to determine execution phases. Additional ordering constraints may be imposed by control constructs. Consider statements S_9 and S_{11} in Figure 1, which are on different branches of the conditional statement with predicate C_4. Since all threads agree on the value of predicate C_4 (i.e., the predicate is *single-valued*),

```
main()
{
    int my_ID, num, i, y, sum = 0;
    ......
    #pragma omp parallel private(my_ID, num, y)
    {
        my_ID = omp_get_thread_num();
C1:     if(my_ID > 2){
S1:         i = 0;
            #pragma omp barrier  // b1
S2:         y = i + 1;
        } else {
S3:         i = 1;
            #pragma omp barrier  // b2
S4:         y = i – 1;
        }
P1:     sum += my_ID;
C2:     if( my_ID == 0){
S5:         ......
            #pragma omp barrier  // b3
S6:         ......
        }
C3:     if( my_ID != 0){
S7:         ......
            #pragma omp barrier  // b4
S8:         ......
        }
P2:     num = omp_get_num_threads();
C4:     if(num > 2){
S9:         ......
            #pragma omp barrier  // b5
S10:        ......
        } else {
S11:        ......
            #pragma omp barrier  // b6
S12:        ......
        }
C5:     if(my_ID == 0)
            printf("i = %d\n", i);
    } // end of parallel
} // end of main
```

Fig. 1. Example OpenMP program. The OpenMP library function calls *omp_get_thread_number()* and *omp_get_num_threads()* return the thread identifier of the calling thread and the total number of threads in the current team, respectively.

statements S_9 and S_{11} can never be executed together in one execution, hence they cannot be concurrent. On the contrary, predicate C_1 is evaluated differently by different threads (i.e., the predicate is *multi-valued*), so that statements on the two branches may execute concurrently. Thus, another key issue in understanding the concurrency constraints is determining whether a control predicate is single- or multi-valued.

In this paper, we propose an interprocedural concurrency analysis technique that addresses the above ordering constraints imposed by synchronization and control constructs. Our analysis computes for each statement s the set of statements that may execute concurrently with s. The analysis proceeds in four major steps:

Step 1: CFG construction: The first step consists of constructing a control flow graph (CFG) that correctly models the various OpenMP constructs.
Step 2: Barrier matching: As a prerequisite to computing execution phases we need to understand which barrier statements synchronize together, i.e, which barrier statements *match*. We solve this problem as an extension to barrier matching analysis [20]. Barrier matching verifies that barriers in an SPMD program are free of synchronization errors. For verified programs, a *barrier matching function* is computed that maps each barrier statement s to the set of barrier statement that synchronize with s in at least one execution. Barrier matching was previously described for MPI programs and we have extended it to handle OpenMP programs. The computed barrier matching function is an input to the next step.

Step 3: Phase partition and aggregation: In this step, we first partition the program into a set of static execution *phases*. A *phase* (b_i, b_j) consists of a set of basic blocks that lie on a barrier-free path between barrier b_i and b_j in the CFG. We then aggregate phases (b_p, b_q) and (b_m, b_n) if b_p matches b_m, and b_q matches b_n. A dynamic execution phase at runtime is an instance of an aggregated static execution phase.
Step 4: Concurrency relation calculation: We first conservatively assume that statements from the same execution phase may be concurrent but statements from different phases are ordered and non-concurrent. We then apply a set of ordering rules that reflect the concurrency constraints from other OpenMP synchronization and work-sharing constructs to iteratively refine the concurrency relation.

We have implemented the analysis for OpenMP programs written in C and evaluated it on programs from the NPB [5] and SpecOMP2001 [17] benchmark suites. Our evaluation shows that our concurrency analysis is sufficiently accurate with the average size of a concurrency set for a statement being less than 6% of total statements in all but one program.

The rest of the paper is organized as follows. We first present related work in Section 2. The control flow graph is presented in Section 3. In Section 4 we first review the barrier matching technique and then present extensions to handle multi-valued expressions in OpenMP and structurally incorrect programs. Phase partition and aggregation is presented in Section 5, and the concurrency relation calculation is presented in Section 6. We present experimental results in Section 7, and finally conclude in Section 8.

2 Related Work

A number of researchers have looked at concurrency analysis for programs with barriers. Lin [10] proposed a concurrency analysis technique (called non-concurrency analysis) for OpenMP programs based on phase partitioning. Lin's analysis differs from our concurrency analysis in two main aspects. First, Lin's method is intraprocedural and cannot compute non-concurrency relationship across procedure calls. Second, Lin's method cannot account for synchronization across textually unaligned barriers. The analysis does not recognize that textually unaligned barriers may in fact synchronize together, resulting in spurious non-concurrency relationships. For instance, Lin's technique would wrongfully conclude that S_1 and S_3 in Figure 1 are non-concurrent.

Jeremiassen and Eggers [7] present a concurrency analysis technique that, similar to our analysis, first partitions the program into phases, then aggregates some phases together. Their analysis avoids the problem of having to identify whether textually unaligned barriers synchronize together by assuming that barriers are named through barrier variables. Barrier statements that refer to the same barrier are assumed to be matched. Their technique also does not account for concurrency constraints imposed by control constructs with a single-valued predicate. For instance, in Figure 1 their analysis would conclude that S_9 and S_{11} are concurrent.

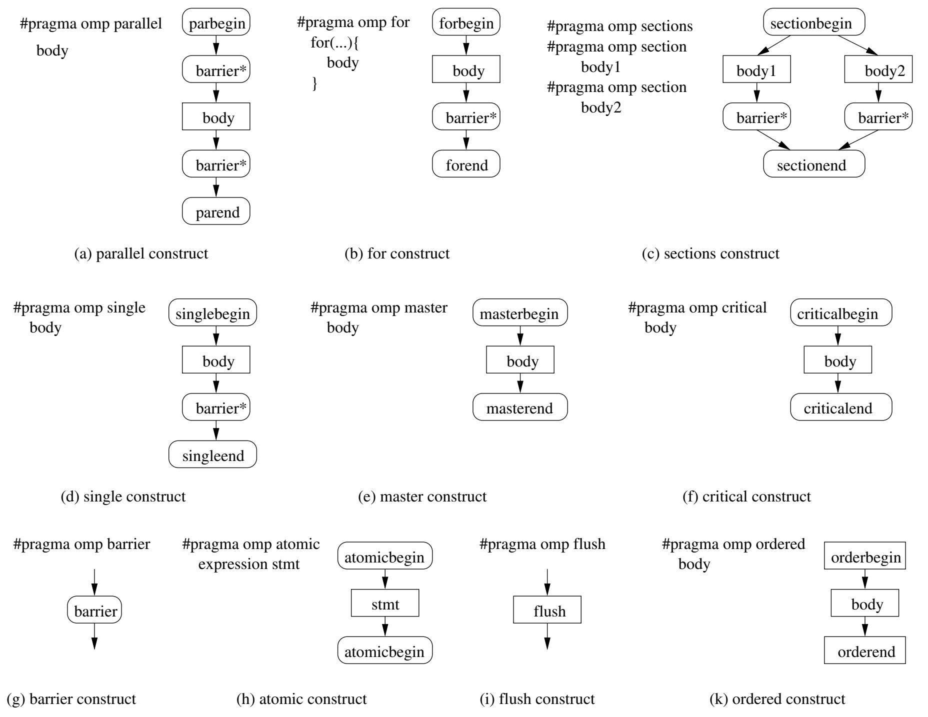


Fig. 2. Control flow graph construction

Kamil and Yelick [8] proposed a concurrency analysis method for the Titanium language [19] in which synchronization across textually unaligned barriers is not allowed.

There also has been a lot of work on concurrency analysis for other parallel programming languages, such as Ada and Java [3,2,6,11,13] in which synchronization is mainly enforced by event-driven constructs like post-wait/wait-notify. Agarwal et.al. [1] presents a may-happen-in-parallel analysis for X10 programs.

3 Step 1: Control Flow Graph Construction

The control flow graph for an OpenMP program is an extension of the control flow graph for a sequential program. Figure 2 illustrates the graph construction for each OpenMP construct. *Begin* and *end* nodes are inserted for each OpenMP directive with a construct body. To model the `sections` construct, we insert a control flow edge from the *begin* node to the first statement node of each section in the construct, and a control flow edge from the last statement node of each section to the *end* node of the `sections` construct. Constructs without a body statement (e.g., `barrier` and `flush`) are represented by a single block.

There is an implicit barrier at the end of the work-sharing constructs `for`, `sections` and `single`, unless the `nowait` clause is specified. Implicit barriers are depicted as `barrier*` in Figure 2. Similarly, there is an implicit barrier at the beginning of a parallel region, and an implicit barrier at the end of a parallel region.

4 Step 2: Barrier Matching

The second step in our concurrency analysis consists of identifying the matching barrier statements that synchronize together. Barrier matching analysis [20] was previously described for MPI programs. In this section we first review the MPI barrier matching analysis and then show how to extend it to handle OpenMP.

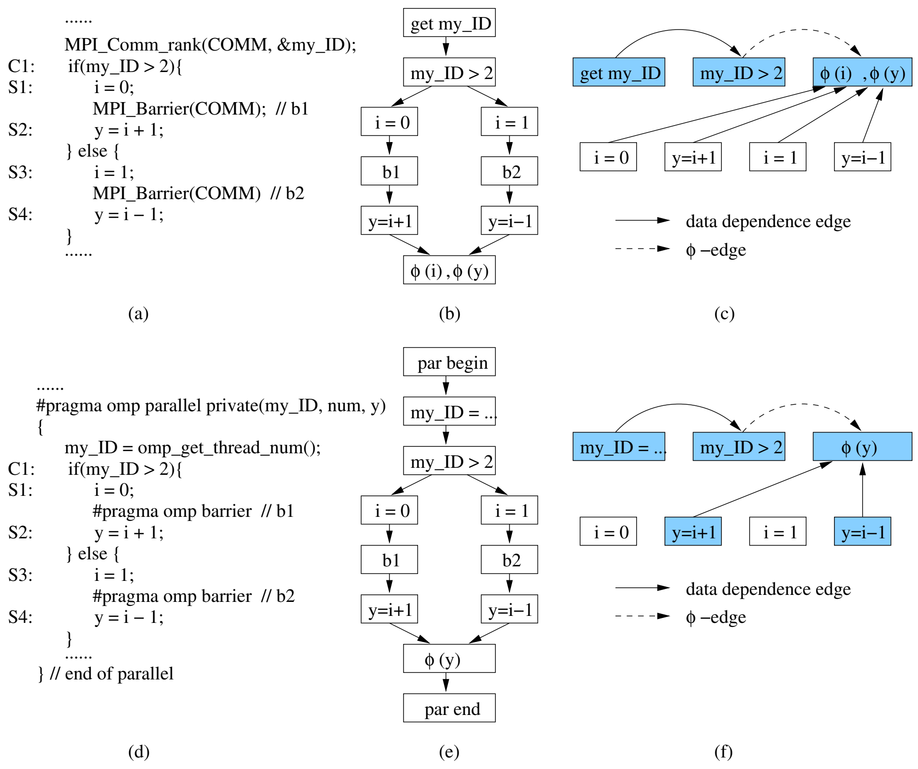
4.1 Review of Barrier Matching for MPI Programs

Barrier matching is an analysis and verification technique to detect stall conditions caused by barriers. When the program is verified, the analysis computes a barrier matching function that maps each barrier statement s to the set of barrier statements that synchronize with s in at least one execution. The MPI barrier matching analysis proceeds in three main steps:

Multi-valued Expression Analysis: In SPMD-style programs all threads execute the same program but they may take different program paths. The ability to determine which program paths may be executed concurrently requires an analysis of the *multi-valued* expressions in the program. An expression is called multi-valued if it evaluates differently in different threads. If used as a control predicate, multi-valued expressions split threads into different program paths that are executed concurrently by different threads. An example of a multi-valued expression is my_ID shown in Figure 3(a). Conversely, an expression that has the same value in all threads is called *single-valued*. SPMD programming paradigms like MPI or OpenMP usually contain multi-valued seed expressions, such as library calls that return the unique thread identifier. All other multi-valued expressions in the program are directly or indirectly dependent on these multi-valued seed expressions.

The interprocedural multi-valued analysis is solved as a forward slicing problem based on a revised program dependence graph. The revised program dependence graph contains nodes to represent statements that are connected through data dependence edges and so called ϕ-edges. ϕ-edges are based on the notion of ϕ-nodes in Static Single Assignment (SSA) form [4]. In SSA, a ϕ-node is inserted at a join node where multiple definitions of a variable merge. The predicate that controls the join node is called a ϕ-gate. A ϕ-edge connects a ϕ-gate with the corresponding ϕ-node. Multi-valued expressions result as those expressions that are reachable from a multi-valued seed expression along either data-dependence or ϕ-edges in the revised program dependence graph.

Figure 3(c) illustrates the revised program dependence graph and multi-valued expression analysis for the MPI program shown in Figure 3(a). It is important



```
        ......
        MPI_Comm_rank(COMM, &my_ID);
C1:     if(my_ID > 2){
S1:         i = 0;
            MPI_Barrier(COMM);  // b1
S2:         y = i + 1;
        } else {
S3:         i = 1;
            MPI_Barrier(COMM)  // b2
S4:         y = i – 1;
        }
        ......
```

```
    ......
    #pragma omp parallel private(my_ID, num, y)
    {
        my_ID = omp_get_thread_num();
C1:     if(my_ID > 2){
S1:         i = 0;
            #pragma omp barrier  // b1
S2:         y = i + 1;
        } else {
S3:         i = 1;
            #pragma omp barrier  // b2
S4:         y = i – 1;
        }
        ......
    } // end of parallel
```

Fig. 3. An MPI program (a), its CFG (b), and its revised program dependence graph (c). An OpenMP program (d), its CFG (e), and its revised program dependence graph (f). The multi-valued expression slices are shown as shaded nodes in (c) and (f).

to note that variables i and y are single-valued for the executing threads inside the conditional statement but they become multi-valued after the conditional paths merge at the ϕ-node.

Barrier Expressions: A barrier expression at a node n in the CFG represents the sequences of barriers that may execute along any paths from the beginning of the program to node n. Barrier expressions are regular expressions with barrier statements and function labels as terminal symbols, and three operators: concatenation $(\cdot)$, alternation $(|)$ and quantification $(*)$, which represents barriers in a sequence, in a condition, and in a loop, respectively. For example, the barrier expression for Figure 3(a) is $(b_1|\ b_2)$. A barrier expression is usually represented by a barrier expression tree. Figure 4 shows the barrier expression tree for the program shown in Figure 1.

Barrier matching: The final step combines the results of the previous two steps to detect potential stall conditions caused by barriers. Recall that multi-valued predicates create concurrent paths. Thus, a barrier subtree whose root is an alternation with a multi-valued predicate describes two concurrent barrier

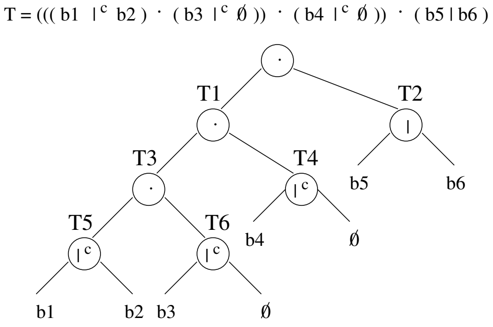

T = (((b1 |c b2) · (b3 |c ∅)) · (b4 |c ∅)) · (b5 | b6)
T1
T2
T3
T4
T5
T6
b1
b2
b3
∅
b4
∅
b5
b6

Fig. 4. The barrier expression tree for the program in Figure 1. The symbol $|^c$ denotes alternation with a multi-valued predicate.

sequences. Similarly, a quantification tree with a multi-valued predicate describes concurrent barrier sequences in a loop in which threads concurrently execute different numbers of iterations.

A barrier tree that does not contain either concurrent alternation or concurrent quantification describes a program in which all threads execute the same sequence of barriers (although the sequence may be different across different executions of the program). Such a tree is obviously free of barrier synchronization errors. A concurrent quantification tree signals a synchronization error because concurrent threads execute different numbers of loop iterations and hence different numbers of barriers. Therefore, the barrier verification problem comes down to checking that all concurrent alternation subtrees in the program's barrier tree are well-matched, i.e., the two alternation subtrees always produce barrier sequences of the same length. The barrier matching analysis implements this check by a counting algorithm that traverses the two subtrees of each concurrent alternation tree. Details of the counting algorithm can be found in [20].

After verifying a concurrent alternation barrier tree, the analysis computes the barrier matching function by ordering the leave nodes from each of its two subtrees in a depth-first order, and then matching barriers in the same position of the two ordered sequences.

4.2 Multi-valued Expressions Analysis for OpenMP Programs

In order to use barrier matching for our concurrency analysis, we developed an extension of the multi-valued expressions analysis for shared variables. In MPI programs all variables are local to the executing thread. In OpenMP programs, on the other hand, variables are either shared or private. Private variables are stored in thread private memory and observable only by the executing thread. Private variables in OpenMP can therefore be handled in the same way as variables in MPI programs. Shared variables in OpenMP programs are stored in global memory and observable by all threads simultaneously. The order of a sequence of shared variables reads and writes that is observed by a thread is not only determined by the program order, but also influenced by the memory consistency model. The memory consistency model also complicates the multi-valued expressions analysis by making some expressions that are not dependent on the seed expression multi-valued.

To simplify the presentation we first assume the sequential consistency model (SC model) [9]. The SC model requires that the result of the execution is the same as if the operations of all the threads were executed in some sequence, and the operations of each individual thread occur in the order specified by the program. All threads observe the same sequence of operations. For instance, consider the program fragment in Figure 5(a), in which variable i is shared, and variable x is private. Figure 5(b) illustrates one of the possible execution sequences with two threads under the SC model. Thread 1 reads $i = 0$ in statement S_3 due to the preceding write operation issued by thread 2, while thread 2 reads $i = 1$ in statement S_3. In this example the shared variable i is multi-valued even if it is not dependent on the seed expression. Therefore, we need to extend the multi-valued expressions analysis for shared variables under the SC model by incorporating all read operations on shared variables as multi-valued, and no further slicing for shared variables is needed.

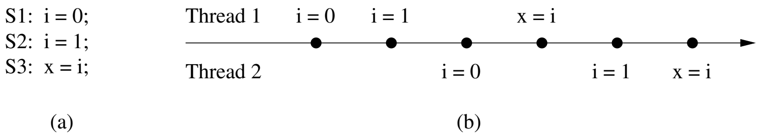


Fig. 5. (a) Example program (b) An execution sequence under the SC model

OpenMP provides a relaxed memory consistency model, under which a thread may have its own temporary view of memory which is not required to be consistent with global memory at all times. As a consequence, different threads may observe different sequences of shared memory operations, and the shared variable i in Figure 5(a) may be single-valued in some executions. However, since the execution sequence illustrated in Figure 5(b) is also a valid execution under OpenMP's relaxed memory model, we still need to conservatively incorporate all read operations on shared variables as multi-valued.

Consider again our sample program shown in Figure 3(d). Since variable i is shared, we treat two reads on i at S_1 and S_3 as multi-valued. We then apply the interprocedural forward slicing algorithm used in the original MPI analysis to compute multi-valued expressions for private variables in OpenMP. Figure 3(f) shows the resulting multi-valued expressions as the set of shaded nodes in the graph.

As in the original MPI multi-valued expression analysis, we assume OpenMP and other library calls are annotated as either single- or multi-valued. Arrays are treated as scalar variables and pointers are conservatively handled by treating every pointer dereference and every variable whose address is taken as multi-valued.

4.3 Barrier Trees and Barrier Matching for OpenMP Programs

Once the multi-valued expressions have been computed, barrier tree construction and barrier matching for OpenMP programs proceed as described for MPI programs. Figure 4 shows the barrier expression tree for the program shown in Figure 1. Barrier matching checks the three concurrent alternation subtrees T_5, T_6 and T_4. The analysis verifies subtree T_5 as correct and reports that barriers b_1 and b_2 match. However, the other subtrees T_6 and T_4 cannot be statically verified and the analysis would report a potential error, warning that the subtrees are structurally incorrect.

4.4 Handling Structurally Incorrect Programs

Barrier matching analysis produces a barrier matching function only for verified programs. As a static analysis, barrier matching is conservative and may therefore reject a program, although the program produces no synchronization errors at runtime. Programs that will always be rejected are so called *structurally incorrect* programs. Informally, structural correctness means that a program property holds for a program if it holds for every structural component of the program, (i.e., every statement, expression, compound statement, etc.). In other words, a structurally incorrect program contains a component that, if looked at in isolation, has a synchronization error, although in the context of the entire program no runtime error may result. Figure 1 is an example of a structurally incorrect program because it contains two structural components, the conditionals C_2 and C_3 that, if looked at in isolation, are incorrect. Thus, the overall program is deemed incorrect although no runtime synchronization error would result because C_3 is the logical complement of C_2. As reported in the previous section, barrier matching analysis reports a potential error for each of the two conditional components.

We discuss in this section modifications to compute partial barrier matching information for programs whose synchronization structure is dynamically correct (i.e., the program terminates) even if they cannot be statically verified. Our approach to handling structural incorrectness is to isolate the program region that cannot be statically verified, and to partition the program into structurally correct and structurally incorrect regions. Based on this partition we can apply barrier matching and, in turn, our concurrency analysis for the structurally correct components of the program. For the structurally incorrect regions we conservatively assume that all statements may execute concurrently.

When barrier matching encounters a program with a structurally incorrect component p, a synchronization error is detected when processing the root of the barrier expression subtree that represents p. We refer to such structural component as an error component. For example, the barrier tree in Figure 4, contains two error components T_4 and T_6.

Based on these error components we define two well-matched regions of a structurally incorrect program. The first well-matched region consists of any sequence of statements along an error-component-free path in the CFG that starts at the program entry and terminates at a program point immediately preceding an error component. Similarly, the second well-matched region consists of any sequence of statements along an error-component-free path in the CFG that starts at a program point immediately following an error component and terminates at program exit. We define the "structurally incorrect region" as the remainder of the program, that is, any statement that is not included in one of the above well-matched regions. We conservatively treat all statements in the structurally incorrect region as concurrent and compute barrier matching functions for the structurally correct regions.

Consider again our example in Figure 4 and recall that barrier matching reports two error components T_4 and T_6. The two well-matched regions of the program in Figure 1 are defined as follows. The first region starts at program entry and terminates at program point P_1 in Figure 1 which immediately precedes the error component T_6. The second region starts at program point P_2 which immediately follows the error component T_4 and extends up to program exit. All statements between P_1 and P_2 are assumed to be concurrent.

5 Step 3: Phase Partition and Aggregation

The third step of the OpenMP concurrency analysis uses the computed barrier matching function to divide the program into a set of static phases. A static phase (b_i, b_j) consists of a sequence of basic blocks along all barrier-free paths in the CFG that start at the barrier statement b_i and end at the barrier statement b_j. Note that b_i and b_j may refer to the same barrier statement.

The phase partition method proceeds as proposed by Jeremiassen and Eggers [7]. First we assume each barrier statement b_i corresponds to a new global variable V_{b_i}. We then treat each barrier statement as a use of its corresponding barrier variable, followed by definitions of all barrier variables in the program. The problem of phase partition is then reduced to computing live barrier variables in the program. Recall that a variable v is live at program point p if the value of v at p is used before being re-defined along some path in the control flow graph starting at p. Precise interprocedural live analysis has been described in [12]. Let $Live(b)$ denote the set of barrier variables live at the barrier b. The set of static phases in an OpenMP program is then summarized as:

$$\{(b_i, b_j) | V_{b_j} \in Live(b_i),\ \ for\ \ all\ \ i\ \ and\ \ j\}$$

In order to determine to which phases a basic block u belongs, we need to reverse the control flow edges in the CFG and calculate live barrier variables for each basic block again. Let $LiveR(u)$ denote the set of live barrier variables at basic block u in the reversed CFG. The phases to which block u belongs are:

$$\{(b_i, b_j) | b_i \in LiveR(u) \wedge b_j \in Live(u)\}$$

According to the barrier matching information, we then aggregate phases (b_m, b_n) and (b_p, b_q) if barriers b_m matches b_p and b_n matches b_q. A dynamic execution phase is an instance of an aggregated phase at runtime.

6 Step 4: Concurrency Relation Calculation

The final step of our concurrency analysis consists of calculating the concurrency relation among basic blocks. Since basic blocks from different aggregated phases are separated by barriers, no two blocks in different phases can be executed concurrently. We can therefore establish a first safe approximation of the concurrency relation in the program by assuming that all blocks from the same aggregated phase may be concurrent. However, this first approximation is overly conservative and does not take concurrency constraints from certain OpenMP constructs into account. We have developed the following set of concurrency rules that address these constraints to refine the initial concurrency approximation.

1. **(Concurrency Rule)** Any two (possibly identical) basic blocks from the same aggregated phase are concurrent. The set of concurrency relationships obtained from this rule is denoted as CR.
2. **(Non-concurrency Rules)**
   (a) Any two basic blocks from a `master` construct under the same parallel region are not concurrent because they are executed serially by the master thread.
   (b) Any two basic blocks from `critical` constructs with the same name (or from within the lock regions, enclosed by the omp_set_lock() and omp_unset_lock() library calls, that are controlled by the same lock variable) are not concurrent because they are executed mutually exclusively. Note that we treat two potentially aliased lock variables as different.
   (c) Two blocks in the same `ordered` construct are not concurrent because the `ordered` construct body within a loop is executed in the order of loop iterations.
   (d) Two blocks from the same `single` construct that is not enclosed by a sequential loop are not concurrent. Note that OpenMP requires a `single` construct body to be executed by one thread in the team, but it does not specify which thread. Therefore two instances of a `single` construct inside a sequential loop might be executed by two different threads concurrently.

   The set of non-concurrency relationships obtained from the non-concurrency rules is denoted as NCR.

Finally, the concurrency relation among basic blocks results as $CR - NCR$.

Returning to our sample program in Figure 1. S_1 and S_3 are concurrent because they are in the same aggregated phase $(start, \{b_1, b_2\})$. The same holds for S_2 and S_4. However, S_9 and S_{11} are not concurrent because barrier b_5 does not match barrier b_6 (due to the single-valued predicate C_4) thus S_9 and S_{11} are in different phases.

7 Experimental Evaluations

We have implemented the concurrency analysis for OpenMP/C programs on top of the open-source CDT (C Development Tool) in Eclipse. The Eclipse CDT

Table 1. Experimental results

Benchmark	FT	IS	LU	MG	SP	quake
Source	NPB2.3-C	NPB3.2	NPB2.3-C	NPB2.3-C	NPB2.3-C	SpecOMP2001
# Souce Lines	1162	629	3471	1264	2991	1591
# Blocks	682	278	2132	909	2503	1191
# Procedures	17	9	18	15	22	27
# Barriers	13	5	30	28	67	13
OpenMP constructs	`single` `master` `for` `critical`	`for`	`for` `single` `critical` `master` `flush`	`single` `for` `critical`	`for` `master`	`for`
# Aggr. phases	29	11	41	103	223	24
Max. concurrency set size	101	59	83	256	130	33
Relative max. concurrency set size	14.8%	21.2%	3.9%	28.1%	5.2%	2.8%
Avg. concurrency set size	40	36	23	50	52	15
Relative avg. concurrency set size	5.9%	12.9%	1.1%	5.5%	2.1%	1.3%

constructs Abstract Syntax Trees for C programs. We evaluated the effectiveness of our OpenMP concurrency analysis on a set of OpenMP programs from the NPB (Nas Parallel Benchmarks) and SpecOMP2001 benchmark suites, as shown in Table 1.

FT (3-D FFT), LU (LU solver), MG (Multigrid), and SP (Pentadiagonal solver) are derived from the serial Fortran versions of NPB2.3-serial by the Omni OpenMP compiler project [14]. IS (Integer sort) is an OpenMP C benchmark from NPB3.2. Quake from SpecOMP2001 benchmark suite simulates seismic wave propagation in large basins.

The top part of Table 1 lists several characteristics of the benchmark programs such as the number of source lines, the number of barriers, either explicit or implicit, and the various OpenMP constructs used in each benchmark.

The results of the concurrency analysis are shown in the bottom part of the table. As an intermediate result, the table lists the number of aggregated phases that have been computed. To estimate the accuracy of our concurrency analysis we computed the average and maximum set size among the concurrency sets for all nodes in the CFG. Our CFG is based on the CDT and includes statement level block nodes. Set sizes would be smaller if statements would be composed into basic block nodes. The table shows the absolute set size and the relative size which is the percentage of the total number of nodes in the CFG. Recall that the concurrency set of a block b consists of a set of blocks that might execute concurrently with b in at least one execution. A concurrency set is usually a superset of the real concurrency relation. Therefore the smaller the concurrency set, the less conservative our concurrency analysis is. Table 1 indicates that our analysis is not overly conservative since the size of the average concurrency set is less than 6% of the total blocks for all benchmarks except IS, for which the average concurrency set is 12.9% of the total number of blocks in the program.

8 Conclusions

In this paper we present the first interprocedural concurrency analysis that can handle OpenMP and, in general, shared memory programs with unnamed and textually unaligned barriers. Our approach is built on the barrier matching technique that has previously been described to verify barrier synchronization in MPI. We extended barrier matching to handle shared variables and OpenMP. We have implemented our analysis for OpenMP C programs and evaluated the effectiveness of our analysis using benchmarks from the NPB and SpecOMP2001 benchmark suites. The experimental results confirm that our analysis is not overly conservative. We are currently exploring the use of our concurrency analysis in combination with a dynamic data race detection tool by limiting the instrumentation points that have to be considered during dynamic checking. Other potential uses are in combination with performance tools to point the user to areas with low levels of concurrency.

Acknowledgement

This work is, in part, based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.

References

1. Agarwal, S., Barik, R., Sarkar, V., Shyamasundar, R.K.: May-happen-in-parallel analysis of x10 programs. In: PPoPP 2007: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 183–193. ACM Press, New York (2007)
2. Callahan, D., Kennedy, K., Subhlok, J.: Analysis of event synchronization in a parallel programming tool. In: PPOPP 1990: Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 21–30 (1990)
3. Callahan, D., Sublok, J.: Static analysis of low-level synchronization. In: PADD 1988: Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, pp. 100–111 (1988)
4. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. 13(4), 451–490 (1991)
5. NASA Advanced Supercomputing Divsion. Nas parallel benchmarks, `http://www.nas.nasa.gov/Software/NPB/`
6. Duesterwald, E., Soffa, M.L.: Concurrency analysis in the presence of procedures using a dataflow framework. In: TAV4: Proceedings of the Symposium on Testing, Analysis, and Verification, pp. 36–48 (1991)

7. Jeremiassen, T., Eggers, S.: Static analysis of barrier synchronization in explicitly parallel systems. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), Montreal, Canada (1994)
8. Kamil, A.A., Yelick, K.A.: Concurrency analysis for parallel programs with textually aligned barriers. Technical Report UCB/EECS-2006-41, EECS Department, University of California, Berkeley (April 2006)
9. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Computers 28(9), 690–691 (1979)
10. Lin, Y.: Static nonconcurrency analysis of openmp programs. In: First International Workshop on OpenMP (2005)
11. Masticola, S.P., Ryder, B.G.: Non-concurrency analysis. In: PPOPP 1993: Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 129–138 (1993)
12. Myers, E.M.: A precise inter-procedural data flow algorithm. In: POPL 1981: Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 219–230. ACM Press, New York (1981)
13. Naumovich, G., Avrunin, G.S., Clarke, L.A.: An efficient algorithm for computing mhp information for concurrent java programs. In: ESEC/FSE- 7: Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 338–354 (1999)
14. Omni OpenMP Compiler Project. Omni OpenMP Compiler, `http://phase.hpcc.jp/Omni/home.html`
15. Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. ACM Transactions on Programming languages and Systems (TOPLAS) 22(2), 416–430 (2000)
16. Sreedhar, V., Zhang, Y., Gao, G.: A new framework for analysis and optimization of shared memory parallel programs. Technical Report CAPSL- TM-063, University of Delaware, Newark, DE (2005)
17. Standard Performance Evaluation Corporation. SPEC OMP (OpenMP benchmark suite), `http://www.spec.org/omp/`
18. Taylor, R.N.: Complexity of analyzing the synchronization structure of concurrent programs (1983)
19. Yelick, K., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., Aiken, A.: Titanium: A high-performance Java dialect. In: ACM (ed.) ACM 1998 Workshop on Java for High-Performance Network Computing ACM Press, New York (1998)
20. Zhang, Y., Duesterwald, E.: Barrier matching for programs with textu- ally unaligned barriers. In: PPoPP 2007: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 194–204 (2007)

Iteration Disambiguation for Parallelism Identification in Time-Sliced Applications

Shane Ryoo, Christopher I. Rodrigues, and Wen-mei W. Hwu

Center for Reliable and High-Performance Computing and
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
{sryoo,cirodrig,hwu}@crhc.uiuc.edu

Abstract. Media and scientific simulation applications have a large amount of parallelism that can be exploited in contemporary multi-core microprocessors. However, traditional pointer and array analysis techniques often fall short in automatically identifying this parallelism. This is due to the allocation and referencing patterns of time-slicing algorithms, where information flows from one time slice to the next. In these, an object is allocated within a loop and written to, with source data obtained from objects created in previous iterations of the loop. The objects are typically allocated at the same static call site through the same call chain in the call graph, making them indistinguishable by traditional heap-sensitive analysis techniques that use call chains to distinguish heap objects. As a result, the compiler cannot separate the source and destination objects within each time slice of the algorithm. In this work we discuss an analysis that quickly identifies these objects through a partially flow-sensitive technique called iteration disambiguation. This is done through a relatively simple aging mechanism. We show that this analysis can distinguish objects allocated in different time slices across a wide range of benchmark applications within tens of seconds even for complete media applications. We will also discuss the obstacles to automatically identifying the remaining parallelism in studied applications and propose methods to address them.

1 Introduction

The pressure of finding exploitable coarse-grained parallelism has increased with the ubiquity of multi-core processors in contemporary desktop systems. Two domains with high parallelism and continuing demands for performance are media and scientific simulation. These often operate on very regular arrays with relatively simple pointer usage, which implies that compilers may be able to identify coarse-grained parallelism in these applications. Recent work has shown that contemporary analyses are capable of exposing a large degree of parallelism in media applications written in C [14].

However, there are still obstacles to be overcome in analyzing the pointer behavior of these applications. A significant percentage of these applications are based on time-sliced algorithms, with information flowing from one time slice to the next. The code of these applications typically consists of a large loop, often with multiple levels of function calls in the loop body, where each iteration corresponds to a time slice. Results from a previous iteration(s) are used for computation in the current iteration. While there are typically dependences between iterations, there is usually an ample amount of coarse-grained parallelism within each iteration, or time slice, of the algorithm. For example, in video encoding applications, there is commonly a dependence between the processing of consecutive video frames, a fundamental time slice of video processing. However, there is significant parallelism within the video frame, where thousands of sub-pictures can be processed in parallel.

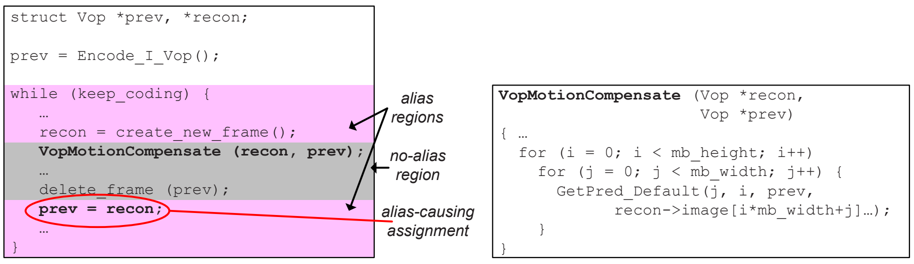
From our experience, time-sliced algorithms typically operate by reading from data objects that are written in the previous time slice, performing substantial computation, and writing to data objects that will be used by the next time slice. The primary step of parallelizing the computation of the time slice lies in the disambiguation between the input and output objects of the time slice. This proves to be a challenging task for memory disambiguation systems today. The difficulty lies in the fact that the code is cyclic in nature. The output objects of an iteration must become the input of the next iteration. That is, the input and output objects are coupled by either a copying action or a pointer swapping operation during the transition from one time slice to the next. Without specialized flow sensitivity, a memory disambiguation system will conclude that the input and output objects cannot be distinguished from each other.

Three different coding styles exist for transitioning output objects to input objects in time-sliced algorithms:

1. **Fixed purpose:** The data objects operated on are allocated in an acyclic portion of the program and designated specifically as input and output structures. At the end of an iteration, the data in the output structure are copied to the input structure for use in the next iteration. Previous parallelism-detection work [14] assumes this coding style.
2. **Swapped buffer**, or double-buffering: Two or more data objects are created in the acyclic portion of the program and pointed to by input(s) and output pointers. At the end of an iteration, the pointer values are rotated.
3. **Iterative allocation:** A new data object is allocated and written during each iteration of the primary program loop and assigned to the output pointer. At the end of the iteration, the output object of the current iteration is assigned to the input pointer in preparation for consumption by the next iteration. Objects that are no longer needed are deallocated.

Many compiler analyses, even some that are flow-sensitive, will see the pointers in the latter two categories as aliasing within the loop. This is true when considering the static code, since the stores in one iteration are writing the objects that will be read in the next iteration. When the dynamic stream of instructions is considered, however, the pointers and any stores and loads from them are independent within a single iteration of the loop. In media and simulation applications, this is where much of the extractable loop-level parallelism lies. The goal of our work is to create a targeted, fast, and scalable analysis that is *cycle-sensitive*, meaning that it can distinguish objects allocated or referenced in cyclic patterns. We will address the third category; the first is adequately disambiguated by traditional points-to analysis and the second can be handled by tracking the independence of the pointers via alias pairs [10] or connection analysis [4].

Figure 1 shows an example of this within a video encoder. At least two related images exist during MPEG-style P-frame encoding: the frame currently being encoded and reconstructed and the frame(s) that was/were previously reconstructed. During motion estimation and compensation, the encoder attempts to achieve a close match to the desired, current image by copying in pieces from the previous reconstructed frame. In terms of memory operations, it reads data from the previous frame and writes it to the current frame. In the reference MPEG-4 encoder [11], these objects can come from the same static call sites and calling contexts and must be disambiguated by determining that they were created in different iterations of a control flow cycle.


```
struct Vop *prev, *recon;

prev = Encode_I_Vop();

while (keep_coding) {
    …
    recon = create_new_frame();
    VopMotionCompensate (recon, prev);
    …
    delete_frame (prev);
    prev = recon;
    …
}
```

(a) Pseudocode of the frame-by-frame loop of encoding: `recon` and `prev` alias because of circled assignment.

```
VopMotionCompensate (Vop *recon,
                     Vop *prev)
{ …
  for (i = 0; i < mb_height; i++)
    for (j = 0; j < mb_width; j++) {
      GetPred_Default(j, i, prev,
            recon->image[i*mb_width+j]…);
    }
}
```

(b) Pseudocode of `VopMotionCompensate`: `recon` and `prev` point to different objects at call site and do not alias within the function.

Fig. 1. An example of cycle-sensitivity enabling parallelism

In the loop shown in Figure 1(a), the pointers `prev` and `recon` will point to the same object at the circled assignment. Within the middle shaded region, however, the two pointers will always point to different objects, since `recon` is allocated in the current loop iteration and `prev` is not. Thus, loops within this region, such as those within `VopMotionCompensate` as shown in Figure 1(b), can have their iterations executed in parallel. The goal of our analysis, *iteration disambiguation*, is to distinguish the most recently allocated object from older objects that are allocated at the same site. This and similar cases require an interprocedural analysis to be effective because the objects and loops involved nearly always span numerous functions and loop scopes.

We first cover related work in Section 2. We then describe the analysis in Section 3. Section 4 shows analysis results for several benchmark programs. We conclude with some final comments and future work.

2 Related Work

The intent of iteration disambiguation is to quickly distinguish objects that come from the same static call site and chain, but different iterations of a control flow loop. It is designed as an extension of a more general pointer analysis system. By doing this, the analysis is able to capture cases which general analyses are incapable of distinguishing or cannot accomplish in an inexpensive manner. For an overview of previous work in pointer analysis, we refer to [8].

The closest existing work to iteration disambiguation, in terms of disambiguation results, is connection analysis, as proposed by Ghiya and Hendren [4]. It attempts to overcome the limitation of basic points-to analysis and naming schemes by using a storeless model [3] to determine the sets of interacting, or connected, pointers that may alias. At each program point and for each calling context, the analysis maintains the sets of connected local and global pointer variables. Each pointer's connection set approximates the set of other pointers with which it may alias. Connections are removed, or "killed", for a pointer when it is assigned, and replaced with the connection information of the right-hand expression, if any. At a given program point, disjoint connection sets for two pointers indicates that they have not interacted in any way and do not conflict. Interprocedural analysis is handled by exhaustive inlining.

Connection analysis distinguishes new and old objects in a loop by making a newly-allocated object be initially unconnected to anything else. The basic example shown in Figure 1 can be disambiguated by connection analysis because the assignment to the variable `recon` kills `recon`'s connection to previous objects. However, control flow within the loop body can foil connection analysis. When a variable is assigned on only some paths through a loop, its connections are not killed on the other paths. This leads to pointer aliasing of the variable and its connected variables after the paths merge. This case has been observed in video encoders and obscures parallelism in those applications.

Shape analysis is a flow-sensitive analysis which attempts to analyze the pattern of pointer usage to express relationships between different instances of the same abstract heap object, examples of which are presented in [5,6,15]. This can improve pointer disambiguation for recursive data structures. Generally, the possible types of structures must be known a priori to the analysis, with the exception of [6]. The purpose of this work is not to identify the relationship between instances of recursive structures, but to disambiguate "top-level" pointers retained in local or global variables that refer to objects created in different iterations of a control flow cycle. Shape analysis generally does not focus on this aspect of pointer behavior.

Two independently developed may-alias points-to analyses [16,17] can distinguish different elements of an array where array elements are initialized in a loop using a monotonically increasing variable to index the array. In their analyses, there can be no data transfer between the array elements, whereas the objects targeted by iteration disambiguation are coupled through copying or pointer assignment at the end of loop iterations. Their work is complementary to ours and can be combined to provide greater precision.

3 Analysis

This section describes iteration disambiguation, a dataflow algorithm that distinguishes objects allocated in different iterations of a cycle at compile-time. It does this by marking objects' references with different *ages*. Intuitively, if one considers a loop body as a code segment observed during dynamic execution, the objects created outside of the segment or in previous instances of the segment are distinct from any objects created in the examined segment.

3.1 Example

Figure 2(a) shows the control flow graph for a simple example of an iteration disambiguation opportunity, while Figure 2(b) shows an unrolled version of the loop to clarify the relationship between pointers `a` and `b` within each outer loop iteration. We define two memory objects `A` and `B` by their static allocation sites. Since object `B` lies within a loop, its instances are given subscripts to indicate the iteration of the loop in which they were allocated.

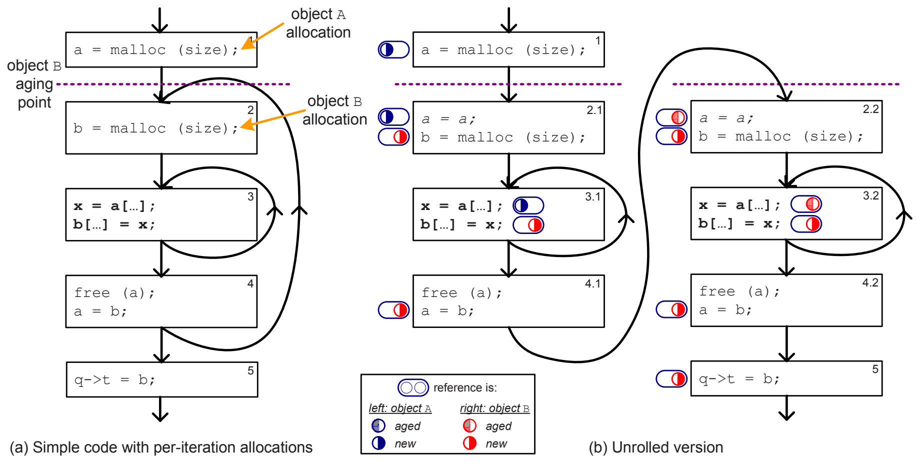


Fig. 2. Iteration disambiguation example

In the first iteration of the loop in Figure 2(b), pointer `a` points to object `A`, while pointer `b` points to object $\texttt{B}_1$, created in block 2.1. These two pointers do not alias within block 3.1 but do in block 4.1. There is a similar situation in block 3.2, except that in this case `a` points to object $\texttt{B}_1$, created in block 2.1, while `b` points to object $\texttt{B}_2$, created in block 2.2. Additional iterations of the loop would be similar to the second iteration. Although `b` aliases with `a` within instances of block 4, the two do not alias within any instance of block 3: `a` points to `A` or $\texttt{B}_{n-1}$, while `b` points to $\texttt{B}_n$. The compiler can determine that loads are independent from stores within the smaller loop in block 3 and can combine this with array index analysis to determine that parallel execution is safe for those loop iterations.

Another way of looking at this relationship is that object $\mathtt{B}_n$ is a separate points-to object from previously created objects. We call the $\mathtt{B}_n$ object *new*, while $\mathtt{B}_{n-1}$ and older objects are lumped together as *aged*. Objects become aged at *aging points*, which are ideally just before a new object is created. Between the *aging points* for an object, or between an aging point and the end of the program, two pointers that point to these two objects cannot alias for any n.

3.2 Algorithm

Our algorithm operates on non-address-taken local variables in the program's control flow graph. A pointer analysis, run prior to the algorithm, annotates variables with sets of objects they may point to. Points-to sets are represented as a set of abstract object IDs, where each ID stands for an unknown number of dynamic objects that exist at runtime. Intuitively, the analysis distinguishes `new` references to an abstract object created within the body of a loop from `aged` ones that must have been created prior to entering the loop or within a prior iteration of the loop. References are `ambiguous` when it becomes unclear whether they refer to `aged` or `new` objects. As long as the ages of two references are distinct and unambiguous, they refer to independent dynamic objects within the scope of an iteration of the most deeply nested loop that contains both references and the allocation for that object. The algorithm is described for a single abstract object with a unique allocation site and calling context. When analyzing multiple objects, analyses of separate abstract objects do not interact. The compiler may choose the objects which are potentially profitable; at this time every heap object allocated within a cycle is analyzed.

The analysis uses *aging points* in the control flow graph to delineate the scope over which a reference is considered `new`. A reference becomes `aged` when it crosses an aging point. Aging points are placed at the entry point of each loop and function containing the allocation. Placing aging points at the beginning of loop iterations ensures that all `new` references within the loop are to objects created in the current loop iteration. New references outside the loop point to objects created in the last loop iteration. Aging at function entry points is necessary for recursive function calls.

Recursive functions require additional consideration. A `new` reference becomes `aged` at the entrance of a recursive function that may allocate it, and could be returned from the function. This is effectively traveling backwards across an aging point, creating a situation where the same reference is both `aged` and `new` in the same region. We avoid this error by conservatively marking an `aged` return value of a recursive function `ambiguous` when it is passed to callers which are also in the recursion. [1]

[1] If a function could have multiple return values, the same situation can occur without recursion. We analyze low-level code that places a function's return data on the stack unless it fits in a single register. Our conservative handling of memory locations yields correct results for multiple return values on the stack.

The example in Figure 2 obtained the `aged` reference via a local variable that was live across the back edge of the loop. However, many programs retain pointers to older objects in lists or other non-local data structures and load them for use. In order to detect these pointers as `aged`, the analysis must determine that the load occurs before any `new` references are stored to non-local memory. Once non-local memory contains a `new` reference, the abstract object is labeled `escaped` until control flow reaches the next aging point. Loads of an `escaped` object return references that are `ambiguous` instead of `aged`. Effectively, non-local memory is an implicit variable that is either ambiguous (`escaped`) or aged (not `escaped`). Escaped reference analysis runs concurrently with propagation of `new` and `aged` reference markings.

Setup. There are several items that need to be performed prior to executing the dataflow algorithm:

1. A heap-sensitive pointer analysis is run to identify dynamically-allocated objects, distinguished by call site and calling context [2], and find which variables may reference the object(s). Figure 3(a) shows a code example with initial flow-insensitive pointer analysis information.
2. SSA notation is constructed for each function in the program, with μ-functions[2] at loop entry points. Although constructing SSA notation prior to pointer analysis can improve the resolution of the input information via partial flow-sensitivity [7], this is not necessary for the algorithm to function correctly.
3. Aging points are marked, for each abstract object, at the entry of each loop or function containing an allocation of the object. This is a bottom-up propagation and visits strongly connected components in the program callgraph only once. In loops, μ-functions that assign references to objects created within the loops are aging points. The input parameters to a function that may create the object are also marked as aging points.
4. A dataflow predicate, representing age, is initialized for each pointed-to object on each pointer assignment, including SSA's μ- and ϕ-functions, and function parameter. The latter are treated as implicit copy operations during interprocedural propagation. We initialize these predicates as follows:
   – Pointer variables which receive the return address of the allocation call are marked with `new` versions of the references.
   – The destination of μ-functions are marked `aged` for an object if the corresponding loop contains an allocation of the object.
   – Destinations of loads that retrieve a reference from memory are optimistically marked `aged`. Unlike the previous two cases, this initialized value may change during propagation of dataflow.
   – All other pointer assignments are marked `unknown`.

   Figure 3(b) shows the results of SSA construction, marking of aging points, and initialization of dataflow predicates for the example in Figure 3(a).

[2] μ-functions were proposed for the Gated Single Assignment notation [1]. Unlike that work, it is not necessary to know which references are propagated from the loop backedges; we use the form simply to mark entries of loops.

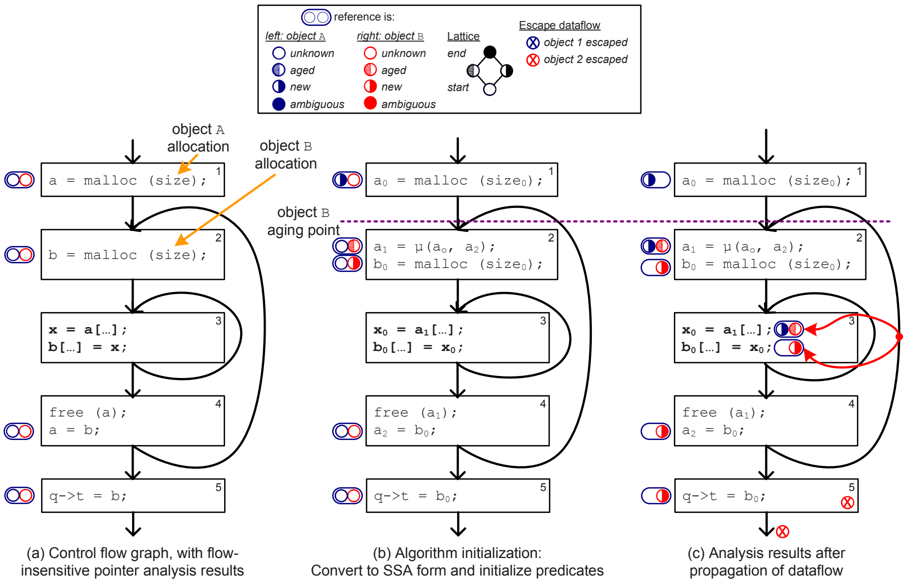


Fig. 3. Iteration disambiguation dataflow

Propagation. Figure 3(c) shows the results of dataflow propagation for the given example. Conceptually, the algorithm marks a reference returned by an allocation routine as `new`. This dataflow predicate propagates forward through the def-use chain until it reaches an aging point, after which it becomes `aged`. This `aged` reference is also propagated forward. If `aged` and `new` references for the same object meet at control flow merges other than aging points, the result becomes `ambiguous`. Separate abstract objects do not interact in any way; for example, propagation from a_0 to a_1 remains `new` for object `A` because there is only an aging point for object `B` for that transition.

As mentioned previously, we desire that the analysis capture cases where older object references are loaded from non-local memory. For this, the analysis identifies regions of the program where `new` or `ambiguous` references have `escaped` to non-local memory. References loaded from memory outside this region are guaranteed to be `aged`, but those loaded within the region are `ambiguous` because a potentially `new` reference has been placed in memory.

Propagation of age markings for object references and detection of escaped references proceed concurrently. All propagation is done in the forward direction. Age markings propagate via SSA def-use chains, while escape markings propagate along intra- and inter-procedural control flow paths. Age propagation uses a three-stage lattice of values, depicted in the legend in Figure 3. The least value of the lattice is `unknown`. The other lattice values are `aged`, `new`, and `ambiguous`. The join or union of an `aged` and a `new` reference is `ambiguous`, which means that the compiler cannot tell the iteration relationship of the reference relative to other object references in the loop. The contents of memory are `ambiguous` where an object has `escaped`; elsewhere, memory only contains `aged` references. Age and escape markings increase monotonically over the course of the analysis. The analysis provides a measure of flow-sensitivity if the base pointer analysis did not support it: references that remain `unknown` at analysis termination are not realizable.

Age markings propagate unchanged through assignment and address arithmetic. The union of the ages is taken for references passed in to ϕ-functions or to input parameters of functions that do not contain an allocation call. `New` and `ambiguous` ages do not propagate through aging points. Function return values are handled differently depending on whether the caller and callee lie in a recursive callgraph cycle. For the nonrecursive case, return values are simply propagated to the caller. For the recursive case, `aged` returns are converted to `ambiguous`. This is necessary to preserve correctness; since the call itself is not an aging point but the callee may contain aging points, a `new` reference passed into the recursive call may be returned as `aged` while other references in the caller to the same dynamic object would remain `new`.

Propagation proceeds analogously for escaped markings. The union of escaped markings is taken at control flow merge points. Escaped markings are not propagated past aging points since all references in memory become `aged` at those points. At the return point of a call where the caller and callee lie in a recursive cycle, memory is conservatively marked `escaped`.

Age and escaped reference markings influence one another through load and store instructions. Stores may cause a `new` or `ambiguous` reference to escape past the bounds that can be tracked via SSA. For our implementation, this occurs when a pointer is stored to a heap object, global variable, or local variable which is address-taken. At that store instruction, the analysis sets an `escaped` marking which propagates forward through the program. The region where this escape dataflow predicate is set is called the *escaped region*.

The analysis optimistically assumes during setup that loaded references are `aged`. If a loaded reference is found to be in the escaped region, the analysis must correct the reference's marking to `ambiguous` and propagate that information forward through def-use chains. Conceptually, the compiler cannot determine whether the reference was the most recent allocation or an earlier one, since a `new` reference has been placed in memory. An example of an escaped reference is shown in the bottom block of Figure 3(c). `Aged` references do not escape because the default state of references loaded from memory is `aged`.

Iteration disambiguation preserves context-sensitivity provided by the base pointer analysis. The calling context is encoded for each object. When the analysis propagates object references returned from functions, the contexts are examined and objects with mismatched contexts are filtered out. The analysis also performs filtering to prevent the `escaped` dataflow from propagating to some unrealizable call paths: references can escape within a function call only if they were created in that function or passed in as an input parameter. Our implementation currently is overly conservative for `escaped` dataflow when a reference is an input parameter which doesn't escape on some call paths. Handling this case requires an analysis to determine which input parameters may escape, and the case has not been prominent in studied applications.

3.3 Properties and Limitations

The iteration disambiguation algorithm explained here is only able to distinguish the object from the current/youngest iteration of a cycle from objects allocated during previous iterations. In other terms, the analysis is k-limited [9] to two ages. The benefit of this is that the analysis is relatively simple and can be formulated as an interprocedural, monotonic dataflow problem. In general only the most recently allocated object is written, while older ones are read-only, so a single age delineation is sufficient to identify parallelism.

The profitability of iteration disambiguation depends on how long a `new` object stays in a local variable and is operated on before escaping. In studied media and simulation applications, `new` references are often created at the top of loop bodies and escape towards the bottom of the loop after significant computation is performed. This exposes the available parallelism within the primary computation loops. However, is not uncommon for a reference to escape immediately on allocation and not be retained in a local variable, which prevents benefit from iteration disambiguation. The common case for this is sub-objects which are linked into an aggregate object, such as separate color planes of an image. Possible methods for resolving these objects are discussed in the next section.

The presented algorithm's effectiveness is also inversely tied to the distance between the aging points and the allocation of the object, since all objects between the aging locations and the allocation are `aged`. These cases might be disambiguable if the aging point were relocated, but this causes more complexity in utilizing the analysis results.

4 Experiments

This section presents empirical results that show that iteration disambiguation generally takes a small amount of time and can identify the distinction between cyclic objects. We covered two categories of benchmarks. For the first, we chose programs from SPEC CPU 2000 and 2006, excluding those from 2000 that have newer versions or equivalents in 2006, and those that the current version of our compiler cannot compile correctly, notably gcc and perl. For the second category, we used several applications from MediaBench I as well as a few independent ones. Our intent with the broad selection is to show that the analysis can disambiguate references in application domains other than media and scientific simulation. We use Fulcra [12] as our base pointer analysis.

4.1 Analysis Statistics

Figure 4 shows analysis statistics for the benchmark programs analyzed. The bars represent iteration disambiguation's time to analyze, annotate, and count statistics

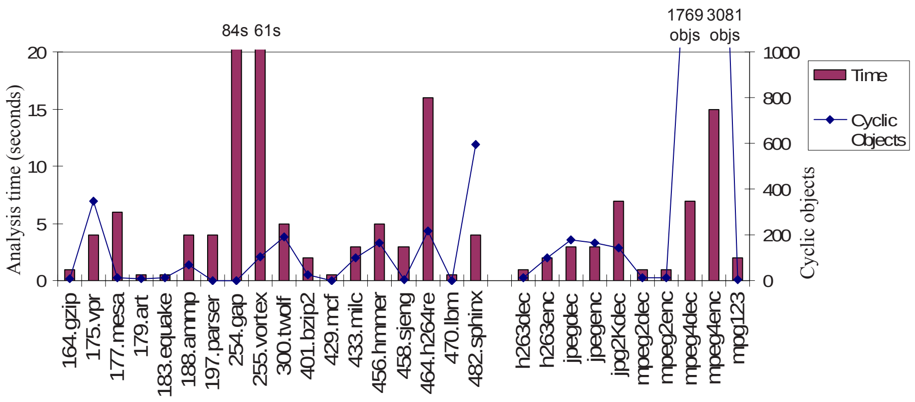


Fig. 4. Iteration disambiguation analysis time and object count

in seconds, on a 1.8 GHz Athlon 64. The dots connected by lines represent the number of distinct cyclic objects, distinguishable by call site and calling contexts. The number of heap objects is dependent on the degree of cloning (replication per call site and path) [13] performed by the base pointer analysis. For example, the MPEG-4 decoder and encoder applications have a large number of nested allocation functions which create the large number of objects seen in the figure. The "lowest-level" objects tend to be replicated the most, which affects some of our metrics.

In general the analysis is fast; for most programs, which have few cyclic objects, the analysis takes only a few seconds. Even for programs with many cyclic objects, such as 464.h264ref and mpeg4enc, the analysis runs within 16 seconds. The majority of analysis time is spent in the setup phase and the time for propagation of age markings and escaped dataflow is usually insignificant. The primary outliers are two SPEC CPU2000 benchmarks, 254.gap and 255.vortex. These benchmarks are over twice as large as the majority of the benchmarks in the program, with a correspondingly higher setup time. The larger size also increases the amount of code that escaped reference dataflow must propagate through. Finally, they have an unusually high number of references relative to the size of the codes. Unlike the other benchmarks, the time for age propagation and escaped reference dataflow is on the same order as setup time.

4.2 Object Classifications

There are two special object classifications which are exposed by the analysis. First, some objects allocated within cycles are used as temporary storage and are deallocated within the same iteration. These cases are interesting because they represent privatization opportunies. In iteration disambiguation, these objects are recognizable since only `new` references are used to load data from or store data to them. The percentage of only-new objects is shown in Figure 5. Benchmarks that have no cyclic objects are omitted.

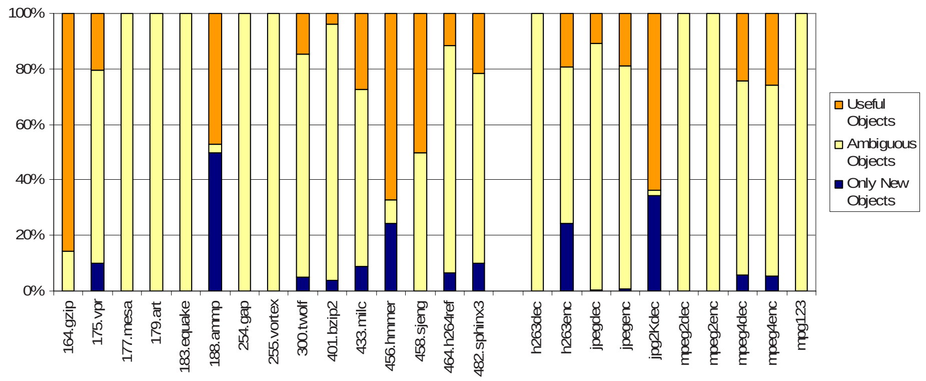


Fig. 5. Iteration disambiguation results: proportion of only new and ambiguous objects

Second, for some cyclic objects, there are either no `new` markings or no `aged` markings. For these objects, iteration disambiguation has no useful effect. We term these *ambiguous objects*. In programs with inherent parallelism, these objects are commonly multidimensional arrays and sub-structures, which require complementary analyses when detecting parallelism. As mentioned previously, these lower-level objects are a significant portion of the total object count due to heap cloning, and thus increase the apparent number of ambiguous objects beyond a static count of call sites when heap cloning has an effect. Even excluding this effect, direct inspection of several of the applications has shown that the majority of the heap objects are ambiguous.

4.3 Analysis Results

Prior to discussing the analysis results, we break the categories of `aged` and `ambiguous` references into subcategories to gain a better understanding of the results and program properties. They are:

– **Loop Aged**: The reference was passed via a local variable across the backedge of a loop, or entered a recursive function that may allocate the object.
– **Loaded Aged**: The reference's source is loaded from memory in a region where a `new` or `ambiguous` reference has not escaped.
– **Merge Ambiguous**: The age of the reference is ambiguous due to a control flow or procedure call merge in which new and aged references of an object merge via dataflow, such as for conditional allocations.
– **Escape Ambiguous**: The age of the reference is ambiguous because it was obtained via a load in a code region where a `new` or `ambiguous` reference has escaped. We also include aged references returned from a recursive function to callers within the recursion in this category.
– **Combination Ambiguous**: This represents a merge of an escape-ambiguous reference with other types of references.

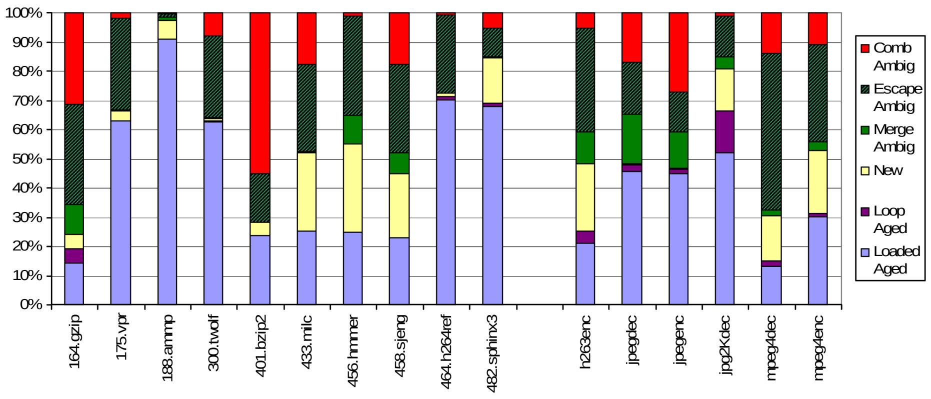


Fig. 6. Iteration disambiguation results: percentages of reference types

Figure 6 shows statistics of the results of iteration disambiguation. Results are shown as a percentage of the total static, heap-referencing memory operations, in an assembly-like representation, for each program. When a memory operation may access multiple cyclic objects, an equal fraction is assigned to each object. References to objects that are only `new` have been omitted because they inflate the apparent utility of the analysis. A significant percentage of both `new` and `aged` references indicates likely independence of operations within a loop body. Applications that have no useful cyclic objects have been omitted.

Although a more appropriate test of this analysis would be to show the amount of parallelism exposed by the analysis, we do not attempt this for this work. The objects of interest in many time-sliced applications are children objects of the top-level objects that iteration disambiguation can operate on, and are identified as ambiguous objects. We currently do not have an analysis to prove that children are unique to a parent object, so the amount of extractable parallelism is small. In the future we hope to show the difference in application performance when the additional analyses are integrated into our framework.

The high percentage of escape and combination ambiguous references indicate that many static operations on heap objects use references that have been stored to and then loaded back from memory prior to an aging point. This is expected for programs that build up large aggregate structures and then operate on them, such as the larger SPEC CPU benchmarks. Despite the fact that ambiguous objects tend to make up the majority of objects, they do not always dominate the references because of fractional counting per memory operation. We observed a tendency for operations to be one type of reference, because the objects they reference usually have similar usage patterns.

Media programs have a high percentage of ambiguous references because the majority of operations work on data substructures linked to top-level structures. As previously mentioned, escaped references are often multidimensional arrays and can be addressed with the appropriate analysis. Another case that is missed by iteration disambiguation is sub-structures of a cyclic object linked into an aggregate structure. We are currently developing a complementary analysis to address this shortcoming.

One interesting case is 464.h264ref, which is a video encoder application and thus expected to do well with iteration disambiguation. However, it has a smaller percentage of `new` references than most applications. The reason is the common use of an exit-upon-error function, which both calls and is called by many of the allocation and free functions used in the application. This creates a large recursion in the call graph, which has the effect of aging `new` references rapidly. In addition, we discovered that approximately half of the loaded `aged` references become escape `ambiguous` if the analysis does not prevent dataflow propagation through unrealizable paths, such as calls to `exit()`.

5 Conclusions and Future Work

This paper discusses iteration disambiguation, an analysis that distinguishes high-level, dynamically-allocated, cyclic objects in programs. This cyclic relationship is common in media and simulation applications, and the appropriate analysis is necessary for automatic detection and extraction of parallelism. We show that we can disambiguate a significant percentage of references in a subset of the presented applications. We also explain some of the reasons why the analysis was not able to disambiguate more references in cases where we would expect a compiler to be able to identify parallelism.

For future work, we will be developing complementary analyses which will enable a compiler to automatically identify parallelism within programs that are amenable to parallel execution. These include array analyses, analyses that identify structure relationships such as trees, and value flow and constraint analyses.

Acknowledgment

This work would not have been possible without the work performed by Erik Nystrom and Sara Sadeghi Baghsorkhi on the Fulcra pointer analysis. We thank Bolei Guo for his advice and the anonymous reviewers for their feedback. We also acknowledge the support of the Gigascale Systems Research Center, funded under the Focus Center Research Program, a Semiconductor Research Corporation program.

References

1. Ballance, R., Maccabe, A., Ottenstein, K.: The Program Dependence Web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In: Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation, pp. 257–271 (1990)
2. Choi, J.D., Burke, M.G., Carini, P.: Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In: Proceedings of the 20th ACM Symposium on Principles of Programming Languages, pp. 232–245 (January 1993)

3. Deutsch, A.: A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In: Proceedings of the 1992 International Conference on Computer Languages, pp. 2–13 (April 1992)
4. Ghiya, R., Hendren, L.J.: Connection analysis: A practical interprocedural heap analysis for C. In: Proceedings of the Eighth Workshop on Languages and Compilers for Parallel Computing, pp. 515–533 (August 1995)
5. Ghiya, R., Hendren, L.J.: Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In: Proceedings of the 23rd ACM Symposium on Principles of Programming Languages, pp. 1–15 (1996)
6. Guo, B., Vachharajani, N., August, D.I.: Shape analysis with inductive recursion synthesis. In: Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (June 2007)
7. Hasti, R., Horwitz, S.: Using static single assignment form to improve owinsensitive pointer analysis. In: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, pp. 97–105 (June 1998)
8. Hind, M.: Pointer analysis: Haven't we solved this problem yet? In: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, pp. 54–61 (2001)
9. Jones, N.D., Muchnick, S.S.: Flow analysis and optimization of LISP-like structures. In: Proceedings of the 6th ACM SIGPLAN Symposium on Principles of Programming Languages, pp. 244–256 (1981)
10. Landi, W., Ryder, B.G.: A safe approximate algorithm for interprocedural pointer aliasing. In: Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation, pp. 235–248 (June 1992)
11. MPEG Industry Forum, `http://www.mpegif.org/`
12. Nystrom, E.M.: FULCRA Pointer Analysis Framework. PhD thesis, University of Illinois at Urbana-Champaign (2005)
13. Nystrom, E.M., Kim, H.-S., Hwu, W.W.: Importance of heap specialization in pointer analysis. In: Proceedings of ACM-SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, pp. 43–48 (June 2004)
14. Ryoo, S., Ueng, S.-Z., Rodrigues, C.I., Kidd, R.E., Frank, M.I., Hwu, W.W.: Automatic discovery of coarse-grained parallelism in media applications. Transactions on High-Performance Embedded Architectures and Compilers 1(1), 194–213 (2007)
15. Sagiv, M., Reps, T., Wilhelm, R.: Solving shape-analysis problems in languages with destructive updating. In: Proceedings of the ACM Symposium on Programming Languages, pp. 16–31 (January 1996)
16. Venet, A.: A scalable nonuniform pointer analysis for embedded programs. In: Proceedings of the International Static Analysis Symposium, pp. 149–164 (2004)
17. Wu, P., Feautrier, P., Padua, D., Sura, Z.: Instance-wise points-to analysis for loop-based dependence testing. In: Proceedings of the 16th International Conference on Supercomputing, pp. 262–273 (2002)

A Novel Asynchronous Software Cache Implementation for the Cell-BE Processor

Jairo Balart[1], Marc Gonzalez[1], Xavier Martorell[1], Eduard Ayguade[1], Zehra Sura[2], Tong Chen[2], Tao Zhang[2], Kevin O'Brien[2], and Kathryn O'Brien[2]

[1] Barcelona Supercomputing Center (BSC), Technical University of Catalunya (UPC)
[2] IBM TJ Watson Research Center
{jairo.balart,marc.gonzalez,xavier.martorell, eduard.ayguade}@bsc.es,
{zsura,chentong,taozhang,caomhin,kmob}@us.ibm.com

Abstract. This paper describes the implementation of a runtime library for asynchronous communication in the Cell BE processor. The runtime library implementation provides with several services that allow the compiler to generate code, maximizing the chances for overlapping communication and computation. The library implementation is organized as a Software Cache and the main services correspond to mechanisms for data look up, data placement and replacement, data write back, memory synchronization and address translation. The implementation guarantees that all those services can be totally uncoupled when dealing with memory references. Therefore this provides opportunities to the compiler to organize the generated code in order to overlap as much as possible computation with communication. The paper also describes the necessary mechanism to overlap the communication related to write back operations with actual computation. The paper includes the description of the compiler basic algorithms and optimizations for code generation. The system is evaluated measuring bandwidth and global updates ratios, with two benchmarks from the HPCC benchmark suite: Stream and Random Access.

1 Introduction

In a system where software is responsible for data transfers between certain memory regions, it is desirable to assist the programmer by automatically managing some or all of these transfers in system software. For asynchronous data transfers, it is possible to overlap the memory access time with computation time by initiating the data transfer request in advance, i.e. before computation reaches the point when it needs to use the data requested. The placement of such memory access calls in the code is important since it can change the amount of overlap between data communication and computation, and thus affect the overall performance of the application. In this work, we target a Cell BE system to explore our approach to automatically managing asynchronous data transfers. Our technique implements a software caching mechanism that works differently from traditional hardware caching mechanisms, with the goal being to facilitate the decoupling of the multiple steps involved in a memory access (including address calculation, cache placement, and data transfer) as well as the actual use of the data. Software caching is not a novel proposal since it has been extensively used in specific domains, like embedded processors [4][5][6].

Our target platform, the Cell BE architecture [2], has nine processing cores on a single chip: one 64-bit Power Processing Element (PPE core) and eight Synergistic Processing Elements (SPE cores) that use 18-bit addresses to access a 256K Local Store. The PPE core accesses system memory using a traditional cache-coherent memory hierarchy. The SPE cores access system memory via a DMA engine connected to a high bandwidth bus, relying on software to explicitly initiate DMA requests for data transfer. The DMA engine can support up to 16 concurrent requests of up to 16K, and bandwidth between the DMA engine and the bus is 8 bytes per cycle in each direction. Each SPE uses its Local Store to buffer data transferred to and from system memory. The bus interface allows issuing asynchronous DMA transfer requests, and provides synchronization calls to check or wait for previously issued DMA requests to complete.

The rest of this paper is organized as follows. In Section 2, we motivate the use of a novel software cache organization for automatically managing asynchronous data transfers. In Section 3, we detail the structure and implementation of this software cache mechanism. In Section 4, we describe the compiler support needed to enable effective use of the runtime software cache services. In Section 5, we evaluate basic performance of our software caching technique using the Stream and Random Access benchmarks from the HPCC benchmark suite. In Section 6 we present some concluding remarks.

2 Motivation

The particular memory model in the Cell BE processor poses several difficulties for generating efficient code for the SPEs. The fact that each SPE owns a proper address space within the Local Storage, plus the limitation on its size, 256Kb shared by data and code, causes the performance being very sensible on how the communications are scheduled along the computation. Overlapping computation with communication becomes a crucial optimization.

When the access patterns in the computation can be easily predicted, static buffers can be introduced by the compiler, double-buffering techniques can be exploited at runtime, usually involving loop tiling techniques [1][7]. In the presence of pointer–based accesses, the compiler is no longer able to transform the code in order to overlap communication and computation. Usually, this kind of access is treated by a runtime library implementing a software cache [1]. The resulting code is difficult to be efficient as every memory reference in the code has to be monitored in order to ensure that the data is present in the Local Store, before any access to it takes place. This is usually implemented through the instrumentation of every memory reference with a runtime call responsible for the monitoring, where many checks have to occur. A general protocol to treat a single memory reference could include the following steps:

1. Check if the data is already present in local storage
2. In case not present, decide where to place it and ...
3. If out of space, decide what to send out from Local Storage

4. If necessary, perform DMA operations
5. If necessary synchronize with DMA
6. Translate from virtual address space to Local Storage address space
7. Perform memory access

Under that execution model, the chances for overlapping computation with communication are quite limited. Besides, the memory references instrumentation incurs in unacceptable overheads. The motivation of this paper is to describe what should be the main features within a software cache implementation that maximizes the chances for overlapping computation and communication, and minimizes overhead related to the memory references instrumentation.

Following the previous scheme, the overlap of communication with computation it can only be implemented by uncoupling the DMA synchronization (step 5) from the previous runtime checks (steps 1 to 4). If the runtime were to support such uncoupling, then it could be possible to reorganize the code, placing some amount of computation between step 4 and step 5 of every reference. Notice that this optimization is conditioned by the computation, in the sense that it might happen that data dependences do not allow the code reorganization. Although that, decoupling steps 4 and 5 still can offer some important benefits. It is also possible to mix the 1, 2, 3, and 4 steps of two or more memory references and group all the DMA synchronization in one single step. That would translate on some overlapping between cache management code and data communication, reducing the overhead impact. But such overlapping needs of some specific features within the implementation of steps 1, 2 and 3. It is necessary that no conflict appears between steps 1, 2 and 3 of every memory reference treated before the synchronization step. That is, the placement and replacement mechanisms must not assign the same cache line for two different memory references. This is one point of motivation of the work in this paper: the implementation of a software cache that enhances the chances for the overlapping of computation (whether it is cache control code or application code) and data communication, by uncoupling steps 4 and 5 and reducing the cache conflicts to capacity conflicts.

Because of the limited size of the Local Storage, it is necessary to provide the cache implementation with a write back mechanism to send out data to main memory. The write back mechanism involves a DMA operation moving data from the Local Storage to main memory, and requires the SPE to synchronize with the DMA engine before the flushed cache line is being reused. Deciding the moment to initiate the DMA operation becomes an important issue to increase performance. If the write back mechanism is invoked just when a modified cache line has to be replaced, then the SPE is going to be blocked until the associated DMA operation ends. The implementation described in this paper introduces two mechanisms to minimize as much as possible the number of lost cycles waiting for a DMA operation to complete (related to a flush operation). First, a mechanism to foresee future flush operations, based on information about what cache lines are referenced by the code, and detecting the precise moment where a cache line becomes unreferenced. Second, a specific replacement policy that delays as much as possible the next assignment for a flushed cache line, thus giving time to the flush operation to complete, and avoid lost cycles dedicated to synchronization at the moment of reuse.

3 Software Cache Implementation

The software cache is described according to the cache parameters, cache structures and the main services: look up, placement/replacement policies, write back, communication/synchronization and address translation.

3.1 Cache Parameters

The main cache parameters are the following: capacity, size of cache line and associativity level. For the rest of this document C stands for capacity, L stands for the cache line size, S stands for the level of associativity and N=C/L stands for the number of cache lines.

3.2 Cache Structures

The cache is composed mainly by three structures. Two list-based structures, where the cache lines can be placed depending on their state and attributes value. These are the *Directory* and the *Unused Cache Lines* lists. A third structure under a table shape, basically used for look up and translation operations: the *Look Up and Translating* table.

- The *Directory* list holds all the cache lines that are resident in the cache.
- The *Unused Cache Lines* list holds all cache lines that are no longer in use by the computation. The notion of being under use is defined by the existence of any memory reference in the computation that references the in-use cache line. The cache implementation is able to keep track of what cache lines are being referenced, and what are not.
- The *Look Up and Translating* table holds information for optimizing the look up mechanism and for implementing the translation from the virtual address space to the Local Storage address space.

3.2.1 Directory

The *Directory* is composed of S lists. Cache lines in the *Directory* are stored in a double –linked list form. There is no limitation on the number of cache lines that can be placed in any of the S lists. That makes the cache implementation a full-associative cache. Basically the S lists are used as a hash structure to speed up the look up process.

3.2.2 Unused Cache Lines List

This list holds the cache lines that were previously used by the computation, but that at a given moment they were no longer in use. The main role for this structure is related to the placement/replacement policy. Cache lines placed in this list become the immediate candidates for replacement, thus placement for other incoming cache lines required by the computation. The cache lines are stored in a double-linked list form.

3.2.3 Look up and Translating Table

This structure is organized as a table, where each row is assigned to a particular memory reference in the computation. A row contains three values used for the look up and translation mechanisms: the base address of the cache line in the Local Storage address space, the base address of the correspondent cache line in the virtual address space and a pointer to the structure representing the cache being used by the memory reference.

3.2.4 Cache Line State and Attributes

For every cache line, the implementation records information about the cache line state and other attributes, necessary to control the placement/replacement, write back, look up, and translation mechanisms.

The state of a cache line is determined by the fact any memory reference in the computation referencing the cache line. The implementation keeps track of what cache lines are under use, by maintaining a reference counter associated to each cache line. The reference counter is incremented/decremented appropriately during the *Look Up* mechanism. Therefore, the state of a cache line can take two different values: USED or UNUSED. Besides the cache line state, there are other attributes:

- CLEAN: the cache line has been only used for READ memory operations. The data stored in the cache line has not been modified.
- DIRTY: the cache line has been used for WRITE and/or READ memory operations. The data stored in the cache line has been modified.
- FLUSHED: the cache line has already been flushed to main memory.
- LOCKED: the cache line is excluded from the replacement policy, which means that a cache line holding this attribute can not be replaced.
- PARTITIONED: the data transfer from/to main memory involves a different amount of data than the actual cache line size. The total number of bytes to be transferred is obtained by dividing the cache line size by a factor of 32.

The implementation also records the mapping between the cache line in the Local Storage, and its associated cache line in virtual memory.

3.3 Look up

The *Look Up* mechanism is divided in two different phases. First phase takes place within the computation code, second phase occurs inside the cache runtime system. For the first phase of look up, it is necessary some coordination with the compiler support. For each memory reference the implementation keeps track about the base address for the cache line being accessed. This information is stored in the *Look Up and Translating* table. Each time a new instance of a memory reference occurs, the implementation checks if the referenced cache line has changed. If this happens, then the second phase for the look up is invoked. Detecting if the cache line has changed is as simple as performing an AND operation between the memory address generated in the memory reference, and a particular mask value (in C syntax: ~(L-1)), plus a comparison with the value in the *Look Up and Translating* table. It is under the compiler responsibility to assign an entry in the *Look Up and Translating* table for each memory reference in the code. Section 4.2 is giving the detailed description on how this is implemented.

The second phase of the *Look Up* mechanism accesses the cache *Directory* looking for the new required cache line. Only one of the S lists has to be selected to perform the search. This is done through a hash function applied to the base address of the cache line. The implementation ignores the offset bits, and takes all other most significant bits. Then applies an S-modulo operation and determines one of the S lists. The *Look Up* continues with the list traversal, and if the cache line is found, a hit is reported. In case not, the placement/replacement mechanisms are invoked, and the necessary DMA operations are programmed.

During the *Look Up* process, the reference counters for the two cache lines that are going to be involved are incremented/decremented. For the cache line that is no longer referenced by the memory reference, the counter is decremented. For the new referenced cache line, the counter is incremented, no matter the *Look Up* ended with a hit or miss.

At the end of the *Look Up* process the *Look Up and Translating* table is updated. The row assigned to the memory reference the *Look Up* operation was treating is appropriately filled: base address of the cache line in the Local Storage, base address of the cache line in virtual memory and a pointer to the structure representing the cache line.

3.4 Write Back

The Write Back mechanism only applies for modified cache lines, that is, those lines that hold the DIRTY attribute. The write back is activated when the reference counter of a modified cache line reaches the zero value. This event is interpreted by the implementation as a hint of future possible uses of the cache line. Particularly, the event is interpreted as if the cache line is not going to be referenced by the computation up to its completion. Therefore, this point becomes a good opportunity to go in advance to the needs of the computation and program the flush of the cache line, under an asynchronous scheme. Notice that this is giving, but not ensuring, time to the implementation to overlap communication and computation. Of course, it is necessary at some point to synchronize with the DMA operation. In order to do so, the implementation records the TAG used in the DMA operation, and delays the synchronization until the next use of the cache line, when ever the replacement policy determines the next reuse to happen.

3.5 Placement / Replacement

The Placement/Replacement mechanisms are executed during the second phase of *Look Up*. The replacement policy relies on the reference counter and the *Unused Cache Lines* list. When the cache line reference counter equals zero, the cache line is placed on the *Unused Cache Lines* list as the LAST of the list, and as stated in previous section, if the line was modified, a flush operation is immediately programmed. Notice that the cache line is not extracted from the *Directory*.

The *Unused Cache Lines* list contains the cache lines candidates for replacement actions. When new data has to be brought in the cache, a cache line has to be selected.

If the *Unused Cache Lines* list is not empty, the implementation selects the FIRST in the list. If the line is holding the FLUSHING attribute, the tag that was recorded during write back execution is used to synchronize with the DMA engine. After that, a DMA operation is programmed under an asynchronous scheme to bring in the data, relying on the compiler for placing in the computation code the necessary synchronization statement. Notice that selecting the FIRST element in the list, while unused cache lines are placed as LAST, is what separates as much as possible the DMA operation associated to a flushed cache line, and its next reuse. Hence, delaying as much as possible the execution of the necessary synchronization with the DMA engine and avoiding unnecessary stalls in the SPE.

If the *Unused cache Lines* list is empty, then the replacement policy traverses the *Directory* from set 0 to S-1, and selects the line that first entered in the cache. This is implemented through the assignment of a number that is incremented each time a cache line is brought in. The minimum number within all resident cache lines determines the cache line to be replaced. If the replaced line was modified, the line is flushed to main memory under a synchronous scheme. After that, the data is brought in through an asynchronous DMA operation, and relying on the compiler to introduce the necessary synchronization statement. Notice that an appropriate relation between the number of cache lines and the number of memory references might perfectly avoid this kind of replacement, since it can be ensured that the list of unused cache lines is never going to be empty (see section 4.6).

Initially, all cache lines are stored in both the *Directory* and the *Unused Cache Lines* list, with the counter reference equaling zero.

3.6 Communications and Synchronization

The implementation distinguishes between DMA transfers related to write back operations and DMA transfers responsible for bringing data into the cache. For the former case, a set of 15 tags are reserved, for the latter another different 15 tags. For both cases tags are assigned in a round robin fashion, which means after 15 DMA operations tags start being reused.

All DMA operations assigned to the same tag, are executed always one after the other. This is achieved through the use of fenced DMA operations that forbids the memory flow controller to reorder any DMA operations associated to the same tag. This becomes necessary for treating the following situation: suppose a modified cache line is no longer in use, so it is flushed to main memory, and placed in the *Unused Cache Lines* list. Then the code being executed references again the data in the cache line, and since it was not extracted from the *Directory*, no miss is produced, but it is necessary to extract the cache line from the *Unused Cache Lines* list. The cache line might or might not be modified, but at some point the cache line will be no longer in use. In the case the cache line was modified it will be flushed again. It is mandatory for memory consistency that the two flush operations get never reordered. To ensure that, the implementation reuses the same tag for both flush operations, and introduces a “memory fence” between them.

All DMA operations are always programmed under an asynchronous scheme, unless those associated to a replacement that found empty the *Unused Cache Lines* list. Those related to flush operations, synchronize at the next reuse of the flushed cache line. Those related to bring data into the cache get synchronized by specific statements introduced by the compiler. It is important to mention that this is what allows the compiler to try to maximize the overlap between communication and communication. Section 4 describes the necessary compiler support to achieve the communication/computation overlapping.

3.7 Address Translation

To perform the translation from the virtual address space to the Local Storage address space the data in the *Look Up and Translating* table is enough. Each memory reference has been assigned with a row in the *Look Up and Translating* table. In that row, the base address for the cache line in the Local Storage can be obtained. Translation is as simple as computing the offset of the access and add the offset to the base address of the cache line in the Local Storage. The offset computation can be done through an AND operation between the generated address in the memory reference and a bit mask according to the size of the cache line (e.g: ~(L-1)).

4 Compiler Code Generation

This section describes the compiler support and code generation for transforming programs to SPE executables relying on the software cache described in the previous section. In this paper we describe the compiler support that is required to target the execution of loops.

4.1 Basic Runtime Services

This section describes the main runtime available services which the compiler should target while generating code.

- _LOOKUP: runtime service performing the phase 1 in the *Look Up* mechanism.
- _MMAP: runtime service executing phase 2 in the *Look Up* mechanism. In case a miss is produced, then the placement/replacement mechanisms are executed, the reference counters are incremented/decremented, and the all necessary DMA operations are performed asynchronously. In case the replacement algorithm indicates the use of a previously flushed cache line, synchronization with the DMA engine occurs.
- _MEM_BARRIER: runtime service that forces the synchronization with the DMA engine. It is a blocking runtime service.
- _LD, _ST: runtime services responsible for the address translation between the virtual address space and the Local Storage address space. Include arithmetic pointer operations such as the computation of the offset in the access to the cache line base address in virtual memory, and the computation of the actual Local Storage address by adding the offset to the base address of the cache line in the Local Storage.

4.2 Code Generation

This section describes the basic algorithms and optimizations related to code generation.

4.2.1 Assign Identifiers to Memory References

The first step for the compiler is to assign a numerical identifier to each different memory reference in the code. This identifier is going to be used at runtime to link each memory reference to the runtime structure supporting the *Look Up* (phase one), and the translating mechanisms. The runtime structure corresponds to one entry in the*Look Up and Translating* table.

```
for (i=0;i<NUM_ITERS;i++) {
  v1[i] = v2[i];
  v3[v1[i]]++;
}
```

Fig. 1. Example of C code for code generation

For the example shown in Figure 1, three different memory references can be distinguished: *v1[]* , *v2[]* and *v3[]* . The compiler would fro example associate identifiers 0, 1, 2 to memory references to *v1[]* , *v2[]* and *v3[]* respectively.

```
for (i=0;i<NUM_ITERS;i++) {
   if (_LOOKUP(0, ,&v2[i],...)) {
     _MMAP(0,&v2[i],...);
     _MEM_BARRIER(0);
  }
  if (_LOOKUP(1, ,&v1[i],...)) {
    _MMAP(1,&v1[i],...);
    _MEM_BARRIER(1);
  }
  _LD(0,&v2[i],_int_tmp00);
  _ST(1,&v1[i],_int_tmp00);
  if (_LOOKUP(2, ,&v3[_int_tmp00],...)) {
    _MMAP(2,&v3[_int_tmp00],...);
    _MEM_BARRIER(2);
  }
  _LD(2,&v3[_int_tmp00],_int_tmp01);
  _int_tmp01++;
  _ST(2,&v3[_int_tmp00],_int_tmp01);
}
```

Fig. 2. Initial code transformation

4.2.2 Basic Code Generation

For every memory reference, the compiler has to inject code to check if the data needed by the computation is in the Local Storage. The compiler injects a _LOOKUP operation for every memory reference, and a conditional statement depending on the output of the _LOOKUP operation. Figure 2 shows the transformed code for the example in figure 1. All _MMAP operations are controlled by a _LOOKUP operation, relying on the runtime structures pointed out by the assigned identifier according to what has been described in the previous section. Right at the end on the conditional branch, the compiler injects a _MEM_BARRIER operation that enforces the synchronization with the DMA engine.

```
_lb_01 = 0; _ub_01 = NELEM;
_work_01 = (_lb_01 < _ub_01);
while (_work_01) {
  _start_01 = _lb_01;
  _LOOKUP(0, ,&v2[i],...,_lookpu_01);
  if (_lookup_01) _MMAP(0, &v2[_start_01], ..., LOCK);
  _LOOKUP(1, ,&v1[i],...,_lookup_01);
  if (_lookup_01) _MMAP(1, &v1[_start_01], ..., LOCK);
  _next_iters_01 = LS_PAGE_SIZE;
  _NEXT_MISS(0, &v2[_start_01], float, sizeof(float), _next_iters_01);
  _NEXT_MISS(1, &v1[_start_01], float, sizeof(float), _next_iters_01);
  _end_01 = _start_01 + _next_iters_01;
  if (_end_01>_ub_01) _end_01 = _ub_01;
  _lb_01 = _end_01;
  _work_01 = (_lb_01 < _ub_01);
  _MEM_BARRIER();
  for (int i = _start_01; i < _end_01; i=i+1) {
     _LD(0,&v2[i],_int_tmp00);
     _ST(1,&v1[i],_int_tmp00);
     if (_LOOKUP(2, &v3[_int_tmp00],...)) {
        _MMAP(2,&v3[_int_tmp00],...);
        _MEM_BARRIER(2);
     }
     _LD(2,&v3[_int_tmp00],_int_tmp01);
     _int_tmp01++;
     _ST(2,&v3[_int_tmp00],_int_tmp01);
   }
}
```

Fig. 3. Code transformation for stride accesses

This preliminary version of the transformed code does not allow any overlap between computation and communication. It contains unnecessary conditional statements that for sure are not going to be optimal. Besides, it does not take into account the different type of accesses in the code, distinguishing between strided accesses and pointer-based accesses. But before describing any optimization technique, it is necessary to outline what are the limitations that condition the compiler transformations. Since the main target of the compiler is to enhance the overlapping of computation (whether cache control code or original computation in the code) it is reasonable to try to reorganize the preliminary code in order to group _MMAP operations, making them to be executed at runtime right one after the other. Notice that such grouping makes all the communication performed within a _MMAP operation, be overlapped with the execution of the following _MMAP operations. In the example, the if statements corresponding to the accesses to *v1[i]* and *v2[i]* could be joined. One if statement should include the two _MMAP operations, and only one _MEM_BARRIER. Generally, the compiler is only limited by the fact that grouping the _MMAP operations must be done taking to account the possibility of conflicts within the grouped _MMAPs. A conflict may appear along the execution of several _MMAP operations if two of them require the same cache line to bring data in the Local Storage. A conflict is not acceptable to appear before the data of the conflicting _MMAP operations has been accessed. That is, between the execution of a particular _MMAP operation and the _LD/_ST operations with the same identifier, it is not acceptable to place a number of _MMAP operations that can cause a conflict. Since the cache implementation follows a full-associative scheme, conflicts may only appear as capacity conflicts. This determines the limits on the grouping: the compiler can not group _MMAP operations if doing so is causing that between a _LD/_ST operation and the corresponding _MMAP operation (indicated by the identifier associated to _MMAP and

_LD/_ST operations) N _MMAP operations are executed, where N stands for the number of cache lines. Formally, we define the distance of a _MMAP operation as *the maximum number of _MMAP operations between the _MMAP and the _LD/_ST operations with the same identifier*. The compiler is now free of reorganizing the code, grouping _MMAP operations, as long as it keeps every _MMAP distance in the range of [0..N].

4.3 Optimization for Strided Accesses

Strided accesses offer the possibility of reducing the number of the _MMAP operations that need to be performed during the execution of the loop. The basis for such optimization is that the runtime can be provided with a service that computes how many accesses are going to be produced along a cache line, given the starting address and the stride. This information can be used to partition the iteration space in different chunks, defining the initial iteration of each chunk, a change of cache line (actually a miss) in a strided memory access.

Figure 3 shows the compiler code for the example code in Figure 1. Notice that the original loop, has been embedded in an outer *while* loop. The *while* loop iterates along the chunks of iterations, and the inner loop iterates along the actual iteration space. The use of the runtime service _NEXT_MISS computes the number of iterations that can be performed without having a miss on that access, given an initial address and a stride. For every strided access, the _NEXT_MISS service is invoked, and the minimum of these values defines the number of iterations for the next chunk. In the example, the two stride accesses are treated with two _MMAP operations that are going to overlap the communication of the first one with the cache control code of the second one.

Notice the attribute LOCK provided to the runtime system _MMAP, that ensures that the mapped cache line is going to be excluded from the replacement policies. This causes the runtime to treat the memory references in the inner loop, with a different distance boundary, since now the compiler has to assume 2 less available cache lines in the overall cache capacity. A _MEM_BARRIER is placed right before the inner loop execution ensuring that the data is resent before the next chunk of iterations is executed. This synchronization only involves incoming data to the Local Storage. Write back operations executed along the _MMAP runtime service corresponding to the *v1[i]* access operation, are synchronized whenever the cache lines associated to this access are being reused.

4.4 Optimization for Non-strided Accesses

Non-strided accesses become an important source of overhead, since they do not usually spatial locality. Therefore, overlapping computation and communication for this type of access should be highly desirable. Figure 4 shows the compiler transformation for this kind of access, corresponding to the *v3[v1[i]]* access in the example in Figure 1. Only the innermost loop where the non-stride access is placed is showed. The loop has been unrolled 2 times, offering the possibility of grouping the 2 _MMAP operations associated to that access. The 2 factor has been only used as example, since the limit on the unrolling factor is going to be determined by the number

```
for (int i = _start_01; i < _end_01; i=i+2) {
  _LD(0,&v2[i],_int_tmp00);
  _ST(1,&v1[i],_int_tmp00);
  _LD(0,&v2[i+1],_int_tmp02);
  _ST(1,&v1[i+1],_int_tmp02);
  _LOOKUP(2, &v3[_int_tmp00],..., _lookup_01)
  _LOOKUP(2, &v3[_int_tmp02],..., _lookup_01)
  if (_look_up_01) {
    _MMAP(2,&v3[_int_tmp00],...);
    _MMAP(3,&v3[_int_tmp02],...);
    _MEM_BARRIER(2);
  }
  _LD(2,&v3[_int_tmp00],_int_tmp01);
  _int_tmp01++;
  _ST(2,&v3[_int_tmp00],_int_tmp01);

  _LD(3,&v3[_int_tmp02],_int_tmp03);
  _int_tmp03++;
  _ST(3,&v3[_int_tmp02],_int_tmp03);
 }
```

Fig. 4. Code transformation for non-stride accesses

of cache lines, minus 2 (two cache lines have been locked for *v1* and *v2* accesses), as the distance boundary has to be preserved for all _MMAP operations. Notice that the compiler has to assign different identifiers for both accesses to *v3* vector, since they define two different memory references.

4.5 Setting the Cache Line Size

Depending on the number of memory references detected in the code, the cache line size has to be adapted to avoid unnecessary capacity conflicts within the execution of a loop iteration. If the number of references exceeds the number of cache lines, then conflicts are quite probable to appear. Therefore, the compiler has to select a cache line size that ensures that the number of available cache lines is greater or equal that the number of memory references.

5 Evaluation

The software cache implementation has been tested with two benchmarks from the HPCC benchmark suite [3]: Stream and Random Access. The Stream benchmark measures bandwidth ratios. It is composed by four synthetic kernels that measure sustainable memory bandwidth (in GB/s) and the corresponding computation rate for simple vector codes. The Random Access benchmark is composed by one kernel that operates on a single vector data type. The benchmark computes Global Updates per Second (GUPS). GUPS are calculated by identifying the number of memory locations that can be randomly updated in one second, divided by 1 billion (1e9). The term "randomly" means that there is little relationship between one address to be updated and the next, except that they occur in the space of 1/2 the total system memory.

All the measures were taken in a Cell BE-based blade machine with two Cell Broadband Engine processors at 3.2 GHz (SMT enabled), with 1 GB XDR RAM (512 MB each processor), running Linux Fedora Core 6 (Linux Kernel 2.6.20-CBE).

The software cache implementation was configured with the following cache parameters: 64Kb of capacity, 1024 sets and a varying cache line size ranging from 128 bytes up to 4096 bytes.

5.1 Stream Benchmark

Figure 5 shows the comparison between three different implementations, differing in the way the communications are managed. The Synchronous version forces a DMA synchronization after every DMA operation is programmed. This version corresponds to an implementation that would not allow for any overlapping between computation and communication. The Synchronous Flush version, allows for having asynchronous data communication from main memory to the Local Storage, but implements the flush operations (transfers from Local Storage to main memory) under a synchronous scheme. This version is not using the reference counter for cache lines, as a hint for determining the moment where a cache line has to be flushed before any reuse of it is required. Finally, the Asynchronous version, implements the software cache described in this paper, trying to maximize the overlapping of computation and communication.

For every version, the performance of each kernel (Copy, Scale, Add and Triad) is shown varying the size of the cache line (128, 256, 512, 1024, 2048 and 4096 bytes, from left to right). The results correspond to the obtained performance while executing with 8 SPEs. For brevity, the results executing with 1, 2 and 4 SPEs have been omitted as they were showing a very similar behavior. Clearly, and as it could be expected, every version significantly improves as long as the cache line size is increased. The comparison of the three versions allows for measuring the capabilities of the software cache implementation to overlap computation and communication. The results for the Synchronous version are taken as a baseline to be improved by the two other versions. The performance for the 128 bytes executions show how the different kernels behave while being dominated by the DMA operations.

The Synchronous version reaches 1.26 Gb/sec in average for the 4 kernels, the Synchronous Flush version reaches 1.75 Gb/sec, and finally the Asynchronous version reaches 2.10 Gb/sec. This corresponds to a speed up about 1.66. Similar behavior is observed when the cache line is increased from 128 up to 2048, reaching the best performance with a 2048 cache line size: 9.17 Gb/sec for Synchronous, 10.46 for

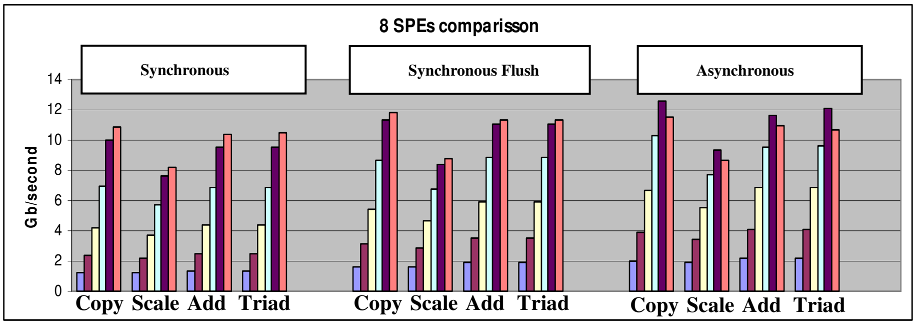


Fig. 5. Code transformation for non-stride accesses

Synchronous Flush and 11.38 for Asynchronous. This corresponds to a speed up about 1.24. Notice that when the cache line size is 4096, the increment of performance is not sustained. For the moment, it is not clear the reason of that behavior, so this needs more study.

5.2 Random Access Benchmark

The Random Access benchmark is used for evaluate the overlapping of computation and communication when the parallel code includes pointer-based memory accesses. Four different versions of the benchmark have been evaluated, depending on the unroll factor in the loop computation. Figure 6 shows the core of the computation. The unroll factor determines how many DMA transfers can be overlapped for the memory references to variable *Table*, according to the transformation described in section 4.5. For a 2 unroll factor, 2 _MMAP operations can be executed one immediate after the other. An unroll factor of 4 allows for overlapping 4 _MMAP operations, a factor of 8 allows for overlapping 8 _MMAP operations.

```
for (i=0; i<NUPDATE/128; i++) {
  for (j=0; j<128; j++) {
     ran[j] = (ran[j] << 1) ^ ((s64Int) ran[j] < 0 ? POLY : 0);
     Table[ran[j] & (TableSize-1)] ^= ran[j];
   }
}
```

Fig. 6. Source code for Random Access benchmark

Figure 7 shows the results executing with 1 and 8 SPEs. The cache line size has been set to 4096, but the access to the *Table* variable it is performed with he _PARTITIONED_ attribute, which makes every DMA transfer just involve 4096/32 = 128 bytes of data. This shows the ability of the software cache implementation to deal with both strided accesses and non strided accesses. The base line measurement corresponds to the version with no loop unrolling. The Y axis measures GUPS (Giga Updates per Second). The 1-SPE version significantly improves while the unrolling factor is increased. Improvements are about 30%, 56% and 73% with 2, 4 and 8 unroll factors respectively. Similar improvements are observed in the 8-SPE version.

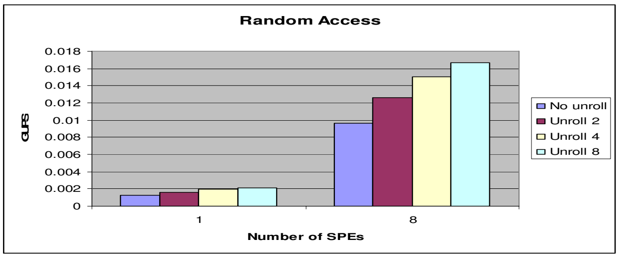


Fig. 7. Performance for Random Access

Although the difference in performance between the non-unrolled and the 8-urolled versions, we have detected a limiting factor due to the relation between the execution time for the _MMAP runtime service, and the DMA time for small data transfers (e.g.: 128 bytes). Small transfers perform very fast in the Cell-BE so they do not offer many chances for overlapping unless the execution time for the _MMAP service is such that can be fitted several times in one DMA transfer. The measurement for the Random Access show that our implementation is limited to the overlapping of 8 _MMAP operations for small DMA transfers. This suggests further study on how to optimize the _MMAP mechanism.

6 Conclusions

This paper describes the main features that have to be included in the implementation of a software cache implementation for the Cell BE processor, in order to maximize the chances for overlapping computation and communication.

It has been proved that a full-associative scheme offers better chances for overlapping computation and communication. It also has been pointed out the necessity of providing with mechanisms to detect the precise moment to initiate write back operations. This translates to overlapping the data transfer from the cache to main memory with actual computation, since the implementation guarantees that the necessary synchronization associated to the write back operation is going to be produced at next reuse of the flushed cache line. Besides, this is accompanied with a replacement policy that tends to increase the time between a use / reuse of the same cache line. Thus, delaying as much as possible the synchronization point and giving the hardware the necessary time to complete the data transfer.

The implementation has been evaluated with two benchmarks in the HPCC suite: Stream and Random Access. For both benchmarks, improvements are significant, ranging from 1.25 and 1.66 of speed up.

Acknowledgement

This work has been supported by the Ministry of Education of Spain under contract TIN2007-60625, and IBM in the context of the SOW Cell project. We would like also to acknowledge the Barcelona Supercomputing Center for letting us access to its Cell BE-based blades, provided by IBM through the SUR Program.

References

1. Eichenberger, A.E., O'Brien, K., O'Brien, K., Wu, P., Chen, T., Oden, P.H., Prener, D.A., Shepherd, J.C., So, B., Sura, Z.: Optimizing Compiler for a Cell Processor. In: 14th Parallel Architectures and Compilation Techniques, Saint Louis (Missouri) (September 2005)
2. Kistler, M., Perrone, M., Petrini, F.: Cell Multiprocessor Communication Network: Built for Speed. IEEE Micro 26(3), 10–23 (2006)

3. Luszczek, P., Bailey, D., Dongarra, J., Kepner, J., Lucas, R., Rabenseifner, R., Takahashi, D.: The HPC Challenge (HPCC) Benchmark Suite. In: SC 2006 Conference Tutorial. IEEE, Los Alamitos (2006)
4. Wang, Q., Zhang, W., Zang, B.: Optimizing Software Cache Performance of Packet Processing Applications. In: LCTES 2007 (2007)
5. Dai, J., Li, L., Huang, B.: Pipelined Execution of Critical Sections Using Software-Controlled Caching in Network Processors. In: Proceedings of the International Symposium on Code Generation and Optimization table of contents, pp. 312–324 (2007), ISBN:0-7695-2764-7
6. Ravindran, R., Chu, M., Mahlke, S.: Compiler Managed Partitioned Data Caches for Low Power. In: LCTES 2007 (2007)
7. Chen, T., Sura, Z., O'Brien, K., O'Brien, K.: Optimizing the use of static buffers for DMA on a Cell chip. In: 19th International Workshop on Languages and Compilers for Parallel Computing, New Orleans, Louisiana, November 2-4 (2006)

Pillar: A Parallel Implementation Language

Todd Anderson, Neal Glew, Peng Guo, Brian T. Lewis, Wei Liu, Zhanglin Liu, Leaf Petersen, Mohan Rajagopalan, James M. Stichnoth, Gansha Wu, and Dan Zhang

Microprocessor Technology Lab, Intel Corporation

Abstract. As parallelism in microprocessors becomes mainstream, new programming languages and environments are emerging to meet the challenges of parallel programming. To support research on these languages, we are developing a low-level language infrastructure called *Pillar* (derived from Parallel Implementation Language). Although Pillar programs are intended to be automatically generated from source programs in each parallel language, Pillar programs can also be written by expert programmers. The language is defined as a small set of extensions to C. As a result, Pillar is familiar to C programmers, but more importantly, it is practical to reuse an existing optimizing compiler like gcc [1] or Open64 [2] to implement a Pillar compiler.

Pillar's concurrency features include constructs for threading, synchronization, and explicit data-parallel operations. The threading constructs focus on creating new threads only when hardware resources are idle, and otherwise executing parallel work within existing threads, thus minimizing thread creation overhead. In addition to the usual synchronization constructs, Pillar includes transactional memory. Its sequential features include stack walking, second-class continuations, support for precise garbage collection, tail calls, and seamless integration of Pillar and legacy code. This paper describes the design and implementation of the Pillar software stack, including the language, compiler, runtime, and *high-level converters* (that translate high-level language programs into Pillar programs). It also reports on early experience with three high-level languages that target Pillar.

1 Introduction

Industry and academia are reacting to increasing levels of hardware concurrency in mainstream microprocessors with new languages that make parallel programming accessible to a wider range of programmers. Some of these languages are domain-specific while others are more general, but successful languages of either variety will share key features: language constructs that allow easy extraction of high levels of concurrency, a highly-scalable runtime that efficiently maps concurrency onto available hardware resources, a rich set of synchronization constructs like futures and transactions, and managed features from modern languages such as garbage collection and exceptions. In addition, these languages will demand good sequential performance from an optimizing compiler. Implementing such a language will require a sizable compiler and runtime, possibly millions of lines of code.

To reduce this burden and to encourage experimentation with parallel languages, we are developing a language infrastructure called Pillar (derived from Parallel Implementation Language). We believe that key parts of the compilers and runtimes for these

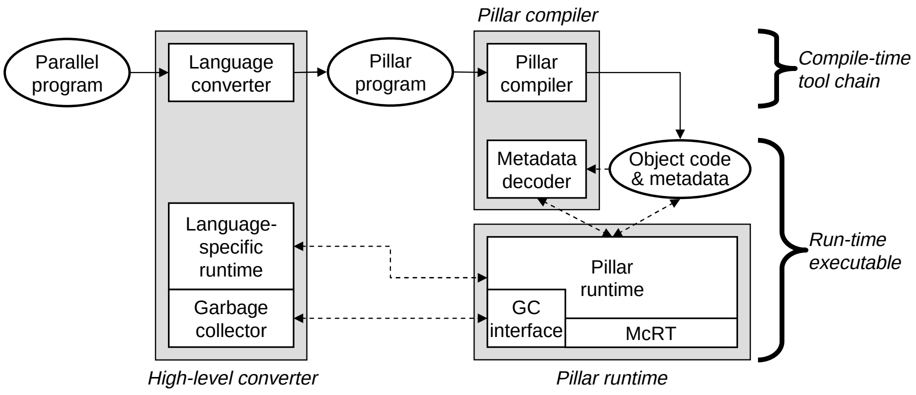


Fig. 1. The Pillar architecture and software stack

languages will have strong similarities. Pillar factors out these similarities and provides a single set of components to ease the implementation and optimization of a compiler and its runtime for any parallel language. The core idea of Pillar is to define a low-level language and runtime that can be used to express the sequential and concurrency features of higher-level parallel languages. The Pillar infrastructure consists of three main components: the Pillar language, a Pillar compiler, and the Pillar runtime.

To implement a parallel language using Pillar, a programmer first creates a *high-level converter* (see Fig. 1). This converter translates programs written in the parallel language into the Pillar language. Its main task is to convert constructs of the parallel language into Pillar constructs. The Pillar language is based on C and includes a set of modern sequential and parallel features (see Section 2). Since the Pillar compiler handles the tasks of code generation and traditional compiler optimizations, creating a high-level converter is significantly easier than creating a new parallel language compiler from scratch.

The second step is to create a runtime for the high-level language that provides the specialized support needed for that language's features. We call this runtime the language-specific runtime (LSR) to distinguish it from the Pillar runtime. The LSR could be written in Pillar and make use of Pillar constructs, or could be written in a language such as C traditionally used for runtime implementation. In either case, the Pillar code generated by the converter can easily call the LSR where necessary. The LSR can also make use of Pillar's runtime that, in addition to supporting the Pillar implementation, provides a set of services for high-level languages such as stack walking and garbage collection (GC) support. The Pillar runtime is layered on top of McRT, the Multi-Core RunTime [3], which provides scheduling, synchronization, and software transactional memory services.

Once the converter and LSR are written, complete executables can be formed by compiling the converted Pillar code with the Pillar compiler to produce object code, and then linking this with the LSR, the Pillar runtime, and McRT. The Pillar compiler produces both object code and associated metadata. This metadata is used by the Pillar runtime to provide services such as stack walking and root-set enumeration, and because of it, the code is said to be *managed*. (Pillar also supports integration with non-Pillar code, such as legacy code, which is said to be *unmanaged*.) The Pillar compiler controls the metadata format, and provides its own *metadata decoder* library to interpret it to the Pillar runtime. The metadata and decoder are also linked into the executable.

The design and implementation of Pillar is still in its early phases, and currently has a few key limitations: most notably, a cache-coherent shared-memory hardware model. Another consequence is that we are not yet in a position to do meaningful performance analysis, so this paper does not present any performance results. We intend to address these issues in the future, and we also hope to increase the range of high-level languages that can target Pillar.

The following sections focus on the Pillar language, compiler, and runtime.

2 The Pillar Language

The Pillar language has several key design principles. First, it is a compiler target language, with the goal of mapping any parallel language onto Pillar while maintaining that language's semantics. As such, Pillar cannot include features that vary across high-level languages, like object models and type-safety rules. C++, C#, and Java, for example, are too high-level to be effective target languages, as their object models and type-safety rules are not appropriate for many languages. Therefore, of necessity, Pillar is a fairly low-level language. Although most Pillar programs will be automatically generated, expert programmers must be able to directly create Pillar programs. As a result, assembly and bytecode languages are too low-level since they are difficult even for experts to use. Although inspired by `C--` [4,5], we decided to define Pillar as *a set of extensions to C* because then we could utilize existing optimizing C compilers to get quality implementations of Pillar quickly.

Since the Pillar language is based on C, type safety properties of the source parallel language must be enforced by the high-level converter. For example, array bounds checks might be implemented in Pillar using a combination of explicit tests and conditional branches. Similarly, null-dereference checks, divide-by-zero checks, enforcing data privacy, and restricting undesired data accesses must be done at a level above the Pillar language by the high-level converter. One notable exception is that we are working on annotations to express immutability and disambiguation of memory references.

Second, Pillar must provide support for key sequential features of modern programming languages. Examples include garbage collection (specifically, the ability to identify live roots on stack frames), stack walking (e.g., for exception propagation), proper tail calls (important when compiling functional languages), second-class continuations (e.g., for exception propagation and backtracking), and the ability to make calls between managed Pillar code and unmanaged legacy code.

Third, Pillar must also support key concurrency features of parallel languages, such as parallel thread creation, transactions, data-parallel operations, and futures. Fig. 2 summarizes the syntax of the Pillar features added to the C language. These features are described in the following sections.

Sequential constructs

Feature	Syntax example
Second-class continuations	`continuation k(a, b, c):` `cut to k(x, y, z);`
Alternate control flow	foo() `also cuts to` k1, k2; foo() `also unwinds to` k3, k4; foo() `never returns;`
Tail call	`tailcall` foo();
Spans	`span TAG value` { ··· }
Virtual stack and destructors	`VSE(k)` { ··· continuation k(target): ... cut to target; }
GC references	`ref` obj;
Managed/ unmanaged calls	`#pragma managed(off)` #include <stdio.h> `#pragma managed(on)` ... printf(· · ·);

Concurrency constructs

Feature	Syntax example
Pcall	`pcall(aff)` foo(a, b, c);
Prscall	`prscall(aff)` foo(a, b, c);
Futures	`fcall(aff, &st)` foo(a, b, c); `ftouch(&st);` `fwait(&st);`
Transactions	`TRANSACTION(k)` { ... continuation k(reason): if (reason==RETRY) ... else if (reason==ABORT) ... }

Fig. 2. Pillar syntactic elements

2.1 Sequential Features

Second-class continuations: This mechanism is used to jump back to a point in an older stack frame and discard intervening stack frames, similar to C's setjmp/longjmp mechanism. The point in the older stack frame is called a continuation, and is declared by the `continuation` keyword; the jumping operation is called a cut and allows multiple arguments to be passed to the target continuation. For any function call in Pillar, if the target function might ultimately cut to some continuation defined in the calling function rather than returning normally, then the function call must be annotated with all such continuations (these can be thought of as all alternate return points) so that the compiler can insert additional control flow edges to keep optimizations safe.

Virtual stack elements: A `VSE` (virtual stack element) declaration associates a cleanup task with a block of code. The "virtual stack" terminology is explained in Section 5.2. This cleanup task is executed whenever a cut attempts to jump out of the region of code associated with the VSE. This mechanism solves a problem with traditional stack cutting (such as in `C--`) where cuts do not compose well with many other operations. For example, suppose that code executing within a transaction cuts to some stack frame outside the transaction. The underlying transactional memory system would not get notified and this is sure to cause problems during subsequent execution. By using a VSE per transaction, the transactional memory system in Pillar is notified when a cut attempts to bypass it and can run code to abort or restart the transaction. Since cuts in Pillar compose well with all the features of Pillar, we call them *composable cuts*.

Stack walking: The Pillar language itself has no keywords for stack walking, but the Pillar runtime provides an interface for iterating over the stack frames of a particular thread. Pillar has the `also unwinds to` annotation on a function call for providing a list of continuations that can be accessed during a stack walk. This is useful for implementing exception propagation using stack walking, as is typical in C++, Java, and C# implementations.

Spans: Spans are a mechanism for associating specific metadata with call sites within a syntactic region of code, which can be looked up during stack walking.

Root-set enumeration: Pillar adds a primitive type called `ref` that is used for declaring local variables that should be reported as roots to the garbage collector. During stack walking these roots can be enumerated. The `ref` type may also contain additional parameters that describe how the garbage collector should treat the reference: e.g., as a direct object pointer versus an interior pointer, as a weak root, or as a tagged union that conditionally contains a root. These parameters have meaning only to the garbage collector, and are not interpreted by Pillar or its runtime. If `refs` escape to unmanaged code, they must be wrapped and enumerated specially, similar to what is done in Java for JNI object handles.

Tail calls: The `tailcall` keyword before a function call specifies a proper tail call: the current stack frame is destroyed and replaced with the callee's new frame.

Calls between managed and unmanaged code: All Pillar function declarations are implicitly tagged with the *pillar* attribute. The Pillar compiler also understands a special pragma that suppresses the *pillar* attribute on function declarations; this pragma is used when including standard C header files or defining non-Pillar functions.[1] Calling conventions and other interfacing depend on the presence or absence of the *pillar* attribute in both the caller and callee, and the Pillar compiler generates calls accordingly.

Note that spans, second-class continuations, and stack walking are `C--` constructs and are described in more detail in the `C--` specification [6].

2.2 Concurrency Features

Pillar currently provides three mechanisms for creating new logical threads: `pcall`, `prscall`, and `fcall`. Adding the `pcall` keyword in front of a call to a function with a void return type creates a new child thread, whose entry point is the target function. Execution in the original parent thread continues immediately with the statement following the `pcall`. Any synchronization or transfer of results between the two threads should use global variables or parameters passed to the `pcall` target function.

The `prscall` keyword is semantically identical to `pcall`, but implements a *parallel-ready sequential call* [7]. `Prscalls` allow programs to specify potential parallelism without incurring the overhead of spawning parallel threads if all processors are already busy. A `prscall` initially starts running the child thread as a sequential call (the parent is suspended). However, if a processor becomes free, it can start executing the parent in parallel with the child. Thus, `prscalls` are nearly as cheap as normal procedure calls, but take advantage of free processors when they become available.

[1] One particularly pleasing outcome of this syntax is that managed Pillar code and unmanaged C code can coexist within the same source files.

The `fcall` construct can be used to parallelize programs that have certain serializable semantics. The `fcall` annotation indicates that the call may be executed concurrently with its continuation, while allowing the call to be eagerly or lazily serialized if the compiler or runtime deems it unprofitable to parallelize it. The `st` parameter to the `fcall` is a synchronization variable, called a future, that indicates the status of the call: *empty* indicates that the call has not yet been started, *busy* indicates that the call is currently being computed, and *full* indicates that the call has completed. Two forcing operations are provided for futures: `ftouch` and `fwait`. If the future is full, both return immediately; if the future is empty, both cause the call to be run sequentially in the forcing thread; if the future is busy, `fwait` blocks until the call completes while `ftouch` returns immediately. The serializability requirement holds if, for each future, its first `ftouch` or `fwait` can be safely replaced by a call to the future's target function.

Both `prscall` and `fcall` are geared toward an execution environment where programs have a great deal of available fine-grain concurrency, with the expectation that the vast majority of calls can be executed sequentially within their parents' context instead of creating and destroying a separate thread.

These three keywords take an additional *affinity* parameter [8] that helps the scheduler place related threads close to each other to, e.g., improve memory locality.

Pillar provides support for transactions. A syntactic block of code is marked as a transaction, and transaction blocks may be nested. Within the transaction block, transactional memory accesses are specially annotated, and a continuation is specified as the "handler" for those situations where the underlying transactional memory system needs the program to respond to situations like a data conflict or a user retry.

The concurrency constructs described so far relate to lightweight thread-level parallelism. To support data parallelism, we intend to add Ct primitives [9] to Pillar. These primitives express a variety of nested data-parallel operations, and their semantics allow the compiler to combine and optimize multiple such operations.

3 Compiler and Runtime Architecture

The design of the Pillar language and runtime has several consequences for the Pillar compiler's code generation. In this section, we discuss some of the key interactions between the compiler-generated code and the runtime before getting into more detailed discussion of the compiler and the runtime in the following sections.

We assume that threads are scheduled cooperatively: that they periodically yield control to each other by executing yield check operations. Our experience shows that cooperative preemption offers several performance advantages over traditional preemptive scheduling in multi-core platforms [3]. The Pillar compiler is expected to generate a yield check at each method entry and backward branch, a well-known technique that ensures yielding within a bounded time. In addition to timeslice management, cooperative preemption is used on a running thread to get race-free access to a target thread's stack, for operations like root-set enumeration and prscall continuation stealing.

As we explain in Section 5, our `prscall` design allows threads to run out of stack space. This requires compiled code to perform an explicit limit check in the method

Compiler phase	Pillar changes	Percentage of compiler code	GC-related Pillar changes
Front-end	5 Kloc	1.2%	5.8%
Middle-end	6 Kloc	0.5%	1.2%
Back-end	11 Kloc	4.2%	20.7%
Total	22 Kloc	1.3%	12.0%

Fig. 3. Compiler modification statistics

prolog, jumping to a special stack extension routine if there is insufficient space for this method's stack frame. This strategy for inserting such copious limit and yield checks is likely to have a noticeable performance impact; future research will focus on mitigating these costs.

Stack walking operations like span lookup, root-set enumeration, and single-frame unwinding require the compiler to generate additional metadata for each call site. One approach would be to dictate a particular metadata format that a Pillar compiler must adhere to. Another approach, which we adopted, is to let the Pillar compiler decide on its own metadata format, and to specify a runtime interface for metadata-based operations. This means that the Pillar compiler also needs to provide its own library of metadata-decoding operations to be linked into Pillar applications, and the Pillar runtime calls those routines as necessary. We favor this latter approach because it gives the compiler more flexibility in generating optimized code.

4 The Pillar Compiler

We implemented our prototype Pillar compiler by modifying Intel's product C compiler. The compiler consists of a front-end, middle-end, and back-end, all of which we modified to support Pillar. Fig. 3 shows the number of lines of source code (LOC) changed or added, as well as the percentage of those source lines compared to those of the original code base. The front-end modifications are relatively small, limited to recognizing new Pillar lexical and syntax elements and translating them into the high-level intermediate representation (IR). In the middle- and back-end, our changes included adding new attributes and nodes to the existing IR and propagating them through compilation phases, as well as generating additional metadata required at run time. In addition, we added new internal data structures to accommodate Pillar constructs (e.g., continuations) and the necessary new analyses and phases (e.g., GC-related analysis).

Some Pillar constructs are implemented as simple mappings to Pillar runtime routines. These include some explicit Pillar language constructs, such as `cut to`, `pcall`, `prscall`, and `fcall`, as well as implicit operations such as stack limit checks, stack extension, yield checks, and managed and unmanaged transitions. The compiler may partially or fully inline calls to the runtime routines to improve performance. Fig. 4 gives an example showing how the compiler deals with some of these constructs:

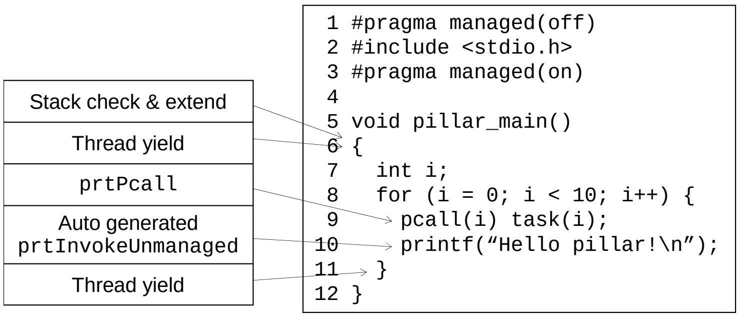
1. The compiler calculates the function's frame size and generates the stack limit check and extension in the prolog (line 5).

2. For cooperative scheduling, the compiler needs to generate the yield check in the prolog and at loop back-edges (line 5 & 11).
3. For Pillar concurrency constructs (`pcall`, `prscall`, and `fcall`), the compiler maps them to corresponding Pillar runtime interface functions (line 9).
4. When calling unmanaged C functions, the compiler automatically generates the transition call to the runtime routine `prtInvokeUnmanaged` (line 10).

Other Pillar constructs required deeper compiler changes. Continuation data structures must be allocated on the stack and initialized at the appropriate time. The continuation code itself must have a *continuation prolog* that fixes up the stack after a cut operation and copies continuation arguments to the right places. Continuations also affect intra-method control flow and register allocation decisions. The `VSE` and `TRANSACTION` constructs require control-flow edges into and out of the region being split so that a VSE is always pushed upon entry and popped upon exit (the push and pop operations are implemented as inlinable calls to the Pillar runtime). For every call site, the compiler must generate metadata to support runtime stack walking operations, such as unwinding a single frame, producing span data, and producing the set of live GC references. Tracking live GC references constitutes the most invasive set of changes to a C compiler, as the new `ref` type must be maintained through all IR generation and optimization phases. GC-related changes account for about 20% of the Pillar modifications in the back-end, and a very small fraction of the front-end and middle-end changes.

Some Pillar constructs need special treatment when implementing function inlining. First, the compiler cannot inline functions containing `tailcall`. Second, if the compiler decides to inline a call, and that call site contains explicit or implicit `also cuts to` and `also unwinds to` annotations, then all call sites within the inlined method inherit these annotations. (Implicit `also cuts to` annotations arise from calls inside a `VSE` or `TRANSACTION` construct—there is an implicit cut edge to the destructor continuation.) Third, the compiler needs to maintain extra metadata to support intra-frame unwinding, to ensure that the stack trace looks identical regardless of inlining decisions.

Even though some deep compiler changes were required, we are pleased that the changes only amounted to about 1–2% of the code base of a highly-optimizing



```
 1 #pragma managed(off)
 2 #include <stdio.h>
 3 #pragma managed(on)
 4
 5 void pillar_main()
 6 {
 7   int i;
 8   for (i = 0; i < 10; i++) {
 9     pcall(i) task(i);
10     printf(“Hello pillar!\n”);
11   }
12 }
```

Fig. 4. A Pillar example

production C compiler, and that they preserved the compiler's traditional optimizations.[2] Of those changes, about 12% overall were related to GC, which is the single most invasive Pillar feature to implement. We believe that Pillar support could be added to other optimizing compilers at a similarly low cost.

One limitation of basing the Pillar compiler on an existing large C compiler is that we are constrained to using Pillar constructs that can be fitted onto C. It would be hard, for example, for us to support struct layout control or multiple return values. The more non-C features we choose to support, the more work we would incur in modifying the compiler to support them. We believe we have chosen a reasonable point in the language design space for Pillar.

5 The Pillar Runtime

The Pillar runtime (PRT) provides services such as stack walking, root-set enumeration, parallel calls, stack management, and virtual stack support to compiled Pillar code and to an LSR. It is built on top of McRT [3], which the PRT relies on primarily for its scheduling and synchronization services, as well as its software transactional memory implementation [10]. The PRT interface is nearly independent of the underlying hardware: its architecture-specific properties include registers in the stack frame information returned by the stack walking support, and the machine word size. The remainder of this section provides some details on how the PRT supports its services.

5.1 Stack Walking and Root-Set Enumeration

The PRT provides support for walking the stack of a thread and enumerating the GC roots in its frames. To do this, PRT functions are called to (cooperatively) suspend the target thread, read the state of its topmost managed frame, then repeatedly step to the next older frame until no frames remain. At each frame, other functions can access that frame's instruction pointer, callee-saved registers, and GC roots. An additional function enumerates any roots that may be present in the thread's VSEs.

Stack walking is complicated by the need to unwind the stack in the presence of interleaved managed and unmanaged frames. The PRT does not presume to understand the layout of unmanaged stack frames, which may vary from compiler to compiler. Instead, it uses the `VSE` mechanism to mark contiguous regions of the stack corresponding to unmanaged code, and skips over the entire region during unwinding.

5.2 Composable Cuts

The PRT provides the implementation of composable cuts. These operate much like simple cuts but execute any destructor or cleanup operations of intervening VSEs.

Each thread contains a *virtual stack* of VSEs, in which the thread explicitly maintains a pointer to the virtual stack top, and each VSE contains a link to the next VSE on the stack. The continuation data structure also contains a slot for the virtual stack top. The

[2] Note, however, that a couple of optimization phases have not yet been made Pillar-safe, and are currently disabled.

PRT provides interfaces to push and pop VSEs. When a continuation is created, the current virtual stack top is stored in the continuation. Later, if a cut is made to this continuation, the PRT compares the current virtual stack top against the value saved in the target continuation. If these are the same, the PRT cuts directly to the target continuation. If they differ, one or more intervening frames require cleanup, and the PRT instead cuts directly to the destructor of the topmost VSE on the virtual stack, passing the original target continuation as an argument. When each VSE destructor is executed, it does its cleanup, removes itself from the virtual stack, then does another cut to the original target continuation passed to the destructor. This sequence continues until the target continuation is reached.

5.3 Prscalls

The Pillar compiler translates a `prscall` into a call to the PRT's prscall interface function. This function pushes a prscall VSE onto the virtual stack, copies arguments, and calls the prscall's child. Thus the child immediately starts executing sequentially. Later, an idle thread looking for work may steal the remainder of the parent's execution (unfortunately also called the parent's "continuation") by setting a continuation-stolen flag and restarting the parent. When the child terminates, it checks the continuation-stolen flag to determine whether to return to the parent or to simply exit because the continuation was stolen.

Our `prscall` design has interesting implications for stack management. When a `prscall` continuation is stolen, the stack becomes split between the parent and child threads, with the parent and child each owning one contiguous half. Since a stack can contain an arbitrary number of active `prscalls`, each of which can be stolen, a stack can become subdivided arbitrarily finely, leaving threads with tiny stacks that will quickly overflow. To deal with this, the Pillar runtime allows a thread to allocate a new "extension" stack (or "stacklet") to hold newer stack frames.

The PRT provides a stack extension wrapper that allocates a new stack (with an initial reference count of one), calls the target function, and deallocates the stack when the function returns. The stack extension wrapper also pushes a VSE whose destructor ensures that the stack will be properly deallocated in the event of a cut operation. To support stack sharing, each full stack contains a reference count word indicating how many threads are using a portion of the stack.

Logically, each `prscall` results in a new thread, regardless of whether the child runs sequentially or in parallel with its parent. When managing locks, those threads should have distinct and persistent thread identifiers, to prevent problems with the same logical thread acquiring a lock in the parent under one thread ID and releasing it under a different ID (the same is true for `pcall` and `fcall`). Each thread's persistent logical ID is stored in thread-local storage, and locking data structures must be modified to use the logical thread ID instead of a low-level thread ID. This logical thread ID is constructed simply as the address of a part of the thread's most recent pcall or prscall VSE. As such, the IDs are unique across all logical threads, and persistent over the lifetimes of the logical threads.

5.4 Fcalls

The Pillar compiler translates an `fcall` into a call to the PRT's future creation function. This function creates a future consisting of a status field (empty/busy/full) and a "thunk" that contains the future's arguments and function pointer, and adds the future to the current processor's future queue. Subsequently, if a call to `ftouch` or `fwait` is made and the future's status is empty, it is immediately executed in the current thread.

At startup, the PRT creates one future pool thread for each logical processor and pins each thread to the corresponding processor. Moreover, the PRT creates a future queue for each future pool thread. A future pool thread tries to run futures from its own queue, but if the queue is empty, it will try to steal futures from other queues to balance system load.

Once the future has been evaluated, the future's thunk portion is no longer needed. To reclaim these, the PRT represents futures using a thunk structure and a separate status structure. These point to each other until the thunk is evaluated, after which the thunk memory is released. The memory for the status structure is under the control of the Pillar program, which may allocate the status structure in places such as the stack, the malloc heap, or the GC heap. We use this two-part structure so that the key part of the future structure may be automatically managed by the GC while minimizing the PRT's knowledge of the existence or implementation of the GC.

6 Experience Using Pillar

This section describes our experience using Pillar to implement three programming languages having a range of different characteristics. These languages are Java, IBM's X10, and an implicitly-parallel functional language.

6.1 Compiling Java to Pillar

As part of our initial efforts, we attempted to validate the overall Pillar design through a simple Java-to-Pillar converter (JPC), leveraging our existing Java execution environment, the Open Runtime Platform (ORP) [11]. Given a trace of the Java classes and methods encountered during the execution of a program, the JPC generates Pillar code for each method from its bytecodes in the method's Java class file.

The resulting code exercises many Pillar features. First, Java variables of reference types are declared using the `ref` primitive type. Second, spans are used to map Pillar functions to Java method identifiers, primarily for the purpose of generating stack traces. They are also used, in conjunction with the `also unwinds to` annotation, to represent exception regions and handlers. Third, when an exception is thrown, ORP uses Pillar runtime functions to walk the stack and find a suitable handler, in the form of an `also unwinds to` continuation. When the continuation is found, ORP simply invokes a `cut to` operation. Fourth, VSEs are used for synchronized methods. Java semantics require that when an exception is thrown past a synchronized method, the lock is released before the exception handler begins. A synchronized method is wrapped inside a VSE whose cleanup releases the lock. Fifth, Java threads are started via the `pcall` construct. Sixth, Pillar's managed/unmanaged transitions are used for implementing JNI calls and other calls into the ORP virtual machine.

Although several Pillar features were not exercised by the JPC, it was still effective in designing and debugging the Pillar software stack, particularly the Pillar compiler that was subjected to hundreds of thousands of lines of JPC-generated code.

6.2 Compiling X10 to Pillar

X10 is a new object-oriented parallel language designed for high-performance computing being developed by IBM as part of the DARPA HPCS program [12]. It is similar to Java but with changes and extensions for high-performance parallel programming. It includes asynchronous threads, multidimensional arrays, transactional memory, futures, a notion of locality (places), and distribution of large data sets.

We selected X10 because it contains a number of parallel constructs not in our other efforts, such as places, data distributions, and clocks. We also want to experiment with thread affinity, data placement, optimizing for locality, and scheduling. X10, unlike our other languages, is a good language in which to do this experimentation.

We currently compile X10 by combining IBM's open-source reference implementation of X10 [13] with the Java-to-Pillar converter. We are able to compile and execute a number of small X10 programs, and this has substantially exercised Pillar beyond that of the Java programs. In the future we will experiment with affinity and data placement.

6.3 Compiling a Concurrent Functional Language

Pillar is also being used as the target of a compiler for a new experimental functional language. Functional languages perform computation in a largely side-effect-free fashion, which means that a great deal of the computational work in a program can be executed concurrently with minimal or no programmer intervention [14,15].

Previous work has compiled functional languages to languages such as C [16], Java byte codes [17,18], and the Microsoft Common Language Runtime (CLR) [19]. These attempts reported benefits such as interoperability, portability, and ease of compiler development. However, they have also noted the mismatches between the functional languages and the different target languages. The inability to control the object model in Java and CLR, the lack of proper tail calls, the restrictions of type safety in Java and CLR, and the inability to do precise garbage collection naturally in C, all substantially complicate compiler development and hurt performance of the final code.

`C--` and Pillar are designed to avoid these problems and provide an easy-to-target platform for functional languages. Like the Java-to-Pillar converter, our experience with the functional language showed Pillar to be an excellent target language. Pillar's lack of a fixed object model, its support for proper tail calls, and its root-set enumeration all made implementing our functional language straightforward. Also, since Pillar is a set of C extensions, we implement most of our lowest IR directly as C preprocessor macros, and generating Pillar from this IR is straightforward. We can include C header files for standard libraries and components (e.g., the garbage collector) coded in C, and Pillar automatically interfaces the Pillar and C code. Pillar's second-class continuations are used to provide a specialized control-flow construct of the language. The stack walking-based exceptions of Java and CLR would be too heavyweight for this purpose, and

C's setjmp/longjmp mechanism is difficult to use correctly and hinders performance. Implementing accurate GC in the converter is as easy as in the Java-to-Pillar converter—simply a matter of marking roots as `ref`s and using the Pillar stack walking and root-set enumeration support.

7 Related Work

The closest language effort to Pillar is `C--` [4,5,6]. `C--` is intended as a low-level target language for compilers—it has often been described as a "portable assembler". Almost all Pillar features can be expressed in `C--`, but we designed Pillar to be slightly higher level than `C--`. Pillar includes, for example, `ref`s and threads instead of (as `C--` would) just the mechanisms to implement them. We also designed Pillar as extensions to C, rather than directly using `C--`, to leverage existing C compilers.

LLVM [20] is another strong and ongoing research effort whose goal is to provide a flexible compiler infrastructure with a common compiler target language. LLVM's design is focused on supporting different compiler optimizations, while Pillar is aimed at simplifying new language implementations, in part by integrating readily into an existing highly-optimizing compiler. Comparing language features, the most important differences between Pillar and LLVM are that LLVM lacks second-class continuations, spans, pcalls, prscalls, and fcalls.

C# and CLI [21,22] are often used as intermediate languages for compiling high-level languages, and early on we considered them as the basis for Pillar. However, they lack second-class continuations, spans, prscalls, and fcalls. Furthermore, they are too restrictive in that they impose a specific object model and type-safety rules.

Pillar uses ideas from or similar to other projects seeking to exploit fine-grained parallelism without creating too many heavyweight threads. Pillar's prscalls are taken directly from Goldstein's parallel-ready sequential calls [7], which were designed to reduce the cost of creating threads yet make effective use of processors that become idle during execution. Also, like Cilk [23] and Lea's Java fork/join framework [24], Pillar uses work stealing to make the most use of available hardware resources and to balance system load. Furthermore, during prscall continuation stealing, Pillar tries to steal the earlier (deeper) continuations as Cilk does, since seizing large amounts of work tends to reduce later stealing costs. Pillar's future implementation differs from the lazy futures of Zhang et al. [25], which are implemented using a lazy-thread creation scheme similar to Pillar's prscalls. Since Pillar supports both prscalls and a separate future pool-based implementation, it will be interesting to compare the performance of both schemes for implementing futures.

8 Summary

We have described the design of the Pillar software infrastructure, consisting of the Pillar language, the Pillar compiler, and the Pillar runtime, as well as the high-level converter that translates programs from a high-level parallel language into Pillar. By defining the Pillar language as a small set of extensions to the C language, we were able to create an optimizing Pillar compiler by modifying only 1–2% of an existing optimizing C compiler. Pillar's thread-creation constructs, designed for a high-level converter that can find a great deal of concurrency opportunities, are optimized for sequential execution to minimize thread creation and destruction costs. Pillar's sequential constructs, many of which are taken from `C--`, have proven to be a good target for languages with modern features, such as Java and functional languages.

Our future work includes adding support for nested data parallel operations [9] to efficiently allow parallel operations over collections of data. In addition, although we currently assume a shared global address space, we plan to investigate Pillar support for distributed address spaces and message passing.

We are still in the early stages of using Pillar, but our experience to date is positive—it has simplified our implementation of high-level parallel languages, and we expect it to significantly aid experimentation with new parallel language features and implementation techniques.

Acknowledgements

We have engaged in many hours of discussions with Norman Ramsey and John Dias of Harvard University, and Simon Peyton Jones of Microsoft Research, on issues regarding Pillar and `C--`. These discussions have been instrumental in focusing the design of Pillar. We also thank the reviewers for their feedback on this paper.

References

1. GNU: The GNU Compiler Collection, `http://gcc.gnu.org/`
2. Open64: The Open Research Compiler, `http://www.open64.net/`
3. Saha, B., Adl-Tabatabai, A., Ghuloum, A., Rajagopalan, M., Hudson, R., Petersen, L., Menon, V., Murphy, B., Shpeisman, T., Sprangle, E., Rohillah, A., Carmean, D., Fang, J.: Enabling Scalability and Performance in a Large Scale CMP Environment. In: EuroSys (March 2007)
4. Peyton Jones, S., Nordin, T., Oliva, D.: C–: A portable assembly language. In: Implementing Functional Languages 1997 (1997)
5. Peyton Jones, S., Ramsey, N.: A single intermediate language that supports multiple implementations of exceptions. In: Proceedings of the SIGPLAN 2000 Conference on Programming Language Design and Implementation (June 2000)
6. Ramsey, N., Peyton Jones, S., Lindig, C.: The C– language specification, version 2.0 (February 2005), `http://cminusminus.org/papers.html`
7. Goldstein, S.C., Schauser, K.E., Culler, D.E.: Lazy threads: implementing a fast parallel call. Journal of Parallel and Distributed Computing 37(1), 5–20 (1996)
8. Rajagopalan, M., Lewis, B.T., Anderson, T.A.: Thread Scheduling for Multi-Core Platforms. In: HotOS 2007: Proceedings of the Eleventh Workshop on Hot Topics in Operating Systems (May 2007)
9. Ghuloum, A., Sprangle, E., Fang, J.: Flexible Parallel Programming for Tera-scale Architectures with Ct (2007), `http://www.intel.com/research/platform/terascale/TeraScale_whitepaper.pdf`

10. Saha, B., Adl-Tabatabai, A.R., Hudson, R.L., Minh, C.C., Hertzberg, B.: McRT-STM: a high performance software transactional memory system for a multi-core runtime. In: PPoPP 2006: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 187–197. ACM Press, New York (2006)
11. Cierniak, M., Eng, M., Glew, N., Lewis, B., Stichnoth, J.: Open Runtime Platform: A Flexible High-Performance Managed Runtime Environment. Intel Technology Journal 7(1) (February 2003), http://www.intel.com/technology/itj/archive/2003.htm
12. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In: OOPSLA 2005: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications, pp. 519–538. ACM Press, New York (2005)
13. IBM: The Experimental Concurrent Programming Language X10. SourceForge (2007), http://x10.sourceforge.net/x10home.shtml
14. Harris, T., Marlow, S., Peyton Jones, S.: Haskell on a shared-memory multiprocessor. In: Haskell 2005: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell, pp. 49–61. ACM Press, New York (2005)
15. Hicks, J., Chiou, D., Ang, B.S.: Arvind: Performance studies of Id on the Monsoon Dataflow System. Journal of Parallel and Distributed Computing 18(3), 273–300 (1993)
16. Tarditi, D., Lee, P., Acharya, A.: No assembly required: compiling standard ML to C. ACM Letters on Programming Languages and Systems 1(2), 161–177 (1992)
17. Benton, N., Kennedy, A., Russell, G.: Compiling standard ML to Java bytecodes. In: ICFP 1998: Proceedings of the third ACM SIGPLAN international conference on Functional programming, pp. 129–140. ACM Press, New York (1998)
18. Serpette, B.P., Serrano, M.: Compiling Scheme to JVM bytecode: a performance study. In: ICFP 2002: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming, pp. 259–270. ACM Press, New York (2002)
19. Benton, N., Kennedy, A., Russo, C.V.: Adventures in interoperability: the sml.net experience. In: PPDP 2004: Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming, pp. 215–226. ACM Press, New York (2004)
20. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO 2004), Palo Alto, California (March 2004)
21. ECMA: Common Language Infrastructure. ECMA (2002), http://www.ecma-international.org/publications/Standards/ecma-335.htm
22. ISO: ISO/IEC 23270 ($C^{\#}$). ISO/IEC standard (2003)
23. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An Efficient Multithreaded Runtime System. Journal of Parallel and Distributed Computing 37(1), 55–69 (1996)
24. Lea, D.: A Java Fork/Join Framework. In: Proceedings of the ACM 2000 Java Grande Conference, pp. 36–43. ACM Press, New York (2000)
25. Zhang, L., Krintz, C., Soman, S.: Efficient Support of Fine-grained Futures in Java. In: PDCS 2006: IASTED International Conference on Parallel and Distributed Computing and Systems (November 2006)

Author Index