Lecture Notes in Computer Science 5335

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

José Nelson Amaral (Ed.)

Languages and Compilers for Parallel Computing

21st International Workshop, LCPC 2008
Edmonton, Canada, July 31 – August 2, 2008
Revised Selected Papers

Springer

Volume Editor

José Nelson Amaral
University of Alberta
Department of Computing Science
Edmonton, AB, T6G-2E8, Canada
E-mail: amaral@cs.ualberta.ca

Library of Congress Control Number: 2008940024

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 3-540-89739-9 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-89739-2 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12567744 06/3180 5 4 3 2 1 0

Preface

In 2008 the Workshop on Languages and Compilers for Parallel Computing left the USA to celebrate its 21st anninversary in Edmonton, Alberta, Canada. Following its long-established tradition, the workshop focused on topics at the frontier of research and development in languages, optimizing compilers, applications, and programming models for high-performance computing. While LCPC continues to focus on parallel computing, the 2008 edition included the presentation of papers on program analysis that are precursors of high performance in parallel environments.

LCPC 2008 received 35 paper submissions. Each paper received at least three independent reviews, and then the papers and the referee comments were discussed during a Program Committee meeting. The PC decided to accept 18 papers as regular papers and 6 papers as short papers. The short papers appear at the end of this volume.

The LCPC 2008 program was fortunate to include two keynote talks. Keshav Pingali's talk titled "Amorphous Data Parallelism in Irregular Programs" argued that irregular programs have data parallelism in the iterative processing of worklists. Pingali described the Galois system developed at The University of Texas at Austin to exploit this kind of amorphous data parallelism.

The second keynote talk, "Generic Parallel Algorithms in Threading Building Bocks (TBB)," presented by Arch Robison from Intel Corporation addressed very practical aspects of using TBB, a production C++ library, for generic parallel programming and contrasted TBB with the Standard Template Library (STL).

LCPC continues to be a strong workshop thanks to the support that it enjoys from both the programming language and optimizing compiler communities and from the various segments of the high-performance computing community. The continued strength of LCPC is also in no small part due to the passionate commitment of David Padua and the Steering Committee as well as due to the time commitment of Program Committee members, anonymous reviewers, and contributing authors. We are grateful for the financial support provided by iCore. The organization of LCPC by the Department of Computing Science at the University of Alberta in Edmonton counted on outstanding volunteers including Fran Moore, Sheryl Maiko, and Sunrose Ko.

September 2008 José Nelson Amaral

Organization

LCPC 2008 was organized by the Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada

Executive Committee

General and Program Chair

José Nelson Amaral	University of Alberta

Steering Committee

Rudolf Eigenmann	Purdue University
Alex Nicolau	University of California at Irvine
David Padua	University of Illinois at Urbana Champaign
Lawrence Reuchwerger	Texas A&M University

Program Committee

Vikram Adve	University of Illinois at Urbana Champaign
Gheorghe Almási	IBM T.J. Watson Research Center
José Nelson Amaral	University of Alberta
Eduard Ayguadé	Universitat Politécnica de Catalunya
Gerald Baumgartner	Louisiana State University
Călin Caşcaval	IBM T.J. Watson Research Center
John Cavazos	University of Delaware
María Garzarán	University of Illinois at Urbana-Champaign
Xiaoming Li	University of Delaware
Lori Pollock	University of Delaware
J. Ramanujam	Louisiana State University
P. Sadayappan	Ohio State University
Peng Wu	IBM T.J. Watson Research Center

Sponsoring Institutions

Department of Computing Science, University of Alberta, Edmonton, AB, Canada

Informatics Circle of Research Excellence (iCORE), Alberta, Canada

Table of Contents

CUDA-Lite: Reducing GPU Programming Complexity 1
Sain-Zee Ueng, Melvin Lathara, Sara S. Baghsorkhi, and Wen-mei W. Hwu

MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs 16
John A. Stratton, Sam S. Stone, and Wen-mei W. Hwu

Automatic Pre-Fetch and Modulo Scheduling Transformations for the Cell BE Architecture 31
Nikola Vujić, Marc Gonzàlez, Xavier Martorell, and Eduard Ayguadé

Efficient Set Sharing Using ZBDDs 47
Mario Méndez-Lojo, Ondřej Lhoták, and Manuel V. Hermenegildo

Register Bank Assignment for Spatially Partitioned Processors 64
Behnam Robatmili, Katherine Coons, Doug Burger, and Kathryn S. McKinley

Smashing: Folding Space to Tile through Time 80
Nissa Osheim, Michelle Mills Strout, Dave Rostron, and Sanjay Rajopadhye

Identification of Heap–Carried Data Dependence Via Explicit Store Heap Models 94
Mark Marron, Darko Stefanovic, Deepak Kapur, and Manuel Hermenegildo

On the Scalability of an Automatically Parallelized Irregular Application 109
Martin Burtscher, Milind Kulkarni, Dimitrios Prountzos, and Keshav Pingali

Statistically Analyzing Execution Variance for Soft Real-Time Applications 124
Tushar Kumar, Romain Cledat, Jaswanth Sreeram, and Santosh Pande

Minimum Lock Assignment: A Method for Exploiting Concurrency among Critical Sections 141
Yuan Zhang, Vugranam C. Sreedhar, Weirong Zhu, Vivek Sarkar, and Guang R. Gao

Set-Congruence Dynamic Analysis for Thread-Level Speculation (TLS) 156
Cosmin E. Oancea and Alan Mycroft

Thread Safety through Partitions and Effect Agreements 172
Nicholas D. Matsakis and Thomas R. Gross

P-Ray: A Software Suite for Multi-core Architecture Characterization 187
Alexandre X. Duchateau, Albert Sidelnik, María Jesús Garzarán, and David Padua

Scalable Implementation of Efficient Locality Approximation 200
Xipeng Shen and Jonathan Shaw

P-OPT: Program-Directed Optimal Cache Management 217
Xiaoming Gu, Tongxin Bai, Yaoqing Gao, Chengliang Zhang, Roch Archambault, and Chen Ding

Compiler-Driven Dependence Profiling to Guide Program Parallelization 232
Peng Wu, Arun Kejariwal, and Călin Caşcaval

gluepy: A Simple Distributed Python Programming Framework for Complex Grid Environments 249
Ken Hironaka, Hideo Saito, Kei Takahashi, and Kenjiro Taura

A Fully Parallel LISP2 Compactor with Preservation of the Sliding Properties 264
Xiao-Feng Li, Ligang Wang, and Chen Yang

A Case Study in Tightly Coupled Multi-paradigm Parallel Programming 279
Sayantan Chakravorty, Aaron Becker, Terry Wilmarth, and Laxmikant Kalé

ASYNC Loop Constructs for Relaxed Synchronization 292
Russell Meyers and Zhiyuan Li

Design for Interoperability in STAPL: `pMatrices` and Linear Algebra Algorithms 304
Antal A. Buss, Timmie G. Smith, Gabriel Tanase, Nathan L. Thomas, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger

Implementation of Sensitivity Analysis for Automatic Parallelization ... 316
Silvius Rus, Maikel Pennings, and Lawrence Rauchwerger

Just-In-Time Locality and Percolation for Optimizing Irregular Applications on a Manycore Architecture 331
Guangming Tan, Vugranam C. Sreedhar, and Guang R. Gao

Exploring the Optimization Space of Dense Linear Algebra Kernels 343
Qing Yi and Apan Qasem

Author Index 357

CUDA-Lite: Reducing GPU Programming Complexity

Sain-Zee Ueng, Melvin Lathara, Sara S. Baghsorkhi, and Wen-mei W. Hwu

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

Abstract. The computer industry has transitioned into multi-core and many-core parallel systems. The CUDA programming environment from NVIDIA is an attempt to make programming many-core GPUs more accessible to programmers. However, there are still many burdens placed upon the programmer to maximize performance when using CUDA. One such burden is dealing with the complex memory hierarchy. Efficient and correct usage of the various memories is essential, making a difference of 2-17x in performance. Currently, the task of determining the appropriate memory to use and the coding of data transfer between memories is still left to the programmer. We believe that this task can be better performed by automated tools. We present CUDA-lite, an enhancement to CUDA, as one such tool. We leverage programmer knowledge via annotations to perform transformations and show preliminary results that indicate auto-generated code can have performance comparable to hand coding.

1 Introduction

In 2007, NVIDIA introduced the Compute Unified Device Architecture (CUDA) [9], an extended ANSI C programming model. Under CUDA, Graphics Processing Units (GPUs) consist of many processor cores, each of which can directly address into a global memory. This allows for a much more flexible programming model than previous GPGPU programming models [11], and allows developers to implement a wider variety of data-parallel kernels. As a result, CUDA has rapidly gained acceptance in application domains where GPUs are used to execute compute intensive, data-parallel application kernels.

While GPUs have been designed with higher memory bandwidth than CPUs, the even higher compute throughput of GPUs can easily saturate their available memory bandwidth. For example, the NVIDIA GeForce 8800 GTX comes with 86.4 GB/s memory bandwidth, approximately ten times that of Intel CPUs on a Front Side Bus. However, since the GeForce 8800 has a peak performance of 384 GFLOPS and each floating point operation operates on up to 12 bytes of source data, the available memory bandwidth cannot sustain even a small fraction of the peak performance if all of the source data are accessed from global memory.

Consequently, CUDA and its underlying GPUs offer multiple memory types with different bandwidth, latency, and access restrictions to allow programmers

J.N. Amaral (Ed.): LCPC 2008, LNCS 5335, pp. 1–15, 2008.

to conserve memory bandwidth while increasing the overall performance of their applications. Currently, CUDA programmers are responsible for explicitly allocating space and managing data movement among the different memories to conserve memory bandwidth. Furthermore, additional hardware mechanisms at the memory interface can enhance the main memory access efficiency if the access patterns follow memory coalescing rules. Currently, CUDA programmers shoulder the responsibility of massaging the code to produce the desirable access patterns. Experiences show that such responsibility presents a major burden on the programmer. CUDA-lite is designed to relieve such burden. Furthermore, CUDA code that is explicitly optimized for one GPU's memory hierarchy design may not easily port to the next generation or other types of data-parallel execution vehicles.

This paper presents CUDA-lite, an experimental enhancement to CUDA that allows programmers to deal only with global memory, the main memory of a GPU, with transformations to leverage the complex memory hierarchy. For increased efficiency, the programmers provide annotations describing certain properties of the data structures and code regions designated for GPU execution. The CUDA-lite tools analyze the code along with these annotations and determine if the memory bandwidth can be conserved and latency can be reduced by utilizing any special memory types and/or by massaging memory access patterns. Upon detection of an opportunity, CUDA-lite performs the transformations and code insertions needed. CUDA-lite is designed as a source-to-source translator. The output is CUDA code with explicit memory-type declarations and data transfers for a particular GPU. We envision CUDA-lite to eventually target multiple types and generations of data-parallel execution vehicles. If maximum performance is desired, the programmer can still choose to program certain kernels at the CUDA level.

In this paper we present CUDA-lite in detail. We cover the memories and techniques that are leveraged by the tool to conserve memory bandwidth and reduce memory latency. We describe how CUDA-lite identifies the opportunities and the hand transformations that it replaces. We have developed plug-ins for the Phoenix compiler [7] from Microsoft to perform all of the transformations as a source-to-source compiler, and evaluated our results by passing the resulting source code through NVIDIA's tool chain. We show that the performance of code generated by CUDA-lite matches or is comparable to hand generated code.

2 CUDA Programming Model

The CUDA programming model is ANSI C extended with keywords and constructs. The GPU is treated as a coprocessor that executes data-parallel kernel functions. The user supplies a single source program encompassing both host (CPU) and kernel (GPU) code. These are separated and compiled by NVIDIA's compiler, *nvcc*. The host starts the kernel code with a function call. The complete description of the programming model can be found in [8,9,10].

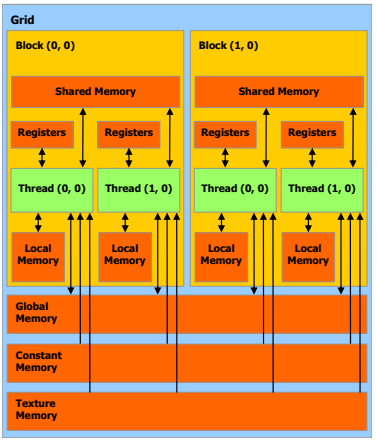


Fig. 1. CUDA Programming Model and Memory Hierarchy

Figure 1 depicts the programming model and memory hierarchy of CUDA. Threads are organized into a three-level hierarchy, and are executed on the *streaming multiprocessors* (SMs) on the GPU. At the highest level, each kernel creates a single *grid*, which consists of many *thread blocks* (TBs) arranged in two dimensions. The maximum number of threads per TB is 512, arranged in a three dimensional manner. Each TB is assigned to a single SM for its execution. Each SM can handle up to eight TBs at a time. Threads in the same TB can share data through the on-chip shared memory and can perform barrier synchronization by invoking the `__syncthreads` primitive. Synchronization across TBs can only be safely accomplished by terminating the kernel.

One of the major bottleneck to achieving performance while using CUDA is the memory bandwidth and latency. The GPU provides several different memories with different behaviors and performance that can be leveraged to improve memory performance. However, the programmer must explicitly and correctly utilize these different memories in the source code in order to gain the benefit. In the rest of this section we will examine *shared memory* and desirable memory access patterns to *global memory* that improve memory performance, and show the work required of programmers. Work that CUDA-lite intends to automate.

We focus on memory coalescing for global memory and shared memory in this work since these are the only writable memories in CUDA. We leave the read-only memories, constant and texture, for future work.

2.1 Global Memory

CUDA exposes a general-purpose, random access, readable and writable off-chip global memory visible to all threads. It is the slowest of the available memory spaces, requiring hundreds of cycles, and is not cached. However, its resemblance to a CPU's memory in its generality and size are also what allows more

```
1  #define ASIZE 3000
   #define TPB 256

   __global__ void
5  kernel (float *a, float *b)
   {
     int thi = threadIdx.x;
     int bki = blockIdx.x;
     float t = (float) thi + bki;
10   int i;

     if (bki * TPB + thi >= ASIZE)
       return;

15   for (i = 0; i < ASIZE; i++)
       {
         b[(bki*TPB+thi)*ASIZE + i] =
           a[(bki*TPB+thi)*ASIZE + i] * t;
       }
20 }
```

```
int main ()
{
  int num_blocks;
  int size = sizeof (float) * ASIZE * ASIZE;

  /* Allocate a_host and b_host,
   * and initialize a_host with values */

  /* Allocate a_device and b_device */
  cudaMalloc ((void **) &a_device, size);
  cudaMalloc ((void **) &b_device, size);

  /* Copy values from host to device */
  cudaMemcpy (a_device, a_host, size,
              cudaMemcpyHostToDevice);

  num_blocks = ASIZE % TPB == 0 ?
    ASIZE / TPB : (ASIZE / TPB) + 1;

  /* Number of thread blocks in the grid */
  dim3 gridDim (num_blocks);
  /* Number of threads per thread block */
  dim3 blockDim (TPB);

  /* Start executing on the GPU */
  kernel <<<gridDim, blockDim>>>
    (a_device, b_device);

  /* Copy values from device back to host */
  cudaMemcpy (b_host, b_device, size,
              cudaMemcpyDeviceToHost);
}
```

Fig. 2. Example Code: Base Case

general-purpose applications to be ported easily onto the GPU. A straightforward implementation of an application would be to utilize only global memory as a proof of concept for parallelizing the algorithm on CUDA.

Figure 2 shows an example CUDA code. The function `main` sets up the data for computation on the CPU while the function `kernel` contains the code that is actually executed on the GPU. Notice that variables that reside in the global memory of the GPU, like `a_device`, are allocated in `main` and data movement is also performed there via API calls to `cudaMemcpy`.

In the `kernel` function, each thread on the GPU traverses a different row of the 2-D array `a`, scaling each element by a thread specific value before storing into the corresponding location in array `b`. Since each TB must have the same number of threads, depending on the data size and program parallelization there may be excess threads that do not have data to operate on. The conditional check on line 12 that exits the kernel function before the loop handles these cases. This check becomes important as we attempt to utilize memory coalescing (Section 2.3).

2.2 Shared Memory

Shared memory is a small (16KB per SM for the GeForce 8800) readable and writable on-chip memory and as fast as register access. Shared memory is uninitialized at the beginning of execution, and resident data is private to each TB

and visible to all threads within the same TB. The intuition is that shared memory should be used for data that is reused, especially if reused across different threads in a TB. However, we found that memory performance improvement from coalesced global memory accesses (Section 2.3) is large enough that shared memory should be leveraged for such purposes even if there is no data reuse.

2.3 Memory Coalescing

Global memory does have a behavior called *memory coalescing* that helps conserve bandwidth while reducing effective latency. Conceptually it is similar to loading an entire cache line from memory versus loading one word at a time. Threads in a TB are numbered along the x direction first and gathered sequentially into *warps*. On the GeForce 8800 a group of 32 threads form a warp. Each warp executes in SIMD (single-instruction, multiple-data) fashion, i.e. all threads in the same warp execute the same instruction at the same time. When the threads of a half-warp execute a global load, the loads are consolidated if they meet constraints necessary for the hardware to perform memory coalescing. Otherwise the loads are serviced individually. We summarize the requirements here and refer interested readers to [10] for full details.

There are four major requirements that memory accesses to global memory have to follow for memory coalescing to happen:

1. Each element of the array has to be 4, 8, or 16 bytes and aligned.
2. The threads in the half-warp have to access consecutive memory addresses in order, e.g. thread number `N` within the half-warp need to access address `BaseAddr + N`.
3. Thread numbering matters only along the first dimension of the thread block, the x dimension.[1]
4. `BaseAddr` must be aligned to a multiple of the element size.

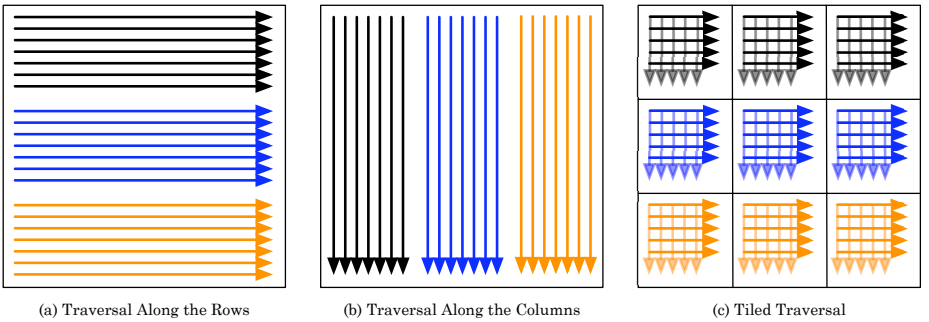


Fig. 3. Graphical View of Data Traversal: (a) Row (b) Column (c) Tiled

[1] Thread blocks are usually created so that the x dimension is a multiple of the number of threads in a warp.

The requirements for memory coalescing are complex. Furthermore, with the exception of access alignment, all of the requirements must be fulfilled or there will be no improvement in the memory performance; a partial improvement usually occurs if alignment is the only requirement missed. The data access pattern to fulfill the memory coalescing requirement is also not natural for all algorithms, e.g. reduction across the rows of an array. When performing a reduction across the rows of an array, it is more natural to have one thread per row, as in Figure 3(a). The different groups of colored arrows represent different TBs. However, traversing one thread per column, shown in Figure 3(b), is needed to fulfill requirement 2 for memory coalescing. The data accessed by threads in a half-warp need to be adjacent to one another in the horizontal direction, not vertical, for the accesses to coalesce. The lack of synchronization across TBs also contributes to making this traversal pattern unnatural for performing a reduction across rows in CUDA. An alternative is to tile the computation, as shown in Figure 3(c). The tile is first traversed along the column and data is coalesced loaded into a buffer in shared memory, indicated by the grayed arrows. The algorithm then operates on the data along the row from shared memory before moving to the next tile. The performance improvement from doing coalesced loads and using shared memory makes this worthwhile despite the instruction overhead.

For example, the memory access to array `a` on line 18 of Figure 2 does not coalesce because it violates rule number 2. For each iteration of the loop, thread `N` accesses `a[N*ASIZE + i]`; `bki` does not matter since the threads are in the same thread block. This means that each thread is accessing data vertically adjacent to each other, as in Figure 3(a), which does not trigger coalescing.

Figure 4 shows the kernel code from Figure 2 rewritten by hand so the algorithm is tiled and the memory accesses coalesced. The amount of code is roughly doubled. The original loop has been tiled and additional code is inserted to load/store data between global and shared memory. The load from array `a` on line 25 is coalesced since thread `N` accesses `a[k*ASIZE + N]` on each iteration. The computation kernel now operates on the data in shared memory, and the loop around it has included the check on line 12 of the original code as an additional condition. In other words, the excess threads we mentioned back in Section 2.1 may be used to perform memory coalescing accesses, but must not be allowed to perform actual computation.

This rewriting is a large additional burden on the programmer. Not only must the programmer fulfill the memory coalescing requirements, the programmer also has to maintain correctness. The performance improvement this optimization provides will be the ideal, or oracle, case for CUDA-lite.

3 CUDA-Lite

Since the behavior of memory coalescing is complex yet understood, we believe that such transformations are best undertaken by an automated tool. This would reduce the potential for errors in writing memory coalescing code, and reduce the burden upon programmers. In our vision, programmers would provide a

straightforward implementation of the kernel code that utilizes only global memory, and depend on tools to optimize the memory performance.

We have developed tools to automate the transformations previously done by hand to maximize memory performance via memory coalescing. The programmer provides a version of the program that has been parallelized for CUDA using only global memory and the tools output a version with the memory accesses optimized. In other words, the tools transform code like the kernel function in Figure 2 to the memory coalescing version in Figure 4. We rely upon information

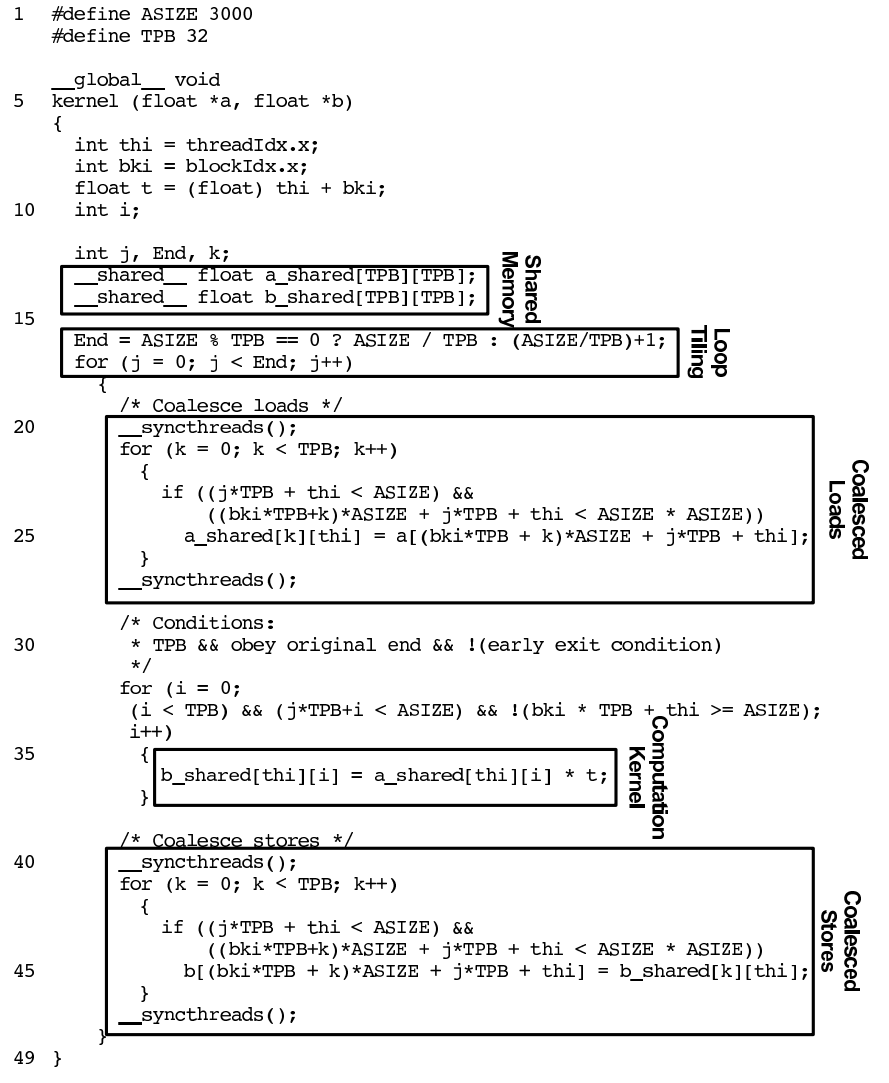


Fig. 4. Example Code: Hand Coalesced Kernel

```
(a) __annotation (L"__global__ <threads per block> <thread blocks per SM>");
(b) __annotation (L"garray <name> <rank> <element size> <rank sizes>");
(c) __annotation (L"BoundChk");
(d) __annotation (L"loop <iterator> <start> <end> <increment>");
```

Fig. 5. CUDA-lite Annotations

from the programmer provided via annotations to perform our transformations. We call the software tools and annotations together *CUDA-lite*.

Figure 5 shows the current form of the annotations in CUDA-lite. Part (a) indicates the functions of interest, i.e. kernel functions running on the GPU, and parallelization factors. While some of the information, like threads per TB, can eventually be derived from CUDA code, the last argument gives programmers some control over how much resources a kernel generated by CUDA-lite should take. Part (b) indicates what arrays in global memory are of interest and their properties. This gives control over which memory accesses are targeted for optimization, which uses up resources. The speedup gained from performing memory coalescing needs to be balanced against excessive resource usage that reduces executing parallelism. We will discuss this in detail in Section 4. Part (c) is for annotating exit checks, such as the conditional check on line 12 of Figure 2 mentioned in Section 2.1. While CUDA threads may terminate early, CUDA-lite may need those threads to satisfy memory coalescing and synchronization requirements. Therefore CUDA-lite removes the early termination and places guards around the original computation, as mentioned in Section 2.3. Finally, part (d) conveys information about the control flow of loops in the program. We currently use this information to perform loop transformations.

We recognize that some of the information provided by the annotations is derivable by advanced compiler techniques. However, the point of the annotations was to quickly provide the additional information needed and enable the transformations so that the memory hierarchy optimization automation work can proceed. It is not necessarily the final form.

Requirement 2 of the four requirements detailed in Section 2.3 is the most difficult to satisfy and check for. CUDA-lite derives the expression used in global memory accesses by performing a backwards dataflow up to the parameters of the kernel function and thread indices. The expression is first simplified by extracting all references to the thread index in the x direction. We leverage the SIMD execution model to eliminate the need for temporal locality checks, since the execution model guarantees that the expression is the same for all threads in the warp. The desired expression is one where every thread in a half-warp accesses the same location, differing only by their order within the half-warp. Consequently, any instance of $\lfloor thi.x/hwarp \rfloor$ can be safely disregarded, where $thi.x$ is the thread index in the x dimension and $hwarp$ is the number of threads in a half-warp. Mathematically this can be seen as the function f in Equation 1.

As long as the expression fits the form of the function, then the memory access is coalesced.

$$f(thi.x) = thi.x + g\left(\left\lfloor \frac{thi.x}{hwarp} \right\rfloor\right) + C \quad (1)$$

Figure 6(a) shows the relevant pseudo-code and expression generated by CUDA-lite for the memory access to array `a` in Figure 2. Due to the `ASIZE` multiplier on the first term, the expression does not fit function f and thus the load is not coalesced. Part (b) shows the memory access to array `a` in Figure 4. Unlike part (a), the expression does fit the form of the function f and therefore the access is coalesced.

If the memory access is not already coalesced, CUDA-lite will attempt to automatically generate a coalescing version. The labels of the additional boxes in Figure 4 outline the majority of the transformations: inserting shared memory variables, performing loop tiling, generating memory coalesced loads and/or stores, and replacing the original global memory accesses with accesses to the corresponding data in shared memory.

The shared memory size and tiling factor are fixed and known for each target GPU, due to the half-warp requirement for memory coalescing. The amount of shared memory allocated can thus be determined by the number of arrays of interest, array dimensions, and array element size. The generation of coalescing loads or stores depends on the relationship between the array dimension and the threading dimension. If they match, then CUDA-lite needs to have each thread load from the appropriate place in global memory into the thread's corresponding position in shared memory. If the array is of higher dimension than the thread organization, two-dimension to one dimension in the running example, then CUDA-lite generates loops that load/store the data. This can be seen in the Coalesced Loads and Stores boxes of Figure 4. These loops must not only be tiled correctly for correct data movement but they must also obey the array bounds.

Pseudo-code:

```
for (i = 0 to ASIZE)        // line 15, Figure 2
  a[(bki*TPB+thix)*ASIZE+i] // line 18, Figure 2
```

Expression:

$$ASIZE*thix + i + a + TPB*ASIZE*bki$$

(a)

Pseudo-code:

```
for (j = 0 to End)                // line 17, Figure 4
  for (k = 0 to TPB)              // line 20, Figure 4
    a[(bki*TPB+k)*ASIZE+j*TPB+thi] // line 18, Figure 4
```

Expression:

$$thix + ASIZE*k + TPB*j + a + TPB*ASIZE*bki$$

(b)

Fig. 6. Array Access and Expression (a) Non-Coalescing (b) Coalescing

```
1   #define ASIZE 3000
    #define TPB 32

    void
5   kernel (float *a, float *b)
    {
      __annotation (L"__global__ TPB 1");
      __annotation (L"garray a 2 4 ASIZE ASIZE");
      __annotation (L"garray b 2 4 ASIZE ASIZE");
10
      int thi = threadIdx.x;
      int bki = blockIdx.x;
      float t = (float) thi + bki;
      int i;
15
      __annotation (L"BoundChk");
      if (bki * TPB + thi >= ASIZE)
        return;

20    for (i = 0; i < ASIZE; i++)
        {
          __annotation (L"loop i 0 ASIZE 1");
          b[(bki*TPB+thi)*ASIZE + i] =
            a[(bki*TPB+thi)*ASIZE + i] * t;
25      }
    }
```

Fig. 7. Example Code: CUDA-lite Kernel

Figure 7 shows how the example kernel would be annotated using the current implementation of CUDA-lite. The programmer only needs to insert the five boxed additional lines instead of doubling the amount of code like in Figure 4.

It is important to note that CUDA-lite does not affect parallelization and threading decisions, and operates under the constraints of how the program has been parallelized. This was a deliberate decision to make the problems that CUDA-lite is tackling more tractable. CUDA-lite can be folded into a more comprehensive programming framework for GPU computing system as the part that handles memory optimization.

4 Experimental Results

We have implemented CUDA-lite using the Phoenix compiler [7] as a source-to-source compiler using two Phoenix plug-ins: one to perform the necessary analysis and code transformations, and another to generate source code back from the IR. The regenerated source code is then fed into NVIDIA's compiler nvcc to generate binaries for execution. We used CUDA version 1.0 for all of our experiments. The CPU was an Opteron 248 system running at 2.2GHz with 1GB of memory. The GPU was a GeForce 8800 GTX. The source codes for Phoenix are straightforward CUDA implementations that use only global memory, with slight manipulations so the CUDA extensions not recognized by Phoenix can be passed through and regenerated correctly.

We present three applications as our benchmark: MRI-FHD, TPACF, and the running example of this paper. These three applications display differences in the arrays to be optimized (e.g. 1-D and 2-D) and the level of control flow sophistication (e.g. loop nesting) that CUDA-lite had to handle. MRI-FHD is one of the compute intensive portions of three-dimensional MRI Reconstruction, of which details can be found in [16]. TPACF stands for the two-point angular

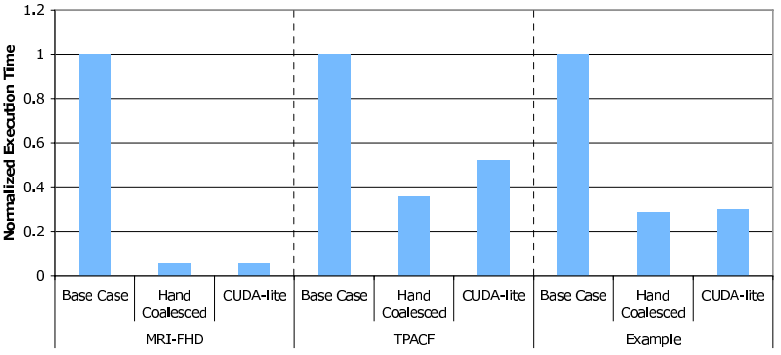


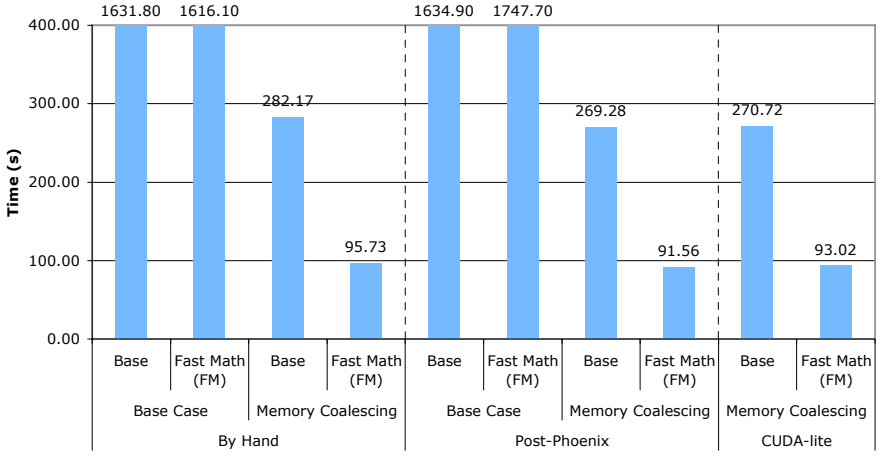
Fig. 8. Overall Results

correlation function, which is used to characterize the probability of finding a cosmological object at a given distance from another cosmological body. A more detailed description of the algorithm can be found in [3]. Both of these programs experienced terrific speedup moving from CPU to GPU [14].

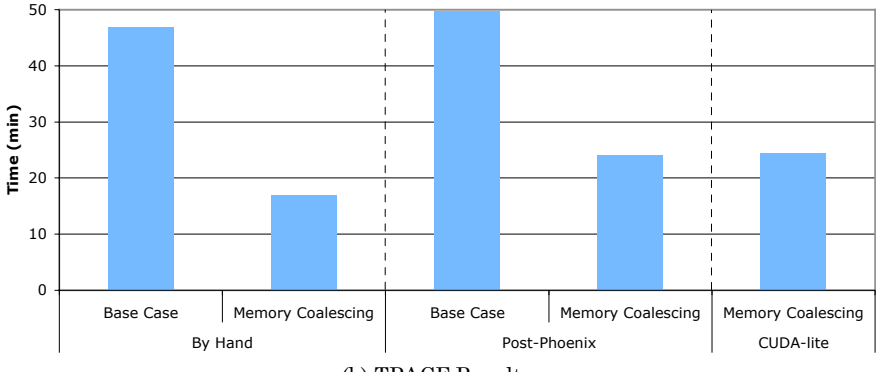
Figure 8 shows the overall results for our benchmarks. The run times are normalized to the base case of the application implemented in CUDA utilizing only global memory. For each application we show base, hand-coalesced, and CUDA-lite results. It is obvious how important improving the memory performance can be, providing between 2 to 17x performance difference in these studies. CUDA-lite, despite being generated from an automated tool, provided performance comparable to the hand-generated versions for all of the applications. We explain the discrepancy of the results between hand-generated and CUDA-lite-generated code in Section 4.1.

Figure 9 shows the detailed results of our experiments. Part (a) is MRI-FHD. Fast math is a compiler option in nvcc to utilize the hardware special function units (SFUs) on the GeForce 8800. This is very beneficial for MRI-FHD because its sine and cosine calculations can be performed on the SFUs. We present three sets of data: Code generated by hand, passing hand-generated code through Phoenix (Post-Phoenix), and CUDA-lite. The second set of data gives an idea of the overhead for going through a translation tool, and provides a more appropriate comparison for CUDA-lite. Note that Post-Phoenix and CUDA-lite memory coalescing code out-perform hand-generated. Fewer registers per thread were allocated by nvcc for the Post-Phoenix and CUDA-lite codes than the hand-generated code, which allowed two TBs to run concurrently on an SM. Otherwise the register allocation allowed only one. Part(b) shows the details of the TPACF results. Although the performance of CUDA-lite is the same as Post-Phoenix, they are both worse than hand-generated. Only one TB could run on an SM at a time in all cases due to shared memory usage.

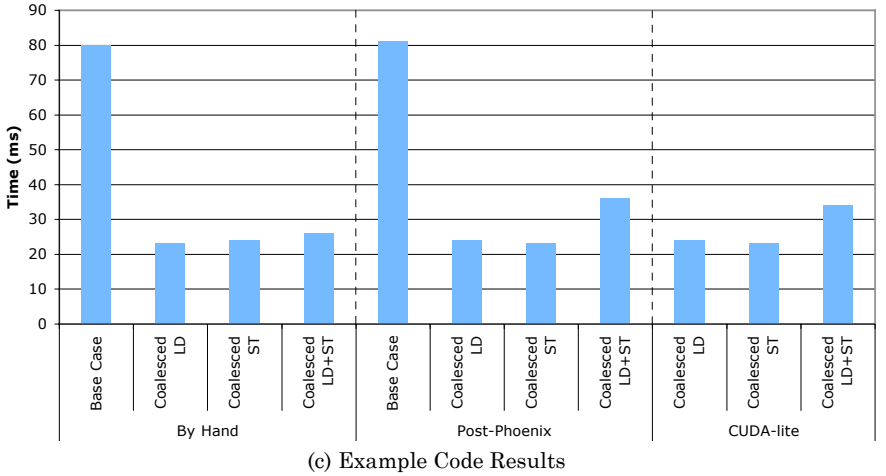
Figure 9(c) shows the results for the running example code in this paper. We compare the benefits of load coalescing, store coalescing, and both. There are



(a) MRI-FHD Results



(b) TPACF Results



(c) Example Code Results

Fig. 9. Detailed Results: (a) MRI-FHD (b) TPACF (c) Example Code

two arrays in the example code. Array `a` is read while array `b` is stored to. Using the annotations in CUDA-lite, we control which accesses are coalesced by the tool. The results indicate that coalescing either the load or the store is better than coalescing both. When only one access to one array is coalesced, up to three TBs can run concurrently on an SM. When accesses to both arrays are coalesced, the amount of shared memory used is doubled and the number of TBs running is reduced to one. Consequently, automatically coalescing all memory accesses is not always a good policy. Resource usage and overall performance need to be taken into account, perhaps in a performance optimization search like in [15].

4.1 Post-phoenix Overhead

Intuitively, regenerating source code from a compiler should add some amount of overhead. Curiously, our results show that this does not always translate into performance loss. Going through Phoenix showed no ill effect for MRI-FHD, a visible slowdown in TPACF, and mixed results in the example code. We narrowed down the problem to a combination of control flow and executing parallelism.

The output of Phoenix uses only GOTO statements to express the control flow of the program. This results in poor performance on CUDA. We verified this by manually generating versions that consist of only GOTO statements for control flow and observed similar degradations in performance. This explains the slowdown of TPACF and coalesced LD+ST in the example code. Multiple TBs executing on an SM provides additional parallelism to mask this overhead in MRI-FHD.

5 Related Work

Techniques have been proposed to allow array-dominated applications to benefit from scratch-pad memories [5,12]. In [2], the authors used the polyhedral model to detect data locality and copy the portion of data that is going to be used in a tile into the shared memory (or "scratch-pad memory") of a GPU. Our motivation and approach is different as we copy data from global memory to shared memory even if there is no data reuse. This is due to the significant performance benefit of coalescing global memory accesses on the GPU architecture.

Related techniques have also been developed to manipulate data accesses for SIMD devices [13,18]. SIMD units typically operate on short vectors, as opposed to the large massively parallel arrays that CUDA prefers. Also, memory coalescing has to be linear access since that is the requirement from the programming model. Data permutation and rearrangement would apply to setting up the data outside of the GPU kernel, or detecting that the data usage within the kernel covers data in such a way that interaction with the array should be coalesced.

We performed loop transformations such as tiling to properly reorganize the execution pattern. Wolf et al. [17] covered the loop transformations that enhance

data locality in loop nests. We decided not to automate the tiling transformations and rely upon programmer annotations instead since that was not the focus of our work. Our approach is not the same as the multi-level tiling schemes presented in [4,6], but we share the view that having a global knowledge of data access patterns facilitates improving locality in higher levels of memory hierarchy and increases global memory bandwidth performance.

There exists a body of work that incorporates programmer knowledge in performing transformations. Among these, the Spec# system by Microsoft [1] is closest to our work. It utilizes annotations to allow a separate verifying compiler to check for program correctness. Our annotations are information that feed directly into compiler analyses and transformations, usually information that would otherwise be missing or difficult to infer automatically by the compiler.

6 Conclusion and Future Work

In this paper we introduced CUDA-lite to help relieve programmers of the burden of optimizing the memory performance of code developed under the CUDA programming environment for GPU, which offers a complex memory hierarchy that needs to be leveraged to best match memory bandwidth with compute throughput. This is an important task due to the large effect memory performance has on overall performance (2-17x).

We show that CUDA-lite produces code with performance comparable to hand-coded versions. The coding requirements for CUDA-lite are lower than performing the same transformations by hand and provides a layer of abstraction from the definition of warps in CUDA, which could change in the future. Since CUDA-lite does not handle the parallelizing aspects of GPU programming, we foresee CUDA-lite as the memory optimizing module of an eventual overall framework for facilitating GPGPU programming that encompasses parallelization and resource usage decisions to maximize performance.

For future work we plan to broaden the application set and to extend CUDA-lite to leverage constant memory. We also hope to simplify the annotations in CUDA-lite, some of which can be replaced by compiler analyses currently not in our infrastructure.

Acknowledgment

We would like to thank David Kirk and NVIDIA for generous hardware loans and support. We also thank the anonymous reviewers for their feedback. The authors acknowledge the support of the Gigascale Systems Research Center, funded under the Focus Center Research Program, a Semiconductor Research Corporation program. Experiments were made possible by NSF CNS grant 05-51665. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF. This work was performed with software donations from Microsoft.

References

1. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
2. Baskaran, M.M., Bondhugula, U., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P.: Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In: PPoPP 2008: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2008)
3. Brunner, R.J., Kindratenko, V.V., Myers, A.D.: Developing and deploying advanced algorithms to novel supercomputing hardware. In: Proceedings of NASA Science Technology Conference - NCTC 2007 (2007)
4. Guo, J., Bikshandi, G., Fraguela, B.B., Garzaran, M.J., Padua, D.: Programming with tiles. In: PPoPP 2008: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2008)
5. Kandemir, M., Choudhary, A.: Compiler-directed scratch pad memory hierarchy design and management. In: DAC 2002: Proceedings of the 39th Conference on Design Automation (2002)
6. Knight, T.J., Park, J.Y., Ren, M., Mike, H., Erez, M., Fatahalian, K., Aiken, A., Dally, W.J., Hanrahan, P.: Compilation for explicitly managed memory hierarchies. In: Proceedings of the 2007 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2007)
7. Microsoft. Phoenix compiler, `http://research.microsoft.com/Phoenix/`
8. Nickolls, J., Buck, I.: NVIDIA CUDA software and GPU parallel computing architecture. Microprocessor Forum (May 2007)
9. NVIDIA. NVIDIA CUDA, `http://www.nvidia.com/cuda`
10. NVIDIA. NVIDIA CUDA Compute Unified Device Architecture Programming Guide: Version 1.0. NVIDIA Corporation (June 2007)
11. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A.E., Purcell, T.J.: A survey of general-purpose computation on graphics hardware. Computer Graphics Forum 26(1), 80–113 (2007)
12. Panda, P.R., Dutt, N.D., Nicolau, A.: Efficient utilization of scratch-pad memory in embedded processor applications. In: EDTC 1997: Proceedings of the 1997 European Conference on Design and Test (1997)
13. Ren, G., Wu, P., Padua, D.A.: Optimizing data permutations for SIMD devices. In: PLDI, pp. 118–131 (2006)
14. Ryoo, S., Rodrigues, C.I., Baghsorkhi, S.S., Stone, S.S., Kirk, D.B., Hwu, W.W.: Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In: PPoPP, pp. 73–82 (2008)
15. Ryoo, S., Rodrigues, C.I., Stone, S.S., Baghsorkhi, S.S., Ueng, S., Stratton, J.A., Hwu, W.W.: Program optimization space pruning for a multithreaded GPU. In: CGO (April 2008)
16. Stone, S.S., Haldar, J.P., Tsao, S.C., Hwu, W.W., Liang, Z., Sutton, B.P.: Accelerating advanced MRI reconstructions on GPUs. In: Proceedings of the 2008 International Conference on Computing Frontiers (May 2008)
17. Wolf, M.E., Lam, M.S.: A data locality optimizing algorithm. In: PLDI 1991: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (1991)
18. Wu, P., Eichenberger, A.E., Wang, A., Zhao, P.: An integrated simdization framework using virtual vectors. In: ICS, pp. 169–178 (2005)

MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs

John A. Stratton, Sam S. Stone, 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
{stratton,ssstone2,hwu}@crhc.uiuc.edu

Abstract. CUDA is a data parallel programming model that supports several key abstractions - thread blocks, hierarchical memory and barrier synchronization - for writing applications. This model has proven effective in programming GPUs. In this paper we describe a framework called MCUDA, which allows CUDA programs to be executed efficiently on shared memory, multi-core CPUs. Our framework consists of a set of source-level compiler transformations and a runtime system for parallel execution. Preserving program semantics, the compiler transforms threaded SPMD functions into explicit loops, performs fission to eliminate barrier synchronizations, and converts scalar references to thread-local data to replicated vector references. We describe an implementation of this framework and demonstrate performance approaching that achievable from manually parallelized and optimized C code. With these results, we argue that CUDA can be an effective data-parallel programming model for more than just GPU architectures.

1 Introduction

In February of 2007, NVIDIA released the CUDA programming model for use with their GPUs to make them available for general purpose application programming [1]. However, the adoption of the CUDA programming model has been limited to those programmers willing to write specialized code that only executes on certain GPU devices. This is undesirable, as programmers who have invested the effort to write a general-purpose application in a data-parallel programming language for a GPU should not have to make an entirely separate programming effort to effectively parallelize the application across multiple CPU cores.

One might argue that CUDA's exposure of specialized GPU features limits the efficient execution of CUDA kernels to GPUs. For example, in a typical usage case of the CUDA programming model, programmers specify hundreds to thousands of small, simultaneously active threads to achieve full utilization of GPU execution resources. However, a current CPU architecture currently supports only up to tens of active thread contexts. On the other hand, some language features in the CUDA model can be beneficial to performance on a CPU, because these features encourage the programmer to use more disciplined

J.N. Amaral (Ed.): LCPC 2008, LNCS 5335, pp. 16–30, 2008.

control flow and expose data locality. Section 2 describes in more detail the key CUDA language features and a deeper assessment of why we expect many CUDA features to map well to a CPU architecture for execution. We propose that if an effective mapping of the CUDA programming model to a CPU architecture is feasible, it would entail translations applied to a CUDA program such that the limiting features of the programming model are removed or mitigated, while the beneficial features remain exposed when possible.

Section 3 describes how the MCUDA system translates a CUDA program into an efficient parallel C program. Groups of individual CUDA threads are collected into a single CPU thread while still obeying the scheduling restrictions of barrier synchronization points within the CUDA program. The data locality and regular control encouraged by the CUDA programming model are maintained through the translation, making the resulting C program well suited for a CPU architecture.

The implementation and experimental evaluation of the MCUDA system is presented in Section 4. Our experiments show that optimized CUDA kernels utilizing MCUDA achieve near-perfect scaling with the number of CPU cores, and performance comparable to hand-optimized multithreaded C programs. We conclude this paper with a discussion of related work in Section 5 and some closing observations in Section 6.

2 Programming Model Background

On the surface, most features included in the CUDA programming model seem relevant only to a specific GPU architecture. The primary parallel construct is a data-parallel, SPMD *kernel* function. A kernel function invocation explicitly creates many CUDA threads (hereafter referred to as *logical threads*.) The threads are organized into multidimensional arrays that can synchronize and quickly share data, called thread *blocks*. These thread blocks are further grouped into another multidimensional array called a *grid*. Logical threads within a block are distinguished by an implicitly defined variable *threadIdx*, while blocks within a grid are similarly distinguished by the implicit variable *blockIdx*. At a kernel invocation, the programmer uses language extensions to specify runtime values for each dimension of threads in a thread block and each dimension of thread blocks in the grid, accessible within the kernel function through the variables *blockDim* and *gridDim* respectively. In the GPU architecture, these independent thread blocks are dynamically assigned to parallel processing units, where the logical threads are instantiated by hardware threading mechanisms and executed.

Logical threads within CUDA thread blocks may have fine-grained execution ordering constraints imposed by the programmer through barrier synchronization intrinsics. Frequent fine-grained synchronization and data sharing between potentially hundreds of threads is a pattern in which CPU architectures typically do not achieve good performance. However, the CUDA programming model does restrict barrier synchronization to within thread blocks, while different thread blocks can be executed in parallel without ordering constraints.

The CUDA model also includes explicitly differentiated memory spaces to take advantage of specialized hardware memory resources, a significant departure from the unified memory space of CPUs. The *constant* memory space uses a small cache of a few kilobytes optimized for high temporal locality and accesses by large numbers of threads across multiple thread blocks. The *shared* memory space maps to the scratchpad memory of the GPU, and is shared among threads in a thread block. The *texture* memory space uses the GPU's texture caching and filtering capabilities, and is best utilized with data access patterns exhibiting 2-D locality. More detailed information about GPU architecture and how features of the CUDA model affect application performance is presented in [2].

In the CUDA model, logical threads within a thread block can have independent control flow through the program. However, the NVIDIA G80 GPU architecture executes logical threads in SIMD bundles called *warps*, while allowing for divergence of thread execution using a stack-based reconvergence algorithm with masked execution [3]. Therefore, logical threads with highly irregular control flow execute with greatly reduced efficiency compared to a warp of logical threads with identical control flow. CUDA programmers are strongly encouraged to adopt algorithms that force logical threads within a thread block to have very similar, if not exactly equivalent, execution traces to effectively use the implicitly SIMD hardware effectively. In addition, the CUDA model encourages data locality and reuse for good performance on the GPU. Accesses to the global memory space incur uniformly high latency, encouraging the programmer to use regular, localized accesses through the scratchpad shared memory or the cached constant and texture memory spaces.

A closer viewing of the CUDA programming model suggests that there could also be an efficient mapping of the execution specified onto a current multi-core CPU architecture. At the largest granularity of parallelism within a kernel, blocks can execute completely independently. Thus, if all logical threads within a block occupy the same CPU core, there is no need for inter-core synchronization during the execution of blocks. Thread blocks often have very regular control flow patterns among constituent logical threads, making them amenable to the SIMD instructions common in current x86 processors [4,5]. In addition, thread blocks often have the most frequently referenced data specifically stored in a set of thread-local or block-shared memory locations, which are sized such that they approximately fit within a CPU core's L1 data cache. Shared data for the entire kernel is often placed in constant memory with a size limit appropriate for an L2 cache, which is frequently shared among cores in CPU architectures. If a translation can be designed such that these attributes are maintained, it should be possible to generate effective multithreaded CPU code from the CUDA specification of a program.

3 Kernel Translation

While the features of the model seem promising, the mapping of the computation is not straightforward. The conceptually easiest implementation is to spawn an

OS thread for every GPU thread specified in the programming model. However, allowing logical threads within a block to execute on any available CPU core mitigates the locality benefits noted in the previous section, and incurs a large amount of scheduling overhead. Therefore, we propose a method of translating the CUDA program such that the mapping of programming constructs maintains the locality expressed in the programming model with existing operating system and hardware features.

There are several challenging goals in effectively translating CUDA applications. First, each thread block should be scheduled to a single core for locality, yet maintain the ordering semantics imposed by potential barrier synchronization points. Without modifying the operating system or architecture, this means the compiler must somehow manage the execution of logical threads in the code explicitly. Second, the SIMD-like nature of the logical threads in many applications should be clearly exposed to the compiler. However, this goal is in conflict with supporting arbitrary control flow among logical threads. Finally, in a typical load-store architecture, private storage space for every thread requires extra instructions to move data in and out of the register file. Reducing this overhead requires identifying storage that can be safely reused for each thread.

The translation component of MCUDA which addresses these goals is composed of three transformation stages: iterative wrapping, synchronization enforcement, and data buffering. For purposes of clarity, we consider only the case of a single kernel function with no function calls to other procedures, possibly through exhaustive inlining. It is possible to extend the framework to handle function calls with an interprocedural analysis, but this is left for future work. In addition, without loss of generality, we assume that the code does not contain *goto* or *switch* statements, possibly through prior transformation [6]. All transformations presented in this paper are performed on the program's abstract syntax tree (AST).

3.1 Transforming a Thread Block into a Serial Function

The first step in the transformation changes the nature of the kernel function from a per-thread code specification to a per-block code specification, temporarily ignoring any potential synchronization between threads. Figure 1 shows an example kernel function before and after this transformation. Execution of logical threads is serialized using nested loops around the body of the kernel function to execute each thread in turn. The loops enumerate the values of the previously implicit threadIdx variable and perform a logical thread's execution of the enclosed statements on each iteration. For the remainder of the paper, we will consider this introduced iterative structure a *thread loop*. Local variables are reused on each iteration, since only a single logical thread is active at any time. Shared variables still exist and persist across loop iterations, visible to all logical threads. The other implicit variables must be provided to the function at runtime, and are therefore added to the parameter list of the function.

By introducing a thread loop around a set of statements, we are making several explicit assumptions about that set of statements. The first is that the

```
void cengery(numatoms, gridspacing, energygrid[] )
{
        int x = blockIdx.x * blockDim.x
                        + threadIdx.x;
        int y = blockIdx.y * blockDim.y
                        + threadIdx.y;
        int outIdx = gridDim.x * blockDim.x * y
                        + x;

        float energy = 0.0;
        int atomid=0;
        while(atomid<numatoms) {
            …
        }
        energygrid[outIdx] = energy;
}
```

```
void cengery(numatoms, gridspacing, energygrid[],
                blockDim, blockIdx, gridDim)
{
    dim3 threadIdx;
    // Thread Loop
    for(threadIdx.y = 0;
          threadIdx.y < blockDim.y;
          threadIdx.y++)
      for(threadIdx.x = 0;
            threadIdx.x < blockDim.x;
            threadIdx.x++)
    {
        int x = blockIdx.x * blockDim.x
                        + threadIdx.x;
        int y = blockIdx.y * blockDim.y
                        + threadIdx.y;
        int outIdx = gridDim.x*blockDim.x * y
                        + x;

        float energy = 0.0;
        int atomid=0;
        while(atomid < numatoms) {
            …
        }
        energygrid[outIdx] = energy;
    }
    // end Thread Loop;
}
```

Fig. 1. Introducing a thread loop to serialize logical threads in Coulombic Potential

program allows each logical thread to execute those statements without any synchronization between threads. The second is that there can be no side entries into or side exits out of the thread loop body. If the programmer has not specified any synchronization point and the function contains no explicit return statement, no further transformation is required, as a function cannot have side entry points, and full inlining has removed all side-exits. In the more general case, where using a single thread loop is insufficient for maintaining program semantics, we must partition the function into sets of statements which do satisfy these properties.

3.2 Enforcing Synchronization with Deep Fission

A thread loop implicitly introduces a barrier synchronization among logical threads at its boundaries. Each logical thread executes to the end of the thread loop, and then "suspends" until every other logical thread (iteration) completes the same set of statements. Therefore, a loop fission operation essentially partitions the statements of a thread loop into two sets of statements with an implicit barrier synchronization between them. A synchronization point found in the immediate scope of a thread loop can be thus enforced by applying a loop fission operation at the point of synchronization.

Although a loop fission operation applied to the thread loop enforces a barrier synchronization at that point, this operation can only be applied at the scope of the thread loop. As mentioned before, the general case requires a transformation that partitions statements into thread loops such that each thread loop contains no synchronization point, and each thread loop boundary is a valid synchronization point. For example, consider the case of Figure 2. There are at minimum four groups of statements required to satisfy the requirements for thread loops: one leading up to the for loop (including the loop initialization

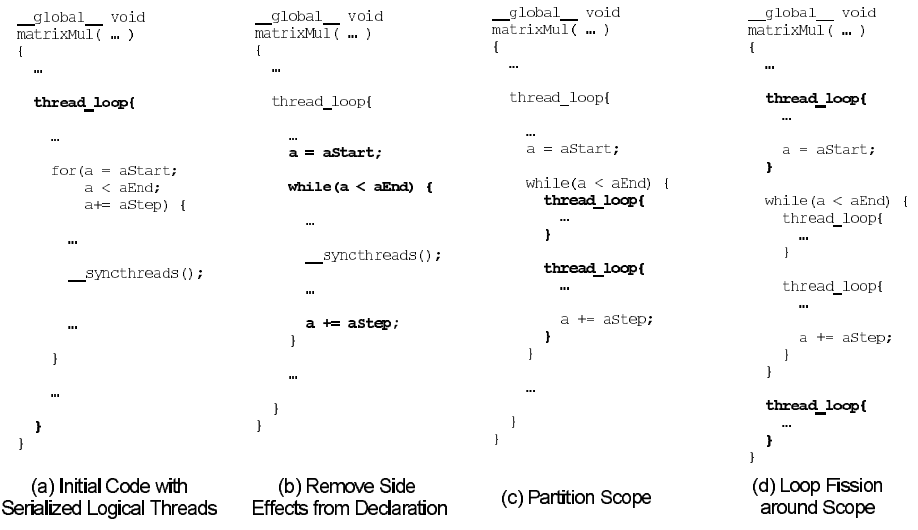


Fig. 2. Applying deep fission in Matrix Multiplication to enforce synchronization

statement), one for the part of the loop before the synchronization point, one after the synchronization point within the loop (including the loop update), and finally the trailing statements after the loop.

In this new set of thread loops, the logical threads will implicitly synchronize every time the loop conditional is evaluated, in addition to the programmer-specified synchronization point. This is a valid transformation because of the CUDA programming model's requirement that control flow affecting a synchronization point must be thread-independent. This means that if the execution of a synchronization point is control-dependent on a condition, that condition must be thread-invariant. Therefore, if any thread arrives at the conditional evaluation, all threads must reach that evaluation, and furthermore must evaluate the conditional in the same way. Such a conditional can be evaluated outside of a thread loop once as a representative for all logical threads. In addition, it is valid to force all threads to synchronize at the point of evaluation, and thus safe to have thread loops bordering and comprising the body of the control structure.

In describing our algorithm for enforcing synchronization points, we first assume that all control structures have no side effects in their declarations. We enforce that *for* loops must be transformed into *while* loops in the AST, removing the initialization and update expressions. In addition, all conditional evaluations with side effects must be removed from the control structure's declaration, and assigned to a temporary variable, which then replaces the original condition in the control structure. Then, for each synchronization statement S, we apply Algorithm 1 to the AST with S as the input parameter.

After this algorithm has been applied with each of the programmer-specified synchronization points as input, the code may still have some control flow for which the algorithm has not properly accounted. Recall that thread loops assume

Algorithm 1. Deep Fission around a Synchronization Statement S

loop
  if Immediate scope containing S is not a thread loop **then**
    Partition all statements within the scope containing S into thread loops. Statements before and after S form two thread loops. In the case of an if-else construct, this also means all statements within the side of the construct not containing S are formed into an additional thread loop. {See Figure 2(c)}
  else
    Apply a loop fission operation to the thread loop around S and return {(See Figure 2(d).)}
  end if
  $S \leftarrow$ Construct immediately containing S {Parent of S in the AST}
end loop

that there are no side entries or side exits within the thread loop body. Control flow statements such as *continue*, *break*, or *return* may not be handled correctly when the target of the control flow is not also within the thread loop. Figure 3(b) shows a case where irregular control flow would result in incorrect execution. In some iterations of the outer loop, all logical threads may avoid the break and synchronize correctly. In another iteration, all logical threads may take the break, avoiding synchronization. However, in the second case, control flow would leave the first thread loop before all logical threads had finished the first thread loop, inconsistent with the program's specification. Again, we note that since the synchronization point is control-dependent on the execution of the break statement, the break statement itself can be a valid synchronization point according to the programming model.

Therefore, the compiler must pass through the AST at least once more to identify these violating control flow statements. At the identification of a control flow statement S whose target is outside its containing thread loop, Algorithm 1 is once again applied, treating S as a synchronization statement. For the example of Figure 3, this results in the code shown in Figure 3(c). Since these

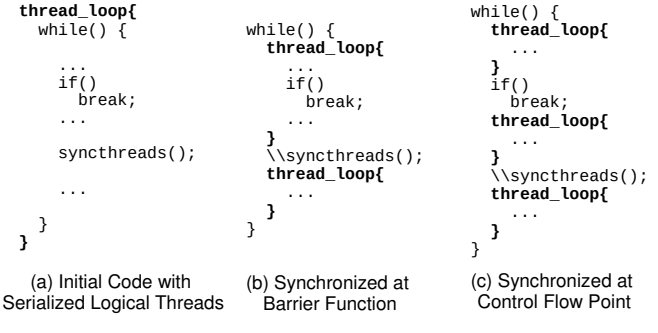


Fig. 3. Addressing unstructured control flow. The break statement is treated as an additional synchronization statement for correctness.

transformations more finely divide thread loops, they could reveal additional control flow structures that violate the thread loop properties. Therefore, this irregular control flow identification and synchronization step is applied iteratively until no additional violating control flow is identified.

The key insight is that we are not supporting arbitrary control flow among logical threads within a block, but leveraging the restrictions in the CUDA language to define a single-threaded ordering of the instructions of multiple threads which satisfies the partial ordering enforced by the synchronization points. This "over-synchronizing" allows us to completely implement a "threaded" control flow using only iterative constructs within the code itself. The explicit synchronization primitives may now be removed from the code, as they are guaranteed to be bounded by thread loops on either side, and contain no other computation. Because only barrier synchronization primitives are provided in the CUDA programming model, no further control-flow transformations to the kernel function are needed to ensure proper ordering of logical threads. Figure 4(a) shows the matrix multiplication kernel after this hierarchical synchronization procedure has been applied.

3.3 Replicating Thread-Local Data

Once the control flow has been restructured, the final task remaining is to buffer the declared variables as needed. Shared variables are declared once for the entire block, so their declarations simply need the *shared* keyword removed. However, each logical thread has a local store for variables, independent of all other logical threads. Because these logical threads no longer exist independently, the translated program must emulate private storage for logical threads within the block. The simplest implementation creates private storage for each thread's instance of the variable, analogous to scalar expansion [7]. This technique, which we call *universal replication*, fully emulates the local store of each logical thread by creating an array of values for each local variable, as shown in Figure 4(b). Statements within thread loops access these arrays by thread index to emulate the logical thread's local store.

However, universal replication is often unnecessary and inefficient. In functions with no synchronization, thread loops can completely serialize the execution of logical threads, reusing the same memory locations for local variables. Even in the presence of synchronization, some local variables may have live ranges completely contained within a thread loop. In this case, logical threads can still reuse the storage locations of those variables because a value of that variable is never referenced outside the thread loop in which it is defined. For example, in the case of Figure 4(b), the local variable k can be safely reused, because it is never live outside the third thread loop.

Therefore, to use less memory space, the MCUDA framework should only create arrays for local variables when necessary. A live-variable analysis determines which variables have a live value at the end of a thread loop, and creates arrays for those values only. This technique, called *selective replication*, results in the code shown in Figure 4(c), which allows all logical threads to use the same

```
__global__ void
matrixMul( float* C, float* A, float* B)
{
  int a, b, c, aEnd, k;
  float Csub;
  __shared__ float As[16][16];
  __shared__ float Bs[16][16];

  thread_loop{
    aEnd = Awidth * threadIdx.y + threadIdx.x;
    a = Awidth * BLOCK_SIZE * blockIdx.y + aEnd;
    b = BLOCK_SIZE * blockIdx.x;
    b =  a + b;
    b = b + aEnd;
    aEnd = a + Awidth;
    Csub = 0;
  }
  while (a < aEnd) {
    thread_loop{
      As[threadIdx.y][threadIdx.x] = A[a];
      Bs[threadIdx.y][threadIdx.x] = B[b];
      a += BLOCK_SIZE;
      b += BLOCK_SIZE*Bwidth;
    }
    thread_loop{
      for (k = 0; k < BLOCK_SIZE; k++)
        Csub += As[threadIdx.y][k] *
                Bs[k][threadIdx.x];
    }
  }
  thread_loop{
    C[c] = Csub;
  }
}
```

(a) Synchronized Kernel

```
__global__ void
matrixMul( float* C, float* A, float* B)
{
  int a[], b[], c[], aEnd[], k[];
  float Csub[];
  float As[16][16];
  float Bs[16][16];

  thread_loop{
    aEnd[tid] = Awidth * threadIdx.y + threadIdx.x;
    a[tid] = Awidth * BLOCK_SIZE * blockIdx.y + aEnd[tid];
    b[tid] = BLOCK_SIZE * blockIdx.x;
    b[tid] =  a[tid] + b[tid];
    b[tid] = b[tid] + aEnd[tid];
    aEnd[tid] = a[tid] + Awidth;
    Csub[tid] = 0;
  }
  while (a[0] < aEnd[0]) {
    thread_loop{
        As[threadIdx.y][threadIdx.x] = A[a[tid]];
        Bs[threadIdx.y][tthreadIdx.x] = B[b[tid]];
        a[tid] += BLOCK_SIZE;
        b[tid] += BLOCK_SIZE*Bwidth;
    }
    thread_loop{
      for (k[tid] = 0; k[tid] < BLOCK_SIZE; k[tid]++)
        Csub[tid] += As[threadIdx.y][k[tid]] *
                     Bs[k[tid]][threadIdx.x];
    }
  }
  thread_loop{
    C[c[tid]] = Csub[tid];
  }
}
```

(b) Universal Replication

```
__global__ void
matrixMul( float* C, float* A, float* B)
{
  int a[], b[], c[], aEnd[], k;
  float Csub[];
  float As[16][16];
  float Bs[16][16];

  thread_loop{
    aEnd[tid] = Awidth * threadIdx.y + threadIdx.x;
    a[tid] = Awidth * BLOCK_SIZE * blockIdx.y + aEnd[tid];
    b[tid] = BLOCK_SIZE * blockIdx.x;
    b[tid] =  a[tid] + b[tid];
    b[tid] = b[tid] + aEnd[tid];
    aEnd[tid] = a[tid] + Awidth;
    Csub[tid] = 0;
  }
  while (a[0] < aEnd[0]) {
    thread_loop{
        As[threadIdx.y][threadIdx.x] = A[a[tid]];
        Bs[threadIdx.y][tthreadIdx.x] = B[b[tid]];
        a[tid] += BLOCK_SIZE;
        b[tid] += BLOCK_SIZE*Bwidth;
    }
    thread_loop{
      for (k = 0; k < BLOCK_SIZE; k++)
        Csub[tid] += As[threadIdx.y][k] *
                     Bs[k][threadIdx.x];
    }
  }
  thread_loop{
    C[c[tid]] = Csub[tid];
  }
}
```

(c) Selective Replication

Fig. 4. Data replication in Matrix Multiplication

memory location for the local variable k. However, a and b are defined and used across thread loop boundaries, and must be stored into arrays.

References to a variable outside of the context of a thread loop can only exist in the conditional evaluations of control flow structures. Control structures must affect synchronization points to be outside a thread loop, and therefore must be uniform across the logical threads in the block. Since all logical threads should

have the same logical value for conditional evaluation, we simply reference element zero as a representative, as exemplified by the while loop in Figure 4 (b-c).

It is useful to note that although CUDA defines separate memory spaces for the GPU architecture, all data resides in the same shared memory system in the MCUDA framework, including local variables. The primary purpose of the different memory spaces on the GPU is to specify access to the different caching mechanisms and the scratchpad memory. A typical CPU system provides a single, cached memory space, thus we map all CUDA memory types to this memory space.

3.4 Work Distribution and Runtime Framework

At this point in the translation process the kernels are now defined as block-level functions, and all that remains is, on kernel invocation, to iterate through the block indexes specified and call the transformed function once for every specified block index. For a CPU that gains no benefits from multithreading, this is an efficient way of executing the kernel computation. However, CPU architectures that do gain performance benefits from multithreading will likely not achieve full efficiency with this method. Since these blocks can execute independently according to the programming model, the set of block indexes may be partitioned arbitrarily among concurrently executing OS threads. This allows the kernels to exploit the full block-level parallelism expressed in the programming model.

4 Implementation and Performance Analysis

We have implemented the MCUDA automatic kernel translation framework under the Cetus source-to-source compilation framework [8], with slight modifications to the IR and preprocessor to accept ANSI C with the language extensions of CUDA version 0.8. MCUDA implements the algorithms presented in the previous section for kernel transformations and applies them to the AST intermediate representation of Cetus. The live variable analysis required for robust selective replication described in Section 3.3 is incomplete, but the current implementation achieves the same liveness results for all the applications presented in this section. For compatibility with the Intel C Compiler (ICC), we replace the CUDA runtime library with a layer interfacing to standard libc functions for memory management. We chose to implement the runtime assignment of blocks to OS threads with OpenMP, using a single "parallel for" pragma to express the parallelism. A large existing body of work explores scheduling policies of such loops in OpenMP and other frameworks [9,10,11,12], For our experiments, we use the default compiler implementation.

Figure 5 shows the kernel speedup of three applications: matrix multiplication of two 4kx4k element matrices (MM4K), Coulombic Potential (CP), and MRI-FHD, a computationally intensive part of high-resolution MRI image reconstruction. These applications have previously shown to have very efficient CUDA implementations on a GPU architecture [13]. The CPU baselines that

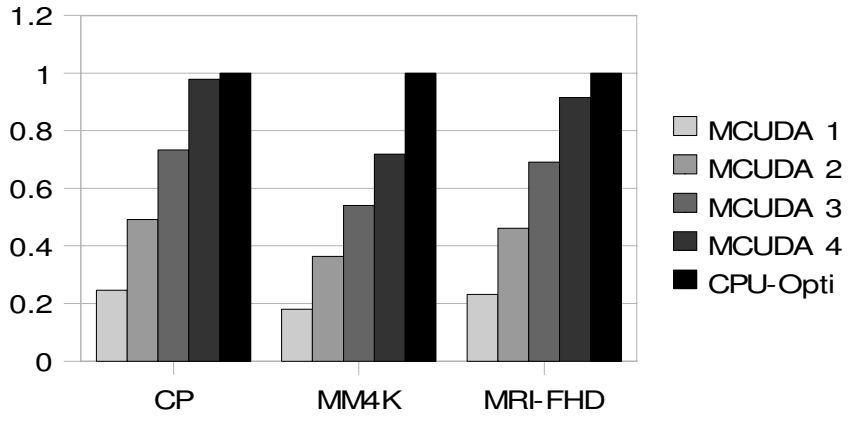


Fig. 5. Performance (inverse runtime) of MCUDA kernels relative to optimized CPU code. MCUDA results vary by the number of worker threads (1-4). CPU Opti implementations are parallelized across 4 threads.

we are measuring against are the most heavily optimized CPU implementations available to us, and are threaded by hand to make use of multiple CPU cores. All performance data was obtained on an Intel Core 2 Quad processor clocked at 2.66 GHz (CPU model Q6700). All benchmarks were compiled with ICC (version 10.1). Additionally, the CPU optimized matrix multiplication application uses the Intel MKL.

We can see that the performance scaling of this implementation is very good, with practically ideal linear scaling for a small number of processor cores. For each application, the performance of the CUDA code translated through the MCUDA framework is within 30% of the most optimized CPU implementation available. This suggests that the data tiling and locality expressed in effective CUDA kernels also gain significant performance benefits on CPUs, often replicating the results of hand-optimization for the CPU architecture. The regularly structured iterative loops of the algorithms were also preserved through the translation. The compiler vectorized the innermost loops of each application automatically, whether those were thread loops or loops already expressed in the algorithm.

Tuning CUDA kernels entails methodically varying a set of manual optimizations applied to a kernel. Parameters varied in this tuning process may include the number of logical threads in a block, unrolling factors for loops within the kernel, and tiling factors for data assigned to the scratchpad memory [14]. The performance numbers shown are the best results found by tuning each application. In the tuning process, we found that not all optimizations that benefit a GPU architecture are effective for compiling and executing on the CPU. In these applications, manual unrolling in the CUDA source code almost always reduced the effectiveness of the backend compiler, resulting in poorer performance. Optimizations that spill local variables to shared memory were also not particularly effective, since the shared memory and local variables reside in the same memory space on the CPU.

In general, the best optimization point for each application may be different depending on whether the kernel will execute on a GPU or CPU. The architectures have different memory systems, different ISAs, and different threading mechanisms, which make it very unlikely that performance tuning would arrive at the similar code configuration for these two architectures. For all the applications we have explored so far, this has always been the case. For example, the best performing matrix multiplication code uses per-block resources that are expressible in CUDA, but well over the hardware limits of current GPUs. The best CUDA code for the CPU uses 20KB of shared memory and 1024 logical threads per block, both over the hardware limits of current GPUs. Similarly, the best CP code uses an amount of constant memory larger than what a current GPU supports. Developing a system for tuning CUDA kernels to CPU architectures is a very interesting area of future work, both for programming practice and toolchain features.

For the benchmarks where the MCUDA performance is significantly below the best hand-tuned performance, we think that this is primarily because of algorithmic differences in the implementations. Projects like ATLAS have explored extensive code configuration searches far broader than we have considered in these experiments, and some of that work may be relevant here as well. People have achieved large speedups on GPU hardware with "unconventional" CUDA programming [15], and it is possible that more variations of CUDA code configurations may eventually bridge the current performance gap. The hand-tuned MRI-FHD implementation uses hand-vectorized SSE instructions across logical threads, whereas ICC vectorizes the innermost loop of the algorithm, seemingly with a minor reduction in efficiency. Future work should consider specifically targeting the thread loops for vectorization, and test the efficiency of such a transformation.

5 Related Work

With the initial release of the CUDA programming model, NVIDIA also released a toolset for GPU emulation [1]. However, the emulation framework was designed for debugging rather than for performance. In the emulation framework, each logical thread within a block is executed by a separate CPU thread. In contrast, MCUDA localizes all logical threads in a block to a single CPU thread for better performance. However, the MCUDA framework is less suitable for debugging the parallel CUDA application for two primary reasons. The first is that MCUDA modifies the source code before passing it to the compiler, so the debugger can not correlate the executable with the original CUDA source code. The second is that MCUDA enforces a specific scheduling of logical threads within a block, which would not reveal errors that could occur with other valid orderings of the execution of logical threads.

The issue of mapping small-granularity logical threads to CPU cores has been addressed in other contexts, such as parallel simulation frameworks [16]. There are also performance benefits to executing multiple logical threads within a single

CPU thread in that area. For example, in the Scalable Simulation Framework programming model, a CPU thread executes each of its assigned logical threads, jumping to the code specified by each in turn. Logical threads that specify suspension points must be instrumented to save local state and return execution to the point at which the logical thread was suspended. Taking advantage of CUDA's SPMD programming model and control-flow restrictions, MCUDA uses a less complex execution framework based on iteration within the originally threaded code itself. The technique used by MCUDA for executing logical threads can increase the compiler's ability to optimize and vectorize the code effectively. However, our technique is limited to SPMD programming models where each static barrier-wait instrinsic in the source code waits on a different thread barrier.

A large number of other frameworks and programming models have been proposed for data-parallel applications for multi-core architectures. Some examples include OpenMP [17], Thread Building Blocks [18], and HPF [19]. However, these models are intended to broaden a serial programming language to a parallel execution environment. MCUDA is distinct from these in that it is intended to broaden the applicability of a previously accelerator-specific programming model to a CPU architecture.

Liao et al. designed a compiler system for efficiently mapping the stream programming model to a multi-core architecture [20]. CUDA, while not strictly a stream programming model, shares many features with stream kernels. MCUDA's primary departure from mapping a stream programming model to multi-core architectures is the explicit use of data tiling and cooperative threading, which allows threads to synchronize and share data. With MCUDA, the programmer can exert more control over the kernels with application knowledge, rather than relying on the toolset to discover and apply them with kernel merging and tiling optimizations. It is also unclear whether the range of optimizations available in the CUDA programming model can be discovered and applied by an automated framework.

6 Conclusions

We have described techniques for efficiently implementing the CUDA programming model on a conventional multi-core CPU architecture. We have also implemented an automated framework that applies these techniques, and tested it on some kernels known to have high performance when executing on GPUs. We have found that for executing these translated kernels on the CPU, the expression of data locality and computational regularity in the CUDA programming model achieves much of the performance benefit of tuning code for the architecture by hand. These initial results suggest that the CUDA programming model could be a very effective way of specifying data-parallel computation in a programming model that is portable across a variety of parallel architectures.

As the mapping of the CUDA language to a CPU architecture matures, we expect that the performance disparity between optimized C code and optimized

CUDA code for the CPU will continue to close. As with any other level of software abstraction, there are more opportunities for optimization at lower levels of abstraction. However, if expressing computation in the CUDA language allows an application to be more portable across a variety of architectures, many programmers may find a slightly less than optimal performance on a specific architecture acceptable.

Acknowledgements

We would like to thank Micheal Garland, John Owens, Chris Rodrigues, Vinod Grover and NVIDIA corporation for their feedback and support. Sam Stone is supported under a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF. We acknowledge the support of the Gigascale Systems Research Center, funded under the Focus Center Research Program, a Semiconductor Research Corporation program. This work was performed with equipment and software donations from Intel.

References

1. NVIDIA: NVIDIA CUDA, `http://www.nvidia.com/cuda`
2. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: NVIDIA Tesla: A unified graphics and computing architecture. IEEE Micro 28(2) (in press, 2008)
3. Woop, S., Schmittler, J., Slusallek, P.: RPU: A programmable ray processing unit for realtime ray tracing. ACM Trans. Graph. 24(3), 434–444 (2005)
4. Intel: Intel 64 and IA-32 Architectures Software Developer's Manual (May 2007)
5. Devices, A.M.: 3DNow! technology manual. Technical Report 21928, Advanced Micro Devices, Sunnyvale, CA (May 1998)
6. Ashcroft, E., Manna, Z.: Transforming 'goto' programs into 'while' programs. In: Proceedings of the International Federation of Information Processing Congress 1971, August 1971, pp. 250–255 (1971)
7. Kennedy, K., Allen, R.: Optimizing Compilers for Modern Architectures: A Dependence-based Approach. Morgan Kaufmann Publishers, San Francisco (2002)
8. Lee, S., Johnson, T., Eigenmann, R.: Cetus - An extensible compiler infrastructure for source-to-source transformation. In: Rauchwerger, L. (ed.) LCPC 2003. LNCS, vol. 2958, Springer, Heidelberg (2004)
9. Ayguadé, E., Blainey, B., Duran, A., Labarta, J., Martínez, F., Martorell, X., Silvera, R.: Is the schedule clause really necessary in OpenMP? In: Proceedings of the International Workshop on OpenMP Applications and Tools, June 2003, pp. 147–159 (2003)
10. Markatos, E.P., LeBlanc, T.J.: Using processor affinity in loop scheduling on shared-memory multiprocessors. In: Proceedings of the 1992 International Conference on Supercomputing, July 1992, pp. 104–113 (1992)
11. Hummel, S.F., Schonberg, E., Flynn, L.E.: Factoring: A practical and robust method for scheduling parallel loops. In: Proceedings of the 1001 International Conference of Supercomputing, June 1991, pp. 610–632 (1991)

12. Bull, J.M.: Feedback guidaed dynamic loop scheduling: Algorithms and experiments. In: European Conference on Parallel Processing, September 1998, pp. 377–382 (1998)
13. Ryoo, S., Rodrigues, C.I., Baghsorkhi, S.S., Stone, S.S., Kirk, D., Hwu, W.W.: Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (February 2008)
14. Ryoo, S., Rodrigues, C.I., Stone, S.S., Baghsorkhi, S.S., Ueng, S.Z., Stratton, J.A., Hwu, W.W.: Program optimization space pruning for a multithreaded GPU. In: Proceedings of the 2008 International Symposium on Code Generation and Optimization (April 2008)
15. Volkov, V., Demmel, J.W.: LU, QR and Cholesky factorizations using vector capabilities of GPUs. Technical Report UCB/EECS-2008-49, EECS Department, University of California, Berkeley, CA (May 2008)
16. Cowie, J.H., Nicol, D.M., Ogielski, A.T.: Modeling the global internet. Computing in Science and Eng. 1(1), 42–50 (1999)
17. OpenMP Architecture Review Board: OpenMP application program interface (May 2005)
18. Intel: Threading Building Blocks, `http://threadingbuildingblocks.org/`
19. Forum, H.P.F.: High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Rice University (May 1993)
20. Liao, S.W., Du, Z., Wu, G., Lueh, G.Y.: Data and computation transformations for Brook streaming applications on multiprocessors. In: Proceedings of the 4th International Symposium on Code Generation and Optimization, March 2006, pp. 196–207 (2006)

Automatic Pre-Fetch and Modulo Scheduling Transformations for the Cell BE Architecture

Nikola Vujić, Marc Gonzàlez, Xavier Martorell, and Eduard Ayguadé

Barcelona Supercomputing Center
Department of Computer Architecture, Technical University of Catalonia
nvujic@bsc.es, marc@ac.upc.edu, xavim@ac.upc.edu,
eduard@ac.upc.edu

Abstract. Ease of programming is one of the main impediments for the broad acceptance of multi-core systems with no hardware support for transparent data transfer between local and global memories. Software cache is a robust approach to provide the user with a transparent view of the memory architecture; but this software approach can suffer from poor performance. In this paper, we propose a hierarchical, hybrid software-cache architecture that targets enabling pre-fetch techniques. Memory accesses are classified at compile time in two classes, high-locality and irregular. Our approach then steers the memory references toward one of two specific cache structures optimized for their respective access pattern. The specific cache structures are optimized to enable high-level compiler optimizations to aggressively unroll loops, reorder cache references, and/or transform surrounding loops so as to practically eliminate the software cache overhead in the innermost loop. The cache design enables automatic pre-fetch and modulo scheduling transformations. Performance evaluation indicates that the optimized software-cache structures combined with the proposed pre-fetch techniques translate into speed-up between 10% and 20%. Evaluation is done on a set of parallel NAS applications.

Keywords: Cell BE Architecture, Software Cache, Pre-fetching, Modulo Scheduling.

1 Introduction

Heterogeneity has become one particular trend in recently proposed computer systems. For instance, the IBM Cell BE processor [1-5] is a multi-core design that mixes two architectures: a traditional superscalar core based on the PowerPC architecture surrounded by eight cores based on the Synergistic Processor Element (SPE)[4]. In the IBM Cell architecture, the SPEs are provided with local memories and data transfers from/to main memory are explicitly performed under software control. In terms of programmability, this adds another level of complexity and programmers have to deal with the burden of explicitly program the necessary data transfers within applications. General compiler-based solutions [5] are difficult to

J.N. Amaral (Ed.): LCPC 2008, LNCS 5335, pp. 31–46, 2008.

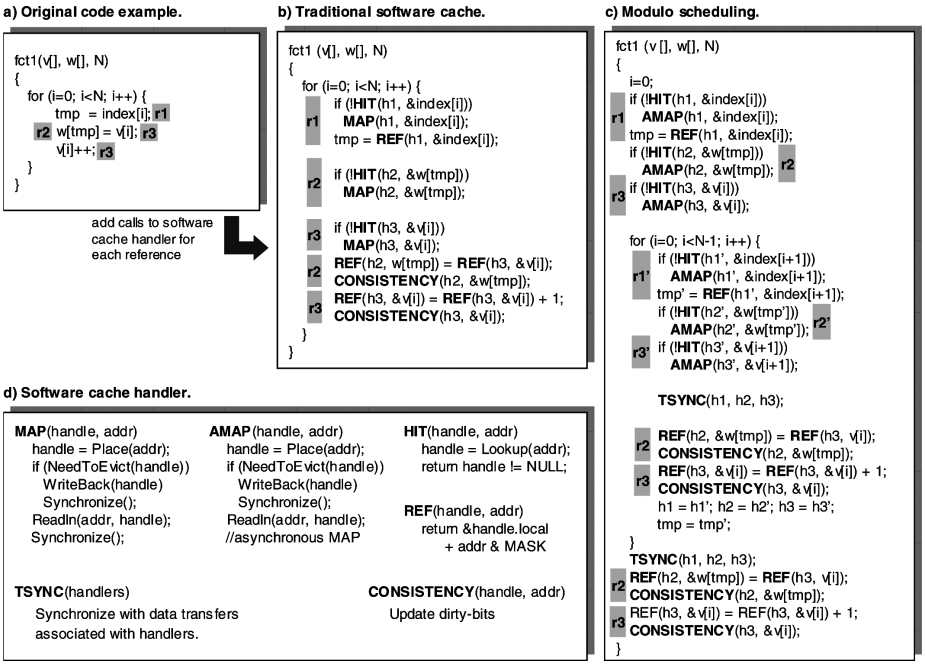


Fig. 1. Overhead of traditional software cache approaches

deploy due to the lack of sufficient information at compile time to generate correct and efficient code.

One global solution is that of emulating a hardware cache by software techniques. In software cache based environments, every memory reference is wrapped by control handlers to ensure correctness. Control handlers are responsible for all cache operations: look-up, placement/replacement, data transfers, synchronization, address translation, and consistency. Figure 1 shows an example of the kind of code emitted by the compiler targeting a software emulated cache.

The memory references *r1*, *r2* and *r3* have been transformed and the correspondent code is showed in Figure 1b. Before the actual use of data, it is necessary to check if the data is resident in the software cache. This checking is done by invoking the HIT runtime call. If data is not resident then miss handler MAP is invoked to serve a miss. The MAP miss handler is responsible for selecting a cache line to be evicted (if necessary, and then perform the write-back operation), and finally loads the requested line in a synchronous manner. When data is resident in the software cache, the actual access can be allowed, but this operation requires an address translation: the REF handler is responsible for that. For memory reference *r3*, it is necessary to update memory consistency structures, in the example this is associated to the CONSISTENCY handler.

Clearly, the transformed code in Figure 1b is far from optimal, especially because of how memory references *r3* and *r1* are treated. Those references expose a high

degree of spatial locality, but every of their instances are going to be checked at runtime introducing unnecessary overhead. For references which expose a high degree of spatial-locality, it is trivial to compute the number of useful data present in the current cache line along the execution of the innermost loop. For such type of memory references we can easily compute the number of loop iterations (within the iteration space of i-loop) for which the current cache line can provide data for such references. This would allow iterating without a miss and without any software cache intervention. But this optimization requires some control over the cache geometry.

First, we must be able to pin a cache line in the cache storage, releasing it only when all high-locality references are done with it. Second, the cache must have at least one unoccupied cache line per distinct high locality references in the loops, if we want to remove all checking code from the innermost loop. Third, it would be desirable to have a "big" cache line size in order to maximize the number of iterations that could be executed with no need of any cache intervention. On the other side, reference *r2* should be treated with very different mechanisms: it exposes very poor spatial locality, so a small cache line would be desirable. This suggests a hybrid design where memory references are mapped to specific storages according to the locality they expose.

Another source of significant overhead is the synchronous communication in the MAP handler. Whatever the implementation of the MAP handler, it is necessary to introduce a synchronization between the data transfer and the actual load/store operation the MAP is associated to. This hinders the possibilities of overlapping communication with computation. Pre-fetch techniques can be introduced to hide the memory latencies, but in the context of software cache systems, pre-fetch does not come for free. Pre-fetching requires execution of control code related to the lookup, placement/replacement and data transfer operations. Besides, it is necessary to ensure that the pre-fetch data is in the range of the valid address space. One well known pre-fetch technique is the modulo scheduling execution [7-9]. In Figure 1c this technique has been applied to the original source code. Basically, data used in iteration *i+1* is pre-fetched in iteration *i*. Now the communications in AMAP are asynchronous, which makes possible to overlap some computation in iteration *i* with some communication related to data used in iteration *i+1*. Notice the TSYNCH call which is responsible that the data required for load/store operations is already in the cache storage. But the problem is not yet solved, since there are two undesirable situations that make the transformation in Figure 1c inapplicable. First, it is necessary to ensure that no conflict appear between the set of consecutive AMAP operations. This is related to the associative level of the cache design and suggests a full associative scheme, always limited by the look-up overheads. Second, it is possible that some write-back operation is triggered along an AMAP operation: this implies some communication that has to be performed synchronously, making useless the modulo scheduling transformation.

Our main contribution is to design a *hierarchical, hybrid software-cache* architecture that is designed from the ground up to enable compiler optimizations that reduce software cache overheads. We identify two main data access patterns, one for high-locality and one for irregular accesses. Because the compiler optimizations targeting these two patterns have different objectives and requirements, we have designed two distinct cache structures that best respond to these distinct access

patterns and optimization requirements. In particular, our design includes: (1) a high-locality cache with a variable configuration, lines that can be pinned, and a sophisticated eager write-back mechanism; and (2) a transaction cache with fast, fully associative lookup, short lines, and an efficient write-through policy. The cache design includes specific support for automatic pre-fetch and modulo scheduling code transformations.

The rest of the paper is organized as follows. Section 2 presents our software cache design. Section 3 describes the code transformations enabled by our approach. Section 4 evaluates our approach using some applications of the NAS benchmarks. Related work is presented in Section 5 and Section 6 concludes the paper with some conclusions.

2 Software Cache Design

We describe in this section the design of our hierarchical, hybrid software cache. Figure 2 shows the high level architecture of our software cache. Memory references exposing a high degree of locality are mapped by the compiler to the *High-Locality Cache*, and the others, irregular accesses are mapped into the *Transactional Cache*. The *Memory Consistency Block* implements the necessary data structures to maintain a relaxed consistency model. The *Pre-fetching Block* implements necessary data structures to maintain pre-fetching for regular memory references.

The cache is accessed through one block only, either the *High-Locality Cache* or the *Transactional Cache*. Both caches are consistent with each other. The hybrid approach is hierarchical in that the *Transactional Cache* is forced to check for the data in the *High-Locality Cache* storage during the lookup process.

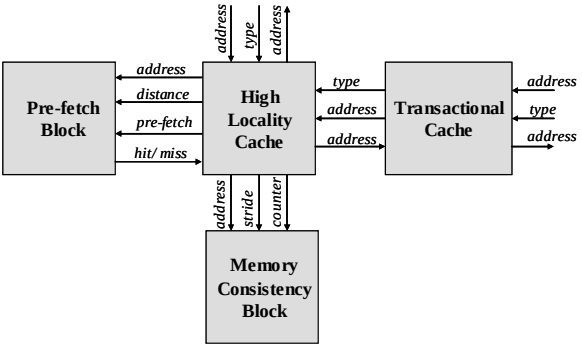


Fig. 2. Block diagram of our software cache design

2.1 The High Locality Cache

The *High-Locality Cache* enables compiler optimizations for memory references that expose a high degree of spatial locality. It is designed to pin cache lines using explicit reference counters, deliver good hit ratios, and maximize the overlap between computation and communication.

2.1.1 High-Locality Cache Structures

The *High Locality Cache* is composed of the following six data structures, depicted in Figure 3: (1) the *Cache Storage* to store application data, (2) the *Cache Line Descriptors* to describe each line in the cache, (3) the *Cache Directory* to retrieve the lines, (4) the *Cache Unused List* to indicate the lines that may be reused, (5) the *Cache Translation Record* to preserve for each reference the address resolved by the cache lookup, and (6) the *Cache Parameters* to record global configuration parameters.

The Cache Storage is a block of data storage organized as N cache lines, where N is total cache storage divided by the line size. The line size is described by the Cache Line Size parameter, and must be a power of 2. In our configuration, we can store between 16 to 128 cache lines of sizes from 512 to 4KB, within its 64KB cache storage.

Each cache line is associated with a unique *Cache Line Descriptor* that describes all there is to know about that line. Its *Global Base Address* is a global memory address that corresponds to the base address associated with this line in global memory. Its *Local Base Address* corresponds to the base address of the cache line in the local-memory cache storage. Its *Cache Line State* records state such as whether the line holds modified data or not. Its *Reference Counter* keeps track of the number of memory references that are currently referencing this cache line. Its *Directory Links* is a pair of pointers used by the cache directory to list all of the line descriptors that map to the same cache directory entry. Its *Free Links* is a pair of pointers used to list all the lines that are currently unused (i.e. with reference counter of zero). Its *Communication Tags* are a pair of integer values used to synchronize data transfers to/from the software cache. In our configuration, we synchronize using DMA fences, using each of the 32 distinct hardware fences. Our communication tags thus range from 0 to 31.

The *Cache Translation Record* preserves information generated by the lookup process and to be later used when data is accessed by the actual reference. It contains 3 elements; the global base address of the original reference, the local base address in the cache storage, and a pointer to the cache line descriptor.

We implement an efficient, fully associative cache structure using the *Cache Directory* structure. It contains a sufficiently large number of double–linked lists (128

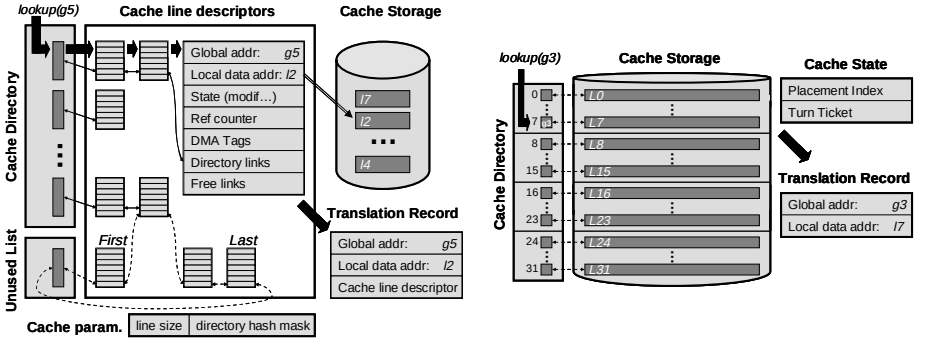


Fig. 3. Structures of the High Locality Cache and Transactional Cache

in our implementation), where each list can contain an arbitrary number of cache line descriptors. A hash function is applied to the global base address to locate its corresponding list, which is then traversed to find a possible match. The use of a hash function enables us to efficiently implement cache configurations with up to 128-way fully associative caches.

The *Cache Unused List* is a double-linked list which contains all the cache line descriptor no longer in use. Other cache parameters include parameters such as the *Cache Directory Hash Mask*, a mask used by the cache directory to associate a global base address with its specific linked list.

2.1.2 High-Locality Cache Operational Model

The operational model for the *High Locality Cache* is composed of all the operations that execute upon the cache structures and implement the primitive operations shown in Figure 1, namely *lookup*, *placement*, *communication*, *synchronization* and *consistency* mechanisms. The following paragraphs describe each type of operation.

The *lookup* operation for a given reference *r*, translation record *h*, and global address *g* is divided in two different phases. The first phase checks if the global address *g* is found in the cache line currently pointed to by the translation record *h*. When this is the case, we have a hit and we are done. Otherwise, we have a situation where the translation record will need to point to a new cache line in the local storage. The lookup process then enters its second phase. The second phase accesses the cache directory to determine if the referenced cache line is already resident in the cache storage. When we have a hit, we update the translation record *h* and we are done. Otherwise, a miss occurs and we continue with placement and communication operations.

The reference counter is often updated during the lookup process. Whenever a translation record stops pointing to a specific cache line descriptor, the reference counter of this descriptor is decremented by one. Similarly, whenever a translation record starts pointing to a new cache line descriptor, the reference counter of this new descriptor is incremented by one.

The placement code is invoked when a new line is required. Free lines are discovered when their descriptor's reference counter reaches zero. Free lines are immediately inserted at the end of the unused cache line list. Modified lines are then eagerly written back to global memory. When a new line is required, we grab the line at the head of the unused cache line list after ensuring that the communication performing the write-back is completed, if the line was modified.

We support a relaxed consistency model. While it is the *Memory Consistency Block* responsibility to maintain consistency, the *High-Locality Cache* is responsible for informing the consistency block of which subsets of any given cache line have been modified and how to trigger the write-back mechanism. Every time a cache line miss occurs, cache thus informs the *Memory Consistency Block* about which elements in the cache line are going to be modified.

The communication code performs all data transfer operations asynchronously. For a system such as the Cell BE processor with a full-featured DMA engine, we reserve the DMA tags *0* to *15* for data transfers from main memory to the local memory, and tags 16 to 31 for data transfers in the reverse direction. In both cases, tags are assigned in a circular manner. Tags used in the communication operations are

recorded in the *Communication Tags* field of the *Cache Line Descriptor*. All data transfers tagged with the same DMA tag are forced by the DMA hardware to strictly perform in the order they were programmed.

The synchronization operation is supported by the data in the *Cache Line Descriptor*, in the *Communication Tags* field. The DMA tags stored in this field are used to check that any pending data transfer is completed. The *Communication Tags* record every tag that invokes the corresponding cache line.

When accessing data, the global to local address translation is supported through the translation record. The translation operation is composed of several arithmetic computations required to compute the reference's offset in the cache line and to add the offset to the local base address.

2.2 The Pre-Fetch Block

The *Pre-Fetch Block* enables automatic pre-fetch for regular memory references. The Pre-fetch Block is selective in the sense that not all regular memory references trigger the pre-fetch. It is activated under demand according to the activity in the High Locality Cache. For selected references, the memory addresses are forwarded to the Pre-fetch Block. Then the pre-fetch can be activated and all forwarded addresses determine the next cache lines to be pre-fetched.

2.2.1 The Pre-Fetch Structures

The *Pre-Fetch Block* is composed of the following four structures: (1) *Pre-Fetch Translation Record* to preserve for each reference the address resolved by the pre-fetch operation, (2) *Pre-fetch Translation Table* to keep track of records being used in pre-fetch operation, and (3) the *Pre-fetch Communication Tags* to preserve DMA tags used for pre-fetching.

The *Pre-Fetch Translation Record* structure consists of four fields: (1) the pre-fetch global address is the base address of the cache line that triggers pre-fetch, (2) the pre-fetch local address is the base address of the cache line allocated to hold the pre-fetched data in the local store, (3) the pre-fetch cache line descriptor is a pointer to the cache line descriptor of the pre-fetched line, and (4) the pre-fetch distance that indicates the next cache line to be pre-fetched as a distance (in a number of cache lines) from the cache line base address that triggered the pre-fetch.

The *Pre-Fetch Translation Table* is a table where each entry holds one *Pre-Fetch Translation Record*. The Pre-fetch Counter keeps track of the number of pending pre-fetch operations.

The *Pre-fetch Communication Tags* consists of all communication tags actively used for pre-fetching purposes. These tags are going to be used to synchronize the data transfers associated to the pre-fetched data.

2.2.2 The Pre-Fetch Block Operational Model

Memory references that have been selected to trigger the pre-fetch are recorded in the Pre-fetch Translation Table. Pre-fetch is activated from the High Locality Cache and this causes the Pre-fetch Block to traverse the Pre-fetch Translation Table and for every non empty entry performs the look-up, placement and replacement operations as if the cache line being pre-fetched was referenced by the actual computation.

Along this process all the communication tags used in the data transfers are recorded in the Pre-fetch Communication Tags register. Under control of the High Locality Cache, it is possible to synchronize with the pre-fetched data using this register.

Introducing pre-fetch support requires reserving some of the available communication tags specifically for that purpose. The range of tags that was used to bring data in to the cache storage is split in two different ranges, one from 0 to 7, the other from 8 to 15. Both ranges are assigned in a circular manner and the High Locality Cache and the Pre-fetch block are coordinated to switch from one range to the other every time the Pre-fetch block is required to perform pre-fetch operations.

2.3 The Transactional Cache

The *Transactional Cache* is aimed at memory references that do not expose any spatial locality. Because miss ratios are expected to be high, this cache is designed to deliver very low hit and miss overhead while enabling overlapped computation and communication. The design introduces very simple structures that allow support for *lookup*, *placement*, *communication*, *consistency*, *synchronization*, and *translation* mechanisms.

In our configuration, the transactional storage is organized as a small 4KB capacity cache, fully associative, and with *32 128*-bytes cache lines. It supports a relaxed consistency model using a write-through policy.

2.3.1 The Transactional Cache Structures

The *Transactional Cache* is composed of the following four data structures, shown in Figure 3: (1) the *Cache Directory* to retrieve the lines, (2) the *Cache Storage* to hold the application data, (3) the *Translation Record* to preserve the outcome of a cache lookup for each reference, and (4) some additional cache state.

The *Cache Directory* is organized as a vector of 32 4-byte entries. Each entry holds the global base address associated with this entry's cache line. The index of the entry in the directory structure is also used as index into the *Cache Storage* to find the data associated with that entry. The directory entries are packed in memory and aligned at a 16-byte boundary so as to enable the use of fast SIMD compares to more quickly locate entries. The *Cache Storage* is organized as 32 cache lines, where each 128-bytes line is aligned at a 128-byte boundary.

To increase concurrency, the cache directory and storage structures are logically divided in four equal-size partitions; the *Cache Turn Ticket* indicates which partition is actively used. Within the active partition, the *Cache Placement Index* points to the cache line that will be used to service the next miss.

At a high level, the active partition is used for buffering cache lines which are going to be used in the current transaction and these cache lines were pre-fetched. The next partition, in circular manner, is used for placing cache lines which we are pre-fetching and which are going to be used in the next transaction in the next iteration of the unrolled loop. Other two partitions are used to buffer data of the two previous transactions while their modified data is being flushed back to the main memory. Based on this explanation, we defined three states in which our partitions can be: *in-use*, *pre-fetching* and *flushing*.

2.3.2 The Transactional Cache Operational Model

In this paper, a transaction is a set of computation and related communication that will happen as a unit (but never rollback). Operations in a transaction happen in four consecutive steps: (1) initialization, (2) communication into local memory, (3) computation associated with the transaction, and (4) propagation of any modified state back to global memory.

During initialization, in Step 1, the *Cache Turn Ticket* is set to point to the next partition in the circular manner. The *Cache Placement Index* is set to point to the first cache line of the new partition. In our configuration, its value can be 0, 8, 16 or 24 when the ticket is, respectively, pointing to partition 0, 1, 2, or 3. In addition, all cache directory entries in the new active partition are erased.

In Step 2, the data corresponding to each global-memory reference is brought into the local memory, using sequences of look-up and possibly calls to the miss-handler. The lookup process for a given reference *r*, translation record *h*, and global address *g* first proceeds with a standard *High-Locality* cache lookup, since we do not want to replicate data in both cache structures. This first lookup can be avoided if address *g* can be guaranteed not to be found in the high-locality cache. When a hit occurs, the *Local Base Address* field in translation record *h* is simply set to point to the appropriate sub-section of the line in the high-locality cache storage. When a miss occurs, however, we proceed by checking the address *g* against the entries in transactional cache directory. This lookup is fast on architectures with SIMD units, such as the SPEs. On platforms where 4 entries fit into one SIMD register, such as the SPEs, we perform a 32-way address match using 8 compare SIMD instructions. When a miss occurs, a placement operation is executed. When a hit occurs, the look-up can operate in one of two ways. If the hit occurred within the active partition (partition where we are going to pre-fetch the data for next iteration), we simply update the translation record *h*. If the hit occurred within the next partition, in circular manner, then we need to do two actions. First, we need to migrate the line to the active partition, a placement operation is used for this operation as well. Second, we need to inform previous partition (partition which is in *in-use* state) about migrated cache line in order to maintain consistency between transactions. If, however hit occurred within the other partitions, we simply update the translation record *h*.

The placement code simply installs a new directory entry and associated cache line data at the line pointed by the *Cache Placement Index*. The placement index is then increased by one (modulo 32). Communications generated by the miss in Step 2 results into an asynchronous 128-byte transfer into local memory.

Step 3 proceeds with the computation, using the same translation record as seen in Section 2.1.

In Step 4, every modified storage location that was modified by a store in Step 3 is directly propagated into global memory. This approach to relaxed consistency eliminates the need for any extra data structures (such as dirty bits) and do not introduce any transfer atomicity issue. These asynchronous communications occur regardless of whether a hit or miss occurred in Step 2. Moreover, only the modified bytes of data are transferred into global memory during Step 4.

In order to ensure consistency within and across transactions, every data transfer is tagged with the index of the cache line being used (from *0* to *31*), and a fence is placed right after the data transfer operation. All data transfers tagged with the same

tag are forced by the hardware to perform strictly in the order under which they were programmed. The synchronization code occurs in precisely two places. The first synchronization is placed between Steps 2 & 3, to ensure that the data arrive before being used. When Partition 0 is active, we wait for data transfer operations with tags [*0..7*], for partition 1 appropriate tags are [*8..15*], for partition 2 tags are [*16..23*] and for partition 3 wait for tags [*24..31*]. For the data transfer initiated in Step 4, the synchronization code is placed at the beginning of the next transaction with the same value for the *Cache Turn Ticket*, synchronizing with the data transfer operations tagged with numbers [*0..7*], [*8..15*], [*16..23*] or [*24..31*].

2.4 The Memory Consistency Block

The *Memory Consistency Block* contains the necessary data structures to maintain a relaxed consistency model. For every cache line in the *High Locality Cache*, information about what data has been modified is maintained using a Dirty Bits data structure. Whenever a cache line has to be evicted, the write-back process is composed by three steps. The cache line in main memory is read, then a merge operation is applied between both versions, the one resident in the cache storage and the one recently transferred, and finally, the output of the merge is sent back to main memory. All data transfers are synchronous and atomic.

3 Code Transformations

We describe in this section the type of code transformation techniques that are now enabled using our pre-fetching and modulo scheduling approach in the software cache. With no loss of generality, the code transformation targets the execution of loops.

The code transformations are performed in three ordered phases: (1) classifying of memory references into regular and irregular accesses; (2) transformation of the code to optimize regular memory accesses, and (3) transformation of the code to optimize irregular memory accesses. We illustrate this process in Figure 4 using the same introductory example presented in Figure 1a.

3.1 Classification of Memory Accesses

In Phase 1, memory accesses are classified as regular or irregular accesses. Figure 4a shows the classification of the references for our exemplary code. Memory accesses *index[i]* and *v[i]* with *i=0…N* are labeled as regular, while memory access *w[tmp]* with *tmp=index[i]* is labeled as irregular memory access.

3.2 Regular Access Transformations

In phase 2, original *for*-loop is transformed into two nested loops (Figure 4b). Dynamic sub-chunking of the iteration space is done by using those two nested loops. In each dynamic sub-chunk of iterations we are sure that all relevant data are permanent in the cache storage and iterating through them, in the inner *for*-loop of the transformed code, is not going to produce miss. Work done in the inner *for*-loop

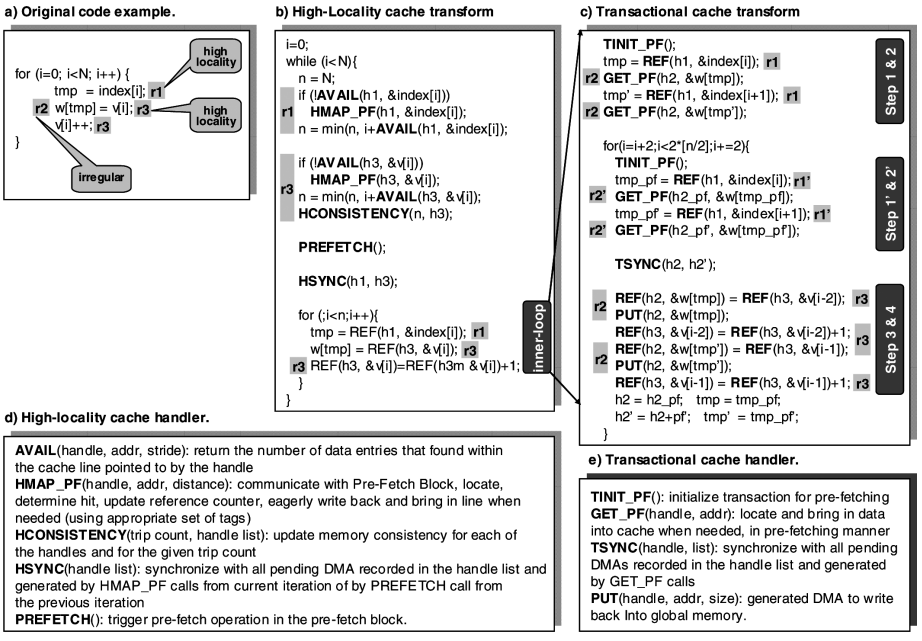


Fig. 4. Example of C code and its code transformation

(related to regular memory accesses) does not have any cache overhead. In the while loop we are introducing necessary code transformations per each high locality memory reference. The lookup, dynamic sub-chunking, consistency maintaining, pre-fetching and synchronization operation are done in the while loop.

The lookup operation checks if the address *&index[i]* of the reference *r1* is in the cache line pointed to by the translation record (handle) *h1*. This checking is done by using AVAIL macro. The AVAIL macro returns for reference *r1* number of iterations for which this reference will be present in the cache line pointed to by handle *h1*. If this number is greater than zero we have hit and then we are proceeding with determining of the upcoming dynamic sub-chunk of the iteration space. If this number is equal to zero then macro HMAP_PF is invoked to serve a miss. Notice the third argument of the HMAP_PF macro, indicating if pre-fetch has to be considered for the given memory reference. This argument corresponds to the pre-fetch distance, indicating the next cache line to be most likely accessed by the memory reference. In case the distance is other than zero, pre-fetch is activated and the address is forwarded to the Pre-fetch Block. Next step is determining of the upcoming dynamic sub-chunk of the iteration space. Once we have sub-chunking factor *n* we can process with consistency and synchronization operations. Since reference *r1* is read only access reference then consistency operation is not processed for *r1* but is processed for *r3* which is read and write access reference. The PREFETCH macro triggers pre-fetch for all forwarded addresses. Notice that the pre-fetch code is executed before the synchronization with the DMA engine takes place, giving the opportunity to overlap the pre-fetch code to actual communication.

3.3 Irregular Access Transformation

In Phase 3, we transform the inner *for*-loop so as to optimize cache overhead for irregular memory accesses. The first task is to determine the transactions. In our work, the scope of a transaction is a basic block, or a subset of. Large transactions are beneficial as they potentially increase the number of concurrent misses, thus increasing communication overlap. In general, a transaction can contain as many distinct irregular accesses as there are entries in a single partition of the transactional cache, 8 in our configuration. Because of our focus on loops, larger transactions are mainly achieved through loop unrolling. In our example, we unrolled the inner *for*-loop by a factor of 2 (for conciseness) so as to include two *w[tmp]* and *w[tmp']* references within a single transaction.

The code generated for a transaction closely follows the four step process outlined in Section 2.2.2. As shown in Figure 4c, we first initialize the transaction using the macro TINIT (Step 1) and then proceed in asynchronously acquiring the data of each irregular reference *r2* (due to loop unrolling of factor 2 we have two *r2* references) using the GET macro (Step 2). Once all irregular references have been processed, we issue a TSYNC operation to synchronize with pending DMAs issued by appropriate GET macros. We then access the data using the REF macro (Step 3) and write-back the modified data using the PUT macro (Step 4).

Conceptually, the work inside transactions in modulo scheduled loop can be visualized as four tasks. In the loop prologue we pre-fetch data which are going to be used within computation section in the first iteration of the unrolled loop. We assign task *Step1&2* to this prologue. At the beginning of the unrolled loop body we pre-fetch data which is going to be used in the next iteration or in the loop epilogue. In this part of the code we use translation records *h2* and *h2'*. We assign task *Step1'&2'* to this part of the unrolled loop body. After this we have a necessary synchronization point where we synchronize with pending DMAs determined by translation records *h2* and *h2'*. When we are sure that data has arrived in the cache, we execute computation section and at the end, modified data is sent back to main memory (PUT macro). This corresponds to task *Step3&4*. In the steady state of the loop, partitions

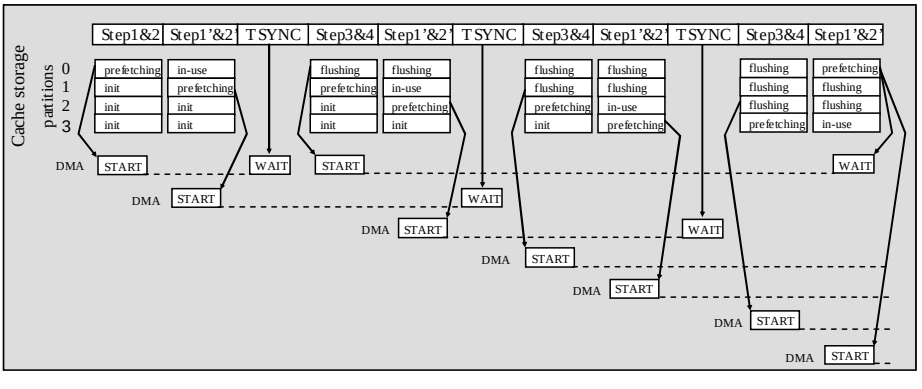


Fig. 5. Sequence of events in a modulo scheduled loop

go changing of state: pre-fetch, in-use, flushing. Note that for conciseness, the loop unrolling has been done assuming that the number of iteration was a multiple of two. Figure 5 shows the evolution of each partition for three iterations of the loop.

4 Evaluation

In the evaluation section we measure the impact in performance of the proposed pre-fetching techniques: automatic pre-fetch for regular references, modulo scheduling for irregular references. In this evaluation we never combine these two techniques in the same loop. We compare two cache configurations, one where pre-fetch is enabled, another where pre-fetch is not active. Improvement is measured in terms of speed-up.

We have evaluated the proposals with the CG, IS and FT parallel applications from the NAS benchmark suite [10] and STREAM parallel application [6], which are parallelized using OpenMP directives. All measurements are performed on a Cell BE blade with two Cell BE processors running at 3.2 GHz with 1 GB of memory (512 MB per processor) under Linux Fedora Core 6 (Kernel 2.6.20-CBE). Only one Cell BE processor is used for the evaluation.

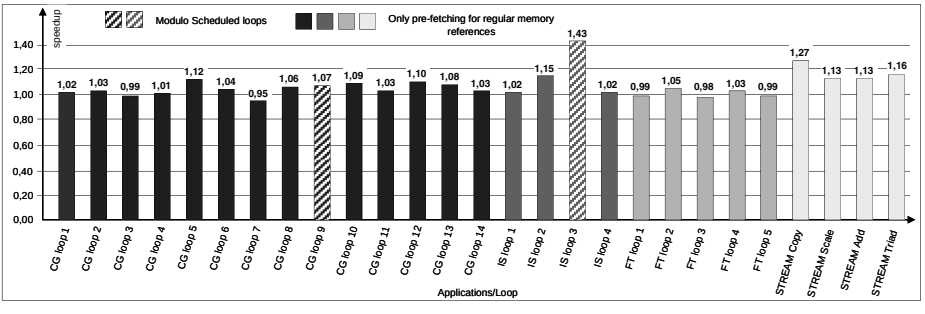


Fig. 6. Speed-up factors for automatic pre-fetch and modulo scheduling. Modulo scheduling is used in CG loop9 and IS loop3, due to CG and IS are totally dominated by irregular memory accesses in the mentioned loops.

Figure 6 shows speedup factors obtained by enabling pre-fetch in a variety of loops from the CG-B, IS-B, FT-B and STREAM benchmarks. Overall execution times for CG-B, IS-B and FT-B are shown in Figure 7. For STREAM, the improvements are noticeable, but very predictable in the sense that the four tested kernels are not computationally bounded. Communications represent an important percentage of overall execution time. This is yet more noticeable in the differences we observe between the four kernels: the *copy* kernel which is not including any floating point operation doubles the performance of the other kernels.

In the case of CG-B, the improvements range from 3% up to 10 % at most. Loops 3 and 7 suffer from slight degradation (not even a 1% and 5% respectively). The reason for that is related to the differences on how deeply the loops are affected by communications. The CG-B loop 9 is dominated with irregular memory references and is the most consuming loop in the CG-B. Improvement achieved in this loop has

good influence on overall execution time of the CG-B (Figure 7). The case of the IS-B is different. Here the benefits are quite impressive: loop 2 improves about 15% and loop 3 improves about 40%. Loop 3 is totally dominated by irregular memory references and the introduction of the modulo scheduling transformation is what causes such improvement. Improvement in loop3 has good influence on overall execution time of the IS-B. The case of FT is very different and exposes very poor improvements, ranging from slight performance degradation (2% at most) up to some improvement close 5%. All loops are dominated by the computation, not by the communication overheads. There is no improvement in overall execution time for FT.

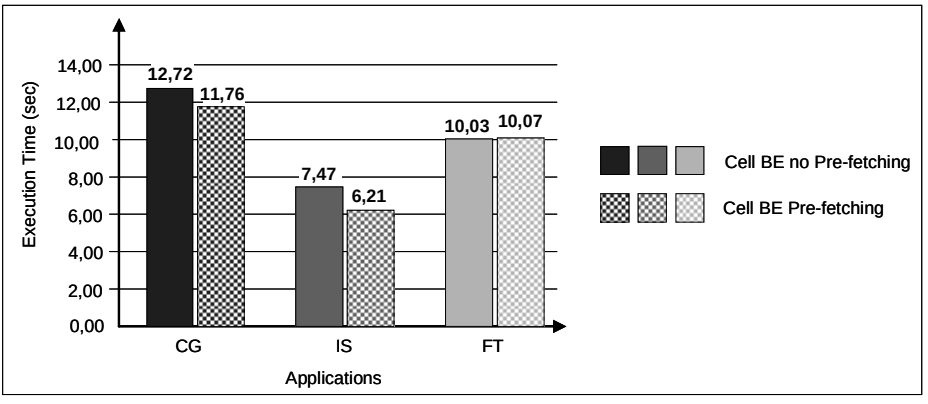


Fig. 7. Execution times of NAS benchmarks. Corresponding speedup factors in overall execution times are: CG - 1,082, IS - 1,203 and FT - 0,996.

5 Related Work

Although a different technique, tiling transformations and static buffers may be used to reach the same level of code optimization [5]. In general, 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. This approach, however, requires precise information about memory aliasing at compile time, which is not always available. In general, the association between static buffers and memory references should be postponed until run time. This is what we do in this paper, since cache lines are treated as buffers that are dynamically allocated, solving all the difficulties related to memory aliasing. Of course, if the performance of a software cache approach is to match that of static buffers, clearly, any efficient implementation should work with a cache line size similar to that of the static buffers (usually 1KB, 2KB, 4KB, depending on the number of memory references to treat) [5]. This is the case of the software cache design presented in this paper.

Specifically for the Cell BE, there has been proposal to perform data pre-fetching under an inspector/executor model [12]. For indirect accesses, a slicing compilation technique is introduced to generate a code version that at runtime computes all memory addresses generated in indirect accesses. This makes possible to overlap DMA transfers with the slice execution. This approach has been showed to return

considerable improvements for indirect accesses, but the technique is limited to the associative level in the cache design. Cache conflicts cause to switch between the inspector and executor code versions, diminishing the effects of this technique.

The Memory Hierarchical Layer Assignment (MHLA) [13] is a unified technique which addresses the problem of optimizing the data assignments into memory layers and the block transfers between memory layers. This technique starts from the source code specification of the application and by collecting profiling information optimizes memory mapping and execution order of data transfers. Also, memory organization is potentially customized by this technique. The similarity of this technique with our approach is that pre-fetching is implemented by invoking DMA operations in order to overlap computation and communication. In contrast to our technique, MHLA is aimed for buffering techniques and simple memory organizations due to application-specific pre-fetching approach.

Hare [14] is a pre-fetching scheme consisting of a programmable engine controlled by the user instructions. This technique uses hardware support for pre-fetching. Indeed, pre-fetching is initiated by the hardware at run-time. Programming the proposed engine by user code takes advantages from compile-time analyzes and hardware eliminates additional pre-fetch instruction overhead. In contrast with this proposal, in our work we do not have any hardware support for pre-fetching.

Interrupt Triggered Software Pre-fetching (ITSP) [15] is a pre-fetching technique for real-time embedded systems that adds pre-fetching instructions in interrupt handler software to target cache misses. Pre-fetching optimizations done in ITSP tunes the software to be executed based on observed performance during previous executions. In contrast with our work, ITSP relies on profiling information collected during previous executions of application and hardware pre-fetching instructions are used.

6 Conclusions

This paper presents a novel hybrid software cache architecture designed for pre-fetching purposes. Hybrid software cache architecture maps memory accesses according to the locality they are exposing. According to difference in mapping, pre-fetching is organized in order to enable good overlap of communication and computation for both types of memory accesses. We show performances of pre-fetching for regular and irregular memory accesses. We also show impact of additional instruction overhead introduced due to software implemented pre-fetching. We show that with our approach good speedup can be obtained in some benchmarks (speedup factors from 1.15 to 1.43) and also we show that additional instruction overhead in software implemented pre-fetching sometimes has negative impact on overall performances of some applications and some particular loops in the applications.

Acknowledgments

This research has been supported by the IBM MareIncognito project, in the context of the research projects between BSC and IBM, and by the Ministry of Education of Spain (CICYT) under contract TIN2007-60625 and by the HIPEAC European Network of Excellence under the contract IST-004408.

References

1. Peter Hofstee, H.: Power Efficient Processor Architecture and The Cell Processor. In: Proceedings of the 11th Int'l. Symposium on High-Performance Computer Architecture (2005)
2. Pham, D., et al.: The Design and Implementation of a First-Generation Cell Processor. In: Proceedings the IEEE International Solid-State Circuits Conference (2005)
3. Kistler, M., et al.: Cell Multiprocessor Communication Network: Built for Speed. IEEE Micro 26(3), 10–23 (2006)
4. Gschwind, M., et al.: A Novel SIMD Architecture for the Cell Heterogeneous Chip-Multiprocessor. In: Hot Chips, vol. 17 (2005)
5. Eichenberger, A.E., et al.: Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture. IBM Systems Journal 45(1) (2006)
6. McCalpin, John, D.: Memory Bandwidth and Machine Balance in Current High Performance Computers. IEEE Computer Society Technical Committee on Computer Architecture (TCCA) (1995)
7. Ramakrishna Rau, B., et al.: Code Generation Schema for Modulo Scheduling Loops. In: Proccedings of the 25th Annual International Symposium on Microarchitecture (1992)
8. Ramakrishna Rau, B., et al.: Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. In: Proceedings of the 27th annual International Symposium on Microarchitecture (1994)
9. Lavery, D.M.: Modulo Scheduling of Loops in Control-intensive Non-numeric Programs. In: Proceedings of the 29th annual ACM/IEEE International Symposium on Microarchitecture (1996)
10. Bailey, D., et al.: The NAS parallel benchmarks. Technical Report TR RNR-91-002, NASA Ames (August 1991)
11. Sinharoy, B., et al.: POWER 5 system micro-architecture. IBM Journal of Research and Development 49(4/5) (July/September 2005)
12. Chen, T., et al.: Prefetching irregular references for software cache on cell. In: Proceedings of the sixth annual IEEE/ACM international symposium on Code Generation and Optimization, pp. 155–164 (2008)
13. Dasygenis, M., et al.: A Combined DMA and Application-Specific Prefetching Approach for Tackling the Memory Bottleneck. IEEE Transactions on Very Large Integration (VLSI) Systems 14(3), 279–291 (2006)
14. Chen, T.-F.: An Effective Programmable Prefetch Engine for On-Chip Caches. In: Proceedings of the 28th Annual International Symposium on Microarchitecture (1995)
15. Batcher, K.W., et al.: Interrupt Triggered Software Prefetching for Embedded CPU Instruction Cache. In: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (2006)

Efficient Set Sharing Using ZBDDs

Mario Méndez-Lojo[1], Ondřej Lhoták[2], and Manuel V. Hermenegildo[1,3]

[1] Dept. of Computer Science, University of New Mexico, USA
[2] D. R. Cheriton School of Computer Science, University of Waterloo, Canada
[3] Dept. of Computer Science, Tech. U. of Madrid, Spain and IMDEA-Software

Abstract. Set sharing is an abstract domain in which each concrete object is represented by the set of local variables from which it might be reachable. It is a useful abstraction to detect parallelism opportunities, since it contains definite information about which variables do not share in memory, i.e., about when the memory regions reachable from those variables are disjoint. Set sharing is a more precise alternative to pair sharing, in which each domain element is a set of all pairs of local variables from which a common object may be reachable. However, the exponential complexity of some set sharing operations has limited its wider application. This work introduces an efficient implementation of the set sharing domain using Zero-supressed Binary Decision Diagrams (ZBDDs). Because ZBDDs were designed to represent sets of combinations (i.e., sets of sets), they naturally represent elements of the set sharing domain. We show how to synthesize the operations needed in the set sharing transfer functions from basic ZBDD operations. For some of the operations, we devise custom ZBDD algorithms that perform better in practice. We also compare our implementation of the abstract domain with an efficient, compact, bitset-based alternative, and show that the ZBDD version scales better in terms of both memory usage and running time.

1 Introduction

Set sharing [11] is an abstract domain aimed at tracking dependency information among sets of variables. In set sharing abstractions, each concrete object is represented by the set of program variables from which it might be reachable. Set sharing-based analyses discover valuable information for parallelizing instructions, statements, function calls, etc. (and are therefore typically used for that purpose), since each abstract state contains definite information about which variables do not share, i.e., which variables cannot reach the same memory location. From this perspective, set sharing analysis can be seen as a compact encoding of the information present in points-to analyses, but in set sharing only the groups of variables that might reach the same object in memory are stored.

Set sharing has been shown to be a more precise alternative to, e.g., *pair* sharing, in which each domain element is a set of all pairs of local variables from which a common object may be reachable. However, some of the intrinsic operations of the set sharing domain are exponential in the number of local variables being tracked, which can become a problem for certain programs and has limited so far wider application. This intrinsic complexity can be dealt with in part by introducing widenings, i.e., simplifying the sharing sets conservatively when they become too large, but of course at the

J.N. Amaral (Ed.): LCPC 2008, LNCS 5335, pp. 47–63, 2008.

expense losing precision. Finding significantly more efficient implementations reduces the need for resorting to such lossy solutions and consequently improves practicality.

We introduce a new, efficient implementation of the set sharing domain using Zero-supressed Binary Decision Diagrams (ZBDDs). ZBDDs were designed to represent sets of combinations (i.e., sets of sets), so they can represent very naturally the elements of the set sharing domain. To the best of our knowledge this is the first link provided between set sharing and ZBDDs. We start by providing set-sharing transfer functions for a subset of Java.[1] We then show how to express the operations needed for implementing the set sharing transfer functions in terms of basic ZBDD operations. Also, for some of the operations, we propose custom ZBDD algorithms that are more appropriate for these particular cases than those in the standard ZBDD libraries. In particular we provide a design for native ZBDD operations that emulate non-standard set manipulations. The introduction of ZBDDs is done at the implementation level and does not alter the definition of the domain operations, so that the domain designer does not need to be aware of their presence. Finally, we provide performance results comparing two implementations of the set-sharing domain: an efficient, compact, bitset-based alternative (representing a highly-tuned version of the traditional approach) and our ZBDD-based implementation. The results show that the ZBDD version scales better in terms of both memory usage and running time. Our custom ZBDD algorithms are also shown to perform better in practice than the stock ones.

2 Reachability and Sharing

As mentioned before, we will concentrate for concreteness on a subset of Java, although set sharing has been shown to be applicable to different classes of imperative and declarative languages. A concrete state $G = (Var \cup Obj, E)$ is a directed graph where every node can be either a variable $v \in Var$ or an object $o \in Obj$. The edges of the graph have been labeled such that $o_1 \xrightarrow{\mathtt{f}} o_2$ means "the field `f` of object o_1 points to o_2." We will assume that edges connecting variables and objects have the special label `-`. An object o is reachable from the variable v in G iff there is a path $v \xrightarrow{\mathtt{-}} o_1 \xrightarrow{\mathtt{f}} o_1 \xrightarrow{\mathtt{g}} o_2 \ldots \xrightarrow{\mathtt{h}} o$. The reachability set of a variable v in the state G is the set of all objects that are reachable from it, i.e., $reach(G, v) = \{\, o \in Obj \mid \; o \text{ is reachable from } v \text{ in } G\}$.

One or more variables *share* in a state G if the intersection of their reachabilty sets is non-empty:

$$share(G, V) \Leftrightarrow \bigcap_{v \in V} reach(G, v) \neq \emptyset.$$

Since null variables have no outgoing edges (conversely, if o`.f` is null, there is no edge in the graph that starts at o and is labeled with `f`), they do not share.

Given graph G, define its set sharing as the set of maximal sets of variables that share:

$$sh(G) = \{\, V' \subseteq Var \mid \; share(G, V') \text{ and } \nexists W \text{ s.t. } V' \subset W \text{ and } share(G, W)\}$$

[1] As we will see later, these transfer functions, which are independent from the specific way in which the internal set-sharing domain operations are implemented, are in fact themselves improvements over those previously proposed.

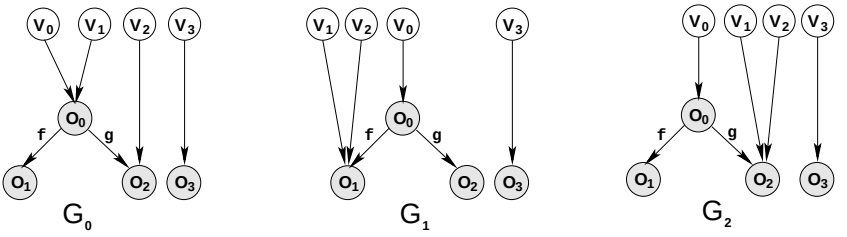


Fig. 1. Three concrete states

The set sharing provides definite information about which variables do not have any memory location in common, i.e., the memory regions reachable from them are disjoint. We can be sure that no object is reachable from more than one variable of a set W if no superset of W is an element of $sh(G)$.

Example 1. Fig. 1 shows three examples of concrete states. We assume that all the variables are of type `Foo`, a class with two fields `f` and `g`, pointing to objects of class `Foo`. In the graph G_0, the reachability sets are $reach(G_0, v_0) = reach(G_0, v_1) = \{o_0, o_1, o_2\}$, $reach(G_0, v_2) = \{o_2\}$ and $reach(G_0, v_3) = \{o_3\}$. The set sharing of G_0 is $sh(G_0) = \{\{v_0, v_1, v_2\}, \{v_3\}\}$. Note that $sh(G_0) = \{\{v_0, v_1\}, \{v_0, v_1, v_2\}, \{v_3\}\}$ is not an acceptable set sharing, even though v_0 shares with v_1, because $\{v_0, v_1\} \subset \{v_0, v_1, v_2\}$, and v_0, v_1, and v_2 all share. The reachability sets of v_1 and v_2 in G_1 and G_2 differ from the ones in G_0; however, the set sharing is the same for all three graphs: $sh(G_0) = sh(G_1) = sh(G_2) = \{\{v_0, v_1, v_2\}, \{v_3\}\}$.

Note that the information provided by set sharing abstract states at program points is instrumental for parallelization: assume that the set sharing of the example, $\{\{v_0, v_1, v_2\}, \{v_3\}\}$, is in fact the abstract state inferred by analysis at the program point just before two consecutive method calls $m(v_0, v_1, v_2)$ and $n(v_3)$. The set sharing represents a number of concrete states (including G_0, G_1, and G_2) in all of which v_3 points to a memory region that is disjoint from the memory regions pointed to by v_0, v_1, or v_2. Since analysis is safe, while actual sharing during execution may be less, there cannot be any concrete states in which there is more sharing than that implied by $\{\{v_0, v_1, v_2\}, \{v_3\}\}$. Thus, under reasonable assumptions regarding the parallel abstract machine, memory management, scheduling, etc., the two method calls can be safely parallelized since they are independent: execution of $m(v_0, v_1, v_2)$ cannot affect that of $n(v_3)$ and they can proceed in parallel without interference. Also, the final state after executing them in parallel will be equivalent to the state obtained after their sequential execution.

3 Sharing Semantics as Set Operations

3.1 Notation

We use double capital letters (like SH) for sets of sets, single capital letters (S) for sets and lowercase letters (for instance, v) to denote elements of a set. We write $SH_V =$

$\{ S \in SH \mid V \subseteq S\}$ to denote the subset of SH containing all sets having V as a subset. Conversely, $SH_{-V} = SH - SH_V$. For singleton sets, we define a more concise notation: $SH_v = SH_{\{v\}}$ and $SH_{-v} = SH_{-\{v\}}$.

We define *projecting out* v from SH as removing v from every set in SH: $SH|_{-v} = \{ S \setminus \{v\} \mid S \in SH\} \setminus \{\{\}\}$. The *replacement operator* on sets of sets replaces all the ocurrences of variable v_1 with v_2 in every set. Formally, $SH|_{v_1}^{v_2} = \left\{ S|_{v_1}^{v_2} \;\middle|\; S \in SH \right\}$, where

$$S|_{v_1}^{v_2} = \begin{cases} S \text{ if } v_1 \notin S \\ S \setminus \{v_1\} \cup \{v_2\} \text{ else} \end{cases}$$

The *binary union operator* $\uplus$ computes the unions of all pairs of sets taken from two sets of sets: $SH_1 \uplus SH_2 = \{ S_1 \cup S_2 \mid S_1 \in SH_1, S_2 \in SH_2\}$.

3.2 Abstract Operations

In this section, we review the abstract set sharing semantics that was defined and proven correct in previous work [16]. We also improve the precision for two of the operations: the field load and the field store. Our compositional semantics defines a denotation function for each expression and command. We define the special variable res, which stores the result of an expression. Thus, the functions for both expressions and commands are transformers on set sharings. The function for an expression transforms the set sharing to abstract a state in which res points to the result of evaluating the expression.

Figs. 2 and 5 contain the semantics of expressions and commands, respectively. They represent the transition from an initial *abstract state* [6] SH to a final abstract state SH'. In our domain, an abstract state SH approximates all the set sharings of a set of concrete states GG: $SH = \alpha(GG) = \bigcup_{G \in GG} sh(G)$, i.e., SH is a correct abstraction of a set of concrete states $\{G_1, \ldots, G_n\}$ if $sh(G_i) \subseteq SH, i = 1..n$. For instance, given a concrete state G such that $sh(G) = \{\{v_3\}\}$, the abstract state $\{\{v_0, v_1, v_2\}, \{v_3\}\}$ is a valid approximation of G. If a variable is null in the concrete states $\{G_1, \ldots, G_n\}$, it does not appear in SH. Thus, the predicate **mustBeNull**(SH, v) returns true when $SH_v = \emptyset$.

In practice, our abstract state is a pair composed of an abstract set sharing and a type component τ. The objective of this second element is to approximate the set of possible types of each variable. This corresponds to the concept of a "type of class" analysis [1,7]. In our context, τ helps in determining which variables are non null and which

$\mathcal{SE}^I_\pi[\![\texttt{null}]\!](SH)$
$SH' = SH$

$\mathcal{SE}^I_\pi[\![\texttt{new } k]\!](SH)$
$SH' = SH \cup \{\{res\}\}$

$\mathcal{SE}^I_\pi[\![v]\!](SH)$
$SH' = (\{\{res\}\} \uplus SH_v) \cup SH_{-v}$

$\mathcal{SE}^I_\pi[\![v\,.\,\texttt{f}]\!](SH)$
$SH' = \begin{cases} \bot \text{ if } \textbf{mustBeNull}(SH, v) \\ SH \cup (\{\{v, res\}\} \uplus \bigcup_{S \in SH_v} \mathcal{P}(S|_{-v})) \text{ else} \end{cases}$

Fig. 2. Abstract semantics for the expressions as set operations

ones may be null. If we consider $null$ as another type [13], then a variable may be null if $null$ is one of its possible types: **mayBeNull**$(\tau, v) = (\overline{\textbf{mustBeNull}}(SH, v)$ and null $\in \tau(v))$. For clarity, we omitted the type component from the transfer functions in Fig. 2 and 5; the full version of the semantics can be found in the Appendix in Fig. 13 and 14.

3.3 Semantics of Expressions

Null, New and Variable Load: The `null` expression loads the null constant into the special variable res, so it has no effect on the abstract state, since res does not point to any object, and therefore does not share with any variable (including itself), both before and after evaluating the expression. The `new` expression adds the singleton $\{res\}$ to the current set sharing, since it creates a fresh object that cannot be reached from any of the existing variables. A variable load `v` forces res to be an alias of v, and therefore res shares with all those variables with which v shares. Sharings in SH_{-v} remain unaffected, since the addition of res cannot change the reachability set of any variable not reachable from v. For instance, given $SH = \{\{v_0, v_1, v_2\}, \{v_3\}\}$, the variable load v_0 results in $SH' = SH_{-v} \cup (\{\{res\}\} \uplus SH_v) = \{\{v_3\}\} \cup (\{\{res\}\} \uplus \{\{v_0, v_1, v_2\}\})$=$\{\{v_3\}\} \cup \{\{res\} \cup \{v_0, v_1, v_2\}\} = \{\{v_0, v_1, v_2, res\}, \{v_3\}\}$.

Field Load: In the case that v`.f` is null, there is no change in the existing set sharing. Because the expression of SH' includes SH, that case is correctly approximated. When v`.f` is not null, we know that the object being assigned to res is reachable from v. The other variables that share with v in SH may or may not share with res in SH'. In the state G_0 of Fig. 3, although v_2 shares with v_0 in the initial and final states, it does not share with res in the final state; however, v_1 will share with both res and v_0 after the load. We write $\{\{v, res\}\} \uplus \bigcup_{S \in SH_v} \mathcal{P}(S|_{-v}))$ to account for objects reachable from v which become also reachable from res, and may be reachable from any subset of the variables that shared with v in SH. Objects not reachable from v (SH_{-v}) are accounted for by the union with SH. For instance, in the same state G_0, if $\{v_3\} \in SH$, then the load of v_0`.f` does not alter that particular element, which has to also be present in SH'.

Example 2. The graphs in Fig. 3 illustrate three different memory states before the evaluation of v_0.`f`. They correspond to the graphs in Fig. 1, but this time we indicate the type of every object and the object pointed to by res after the expression evaluation. The initial set sharing is identical in all cases: $sh(G_0) = sh(G_1) = sh(G_2) = \{\{v_0, v_1, v_2\}, \{v_3\}\}$. However, the evaluation results in a different set sharing for each resulting graph G'_i: $sh(G'_0) = \{\{v_0, v_1, v_2\}, \{v_0, v_1, res\}, \{v_3\}\}$, $sh(G'_1) = \{\{v_0, v_1, v_2, res\}, \{v_3\}\}$, and $sh(G'_2) = \{\{v_0, v_1, v_2\}, \{v_0, res\}, \{v_3\}\}$. Assume that the abstract state that approximates all the initial concrete states is also $SH = \{\{v_0, v_1, v_2\}, \{v_3\}\}$. The transfer function for v_0.`f` results in a final abstract state $SH' = SH \cup (\{\{v_0, res\}\} \uplus \mathcal{P}(\{v_1, v_2\}))$ = $\{\{v_0, v_1, v_2\}, \{v_3\}\} \cup (\{\{v_0, res\}\} \uplus \{\{\}, \{v_1\}, \{v_2\}, \{v_1, v_2\}\}) = \{\{v_0, v_1, v_2\}, \{v_0, v_1, v_2, res\}, \{v_0, v_1, res\}, \{v_0, v_2, res\}, \{v_0, res\}, \{v_3\}\}$. As required, all the sharings $sh(G'_0)$, $sh(G'_1)$, and $sh(G'_2)$ are included in SH'.

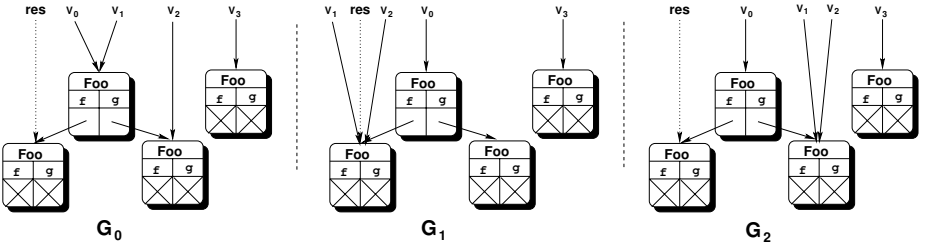


Fig. 3. Three concrete states

3.4 Semantics of Commands

Variable Store: For a store of the form v=$expr$, the semantics comprises three steps. First, the expression on the right-hand side is evaluated. Second, all ocurrences of v are removed from the current abstract state, since the value of v is being overwritten. Finally, all appearances of res are replaced by v, which deletes res from the abstract state.

Field Store: First, we evaluate the expression whose result is being stored; SH_1 contains that intermediate value. Sharings in SH_1 unrelated to v or res are unaffected by the store and contained in $SH_2 = SH_{1_{-\{v,res\}}}$, which is a subset of the final state. For each sharing in SH_{1_v}, the store might affect the reachability set of each variable involved and result in many smaller sharings. For example, in a memory state like G in Fig. 4, an assignment to v_0`.f` destroys any sharing between v_0 and v_1 (note that res does not share with v_1), but not the one between v_0 and v_2. All the possible combinations for the final sharings that have to do with v are contained in $SH_3 = \bigcup_{S \in SH_{1_v}} \mathcal{P}(S) \setminus \{\{\}\}$.

Now, for every sharing in SH_3 that contains v we have two possibilities: all the variables share also with res (and therefore, with $SH_{1_{res}}$), or none of them does. Note that every possible intermediate case in which just a few of the variables share with $SH_{1_{res}}$ is represented by a smaller subset in SH_3 containing only those variables. While $SH_4 = SH_{1_{res}} \uplus SH_{3_v}$ includes the combinations in which all the variables do share with $SH_{1_{res}}$, SH_3 approximates the situations in which none of them do share with res.

Example 3. Assume an initial state (after evaluating the expression) G depicted in Fig. 4. The dotted edge indicates where v_0`.f` will point after the execution of v_0`.f`$=$ $expr$. The initial set sharing is $sh(G) = \{\{v_0, v_1\}, \{v_0, v_2\}, \{res\}\}$. After the load, $sh(G') = \{\{v_0, v_2\}, \{v_0, res\}, \{v_1\}\}$. Assume that the starting abstract state, after the evaluation of the expression $expr$, is also $SH_1 = \{\{v_0, v_1\}, \{v_0, v_2\}, \{res\}\}$. Since there is no sharing unrelated to v or res, $SH_2 = \emptyset$. The next step is to calculate $SH_3 = \mathcal{P}(\{v_0, v_1\}) \cup \mathcal{P}(\{v_0, v_2\}) \setminus \{\{\}\}$= $\{\{v_0\}, \{v_0, v_1\}, \{v_1\}\} \cup \{\{v_0\}, \{v_0, v_2\}, \{v_2\}\}$ = $\{\{v_0\}, \{v_0, v_1\}, \{v_0, v_2\}, \{v_1\}, \{v_2\}\}$. Since $SH_{1_{res}} = \{\{res\}\}$ and $SH_{3_{v0}} =$

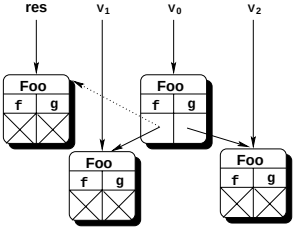


Fig. 4. Graph G

$\mathcal{SC}^I_\pi[\![v{=}expr]\!](SH)$
$SH_1 = \mathcal{SE}^I_\pi[\![expr]\!](SH)$ $SH_2 = SH_1\|_{-v}$ $SH' = SH_2\|^v_{res}$
$\mathcal{SC}^I_\pi[\![v\,.\,\texttt{f}{=}expr]\!](SH)$
$SH_1 = \mathcal{SE}^I_\pi[\![expr]\!](SH)$ $SH_2 = SH_{1_{-\{v,res\}}}$ $SH_3 = \bigcup_{S \in SH_{1_v}} \mathcal{P}(S) \setminus \{\{\}\}$ $SH_4 = SH_{1_{res}} \uplus SH_{3_v}$ $SH' = \begin{cases} \bot \text{ if } \textbf{mustBeNull}(SH_1, v) \\ SH_2 \cup (SH_3 \cup SH_4)\|_{-res} \text{ else} \end{cases}$

$\mathcal{SC}^I_\pi[\![\texttt{if } v{=}{=}\texttt{null } com_1 \texttt{ else } com_2]\!](SH)$
$SH_1 = \mathcal{SC}^I_\pi[\![com_1]\!](SH\|_{-v})$ $SH_2 = \mathcal{SC}^I_\pi[\![com_2]\!](SH)$ $SH' = \begin{cases} SH_1 \text{ if } \textbf{mustBeNull}(SH, v) \\ SH_1 \cup SH_2 \text{ if } \textbf{mayBeNull}(\tau, v) \\ SH_2 \text{ else} \end{cases}$
$\mathcal{SC}^I_\pi[\![\texttt{if } v{=}{=}w\ com_1 \texttt{ else } com_2]\!](SH)$
$SH_1 = \mathcal{SC}^I_\pi[\![com_1]\!](SH)$ $SH_2 = \mathcal{SC}^I_\pi[\![com_2]\!](SH)$ $SH' = \begin{cases} SH_1 \text{ if } \textbf{mustAlias}(SH, v, w) \\ SH_1 \cup SH_2 \text{ if } \textbf{mayAlias}(SH, v, w) \\ SH_2 \text{ else} \end{cases}$
$\mathcal{SC}^I_\pi[\![com_1\,;\,com_2]\!](SH)$
$SH' = \mathcal{SC}^I_\pi[\![com_2]\!](\mathcal{SC}^I_\pi[\![com_1]\!](SH))$

Fig. 5. Abstract semantics for the commands

$\{\{v_0\}, \{v_0, v_1\}, \{v_0, v_2\}\}$, $SH_4 = \{\{v_0, res\}, \{v_0, v_1, res\}, \{v_0, v_2, res\}\}$. The final abstract state $SH' = \{\{v_0\}, \{v_0, v_1\}, \{v_0, v_2\}, \{v_1\}, \{v_2\}\}$ is the union of $SH_3|_{-res} = SH_3$ and $SH_4|_{-res} \subset SH_3$. As required, $sh(G') \subseteq SH'$ holds after the removal of the auxiliary variable res from G'.

Conditional Statements: In the case where the guard is (v==null), the type component may contain definite information about whether a variable v is not null ($null \notin \tau(v)$). If we cannot determine exactly the nullity of v (i.e., **mayBeNull**(τ, v) is true), then the final state is the least upper bound of the resulting set sharing for the two branches. In particular, $SH_1 \sqcup SH_2 = SH_1 \cup SH_2$.

In the case where the condition is v==w, the sharing information may be enough to tell that the two variables are definitely equal, because they are both null: **mustAlias**(SH, v, w)=(**mustBeNull**(SH, v) and **mustBeNull**(SH, w)). On the other hand, v and w do not share if they do not appear together within a subset of SH. Therefore **mayAlias**(SH, v, w) = ($\overline{\textbf{mustAlias}}(SH, v, w)$ and $SH_{\{v,w\}} \neq \emptyset$). It is important to see that sharing information does not imply equality: a set sharing like $\{\{v, w\}\}$ indicates that v and w might reach a common object, not that they must be aliases.

Example 4. Given a command like `if (cond)` $v0 = v1$ `else` $\{v0 =$ `null;` $v1 =$ `null`$\}$, and assuming an initial abstract state $SH = \emptyset$ that does not contain enough information to determine $cond$, the set sharing corresponding to the `if` branch is $SH_1 = \{\{v_0, v_1\}\}$. The abstract state after simulating the `else` branch is $SH_2 = \{\}$. Therefore, the final state is $SH' = SH_1 \cup SH_2 = \{\{v_0, v_1\}\}$. However, SH' does not imply that v_0 necessarily shares with v_1, even when they appear together in SH', but that v_0 *might* reach an object reachable from v_1 in some of the concrete states approximatted by SH'; in the example, if $cond$ would be false, both variables are null and do not share.

4 Semantics as ZBDD Operations

Zero-suppressed BDDs (ZBDDs) [8,9] are a data structure similar to binary decision diagrams (BDDs) [3], but designed to encode sets of combinations (i.e., sets of sets of primitive elements). To encode the set sharing domain using ZBDDs, we define the primitive elements to be the variables in the program being analyzed. ZBDDs have been demostrated to perform better [14,15] than standard BDDs when encoding sets of combinations that are sparse in the sense that a) the set contains just a small fraction of all the possible combinations, and b) each combination contains just a few literals. A ZBDD is a rooted directed acyclic graph (DAG) of non-terminal and terminal nodes. Each non-terminal ZBDD node is labeled with a variable, and has two outgoing edges to other nodes, called the zero-edge and the one-edge. There are two terminal nodes, the zero node and the one node. They do not have variables or outgoing edges. The universe of all variables is totally ordered, and the order of the variables appearing on the nodes of any path through the ZBDD is consistent with the total order. Each path through the ZBDD that ends at the one terminal node defines a set of variables. The set contains a variable v if the path passes through a node labeled with v, and leaves the node along its one edge. Assuming the variable ordering is fixed, the smallest ZBDD representing a given set of sets is unique, and can be found efficiently.

Example 5. Assume a set of variables $Var = \{v_0, v_1, v_2\}$ and the variable ordering v_0, v_1, v_2. The unique smallest ZBDD representing the set of sets $\{\{v_0, v_2\}, \{v_1\}\}$ is the ZBDD shown in Fig. 6. There are two paths from the root of the ZBDD to the one terminal node. On the path containing the v_0 and v_1, only the node labeled v_1 is exited through the one edge; thus, this path represents the set $\{v_1\}$. On the path containing v_0 and v_2, both nodes are exited through their one edges; thus, this path represents the set $\{v_0, v_2\}$.

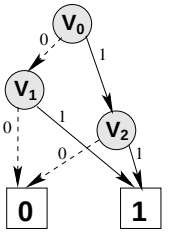


Fig. 6.

Efficient algorithms exist for common operations on the set of sets encoded by a ZBDD, including union (denoted +), intersection, set difference, product ($SH_1 * SH_2 = \{S_1 \cup S_2 \mid S_1 \in SH_1 \text{ and } S_2 \in SH_2\}$), and division ($SH/v = \{S \setminus \{v\} \mid S \in SH \text{ and } v \in S\}$ and $SH\%v = \{S \in SH \mid v \notin S\}$).

A set sharing like $SH = \{\{v_0, v_2\}, \{v_1\}\}$ is expressed in ZBDD notation as $SH = v_0v_2 + v_1$. Note that we will denote single literal sets by a single lower case letter (like v), while generic ZBDDs will be referred to with double upper case (normally, SH). For instance, given the set sharings $SH = v_0v_2 + v_1$ and v_0 , an expression like $SH * v_0 = v_0v_1 + v_0v_2$ is legal. The empty set is written as 0, and the set containing only the empty set is written as 1.

4.1 Expressions and Commands; Native Operations

Figs. 7 and 9 show the ZBDD version of the transfer functions[2] in Fig. 2 and 5. For most of the set operations, there is an equivalent native ZBDD operation. For instance,

[2] The type component is again omitted, although in practice it is updated in an identical fashion to Fig. 13 and 14.

$\mathcal{SE}^I_\pi[\![\texttt{null}]\!](SH)$
$SH' = SH$
$\mathcal{SE}^I_\pi[\![\texttt{new } k]\!](SH)$
$SH' = SH + res$
$\mathcal{SE}^I_\pi[\![v]\!](SH)$
$SH' =$ **setResEqTo**(SH, v)
$\mathcal{SE}^I_\pi[\![v\,.\,\texttt{f}]\!](SH)$
$SH' = \begin{cases} \bot \text{ if } \textbf{mustBeNull}(SH, v) \\ SH + v * res * \textbf{powUnion}(SH/v) \text{ else} \end{cases}$

```
setResEqTo(P) {
  if (P = 0 or P = 1 or P.top > v)
    return P
  if (P.top < v))
    return Getnode(P.top, P0, P1)
  return Getnode(P.top, P0, res*P1)
}

powUnion(P) {
  if (P = 0 or P = 1)
    return P
  R0 ← powUnion(P0)
  R1 ← powUnion(P1)
  return Getnode(P.top, Ro + R1, 1 + R1)
}
```

Fig. 7. Abstract semantics for the expressions as ZBDD operations

$SH_1 \uplus SH_2$ is equivalent to $SH_1 * SH_2$ and SH_{-v} is equivalent to $SH\%v$. This correspondence is useful because it results in no gap between the denotational semantics of Sect. 3 and the implementation. However, we added a number of non-standard ZBDD operators to improve the readability of the equations. The set of elements in SH containing v (SH_v, in set notation) is obtained via $SH//v = SH/v * v$. We delete all the ocurrences of v in SH using **projOut**$(SH, v) = SH/v + SH\%v - 1$. The unit set 1 (which represents the set containing the empty set) has to be deleted because SH might contain the single literal v, as we did in the corresponding *project out* set operator $SH|_{-v}$.

In other occasions, we created new ZBDD operators because of efficiency reasons. For instance, the variable load set equation $SH' = (\{\{res\}\} \uplus SH_v) \cup SH_{-v}$ can be expressed as $SH' = res*(SH//v) + SH\%v$. This combination of standard operators, while intuitive, has the disadvantage of being inefficient in practice. Since we expect this function to be invoked with high frequency (every time a variable is on the right hand side of an assignment), we devised a dedicated ZBDD algorithm that computes the same result, **setResEqTo**(SH, v). The algorithm, shown in Fig. 7, uses the same notation as in [9]: P_0 and P_1 for the graph reachable through the zero-edge and one-edge, respectively, $P.top$ for the current variable, and $Getnode(v, P_0, P_1)$ for the procedure that generates a node with the variable v and subgraphs P_0 and P_1. The correctness of **setResEqTo**(SH, v) is based on a variable order in which res is always the last variable, the one closer to the leaves. Given this precondition, we only need to find v in the graph, and then multiply its one-edge child by res, which will preserve the variable order.

With the basic ZBDD operators and **setResEqTo** we can understand the transfer functions of the null, new, and variable load expressions. The field load, on the other hand, depends on the ZBDD version of the predicate that determines whether a variable is null: **mustBeNull**(SH, v) = $(SH/v = 0)$. It also requires computing the union of the powersets of the elements of a set sharing SH: $\{\mathcal{P}(S) \mid S \in SH\}$. Although this seems to be a complex operation, it has a very natural description in terms of an algorithm in ZBDDs. We have devised a native ZBDD algorithm, **powUnion**(SH), shown in pseucode in Fig. 7. The correctness proof of the algorithm is given in the appendix.

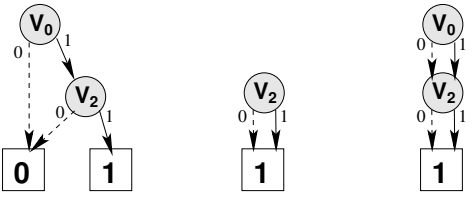


Fig. 8. ZBDDs representing v_0v_2, $1 + v_2$, and $1 + v_0 + v_0v_2 + v_2$

$\mathcal{SC}^I_\pi[\![v{=}expr]\!](SH)$
$SH_1 = \mathcal{SE}^I_\pi[\![expr]\!](SH)$ $SH_2 = \textbf{projOut}(SH_1 \% res, v)$ $SH' = SH_1/res * v + SH_2$
$\mathcal{SC}^I_\pi[\![v\,.\,\texttt{f}{=}expr]\!](SH)$
$SH_1 = \mathcal{SE}^I_\pi[\![expr]\!](SH)$ $SH_2 = SH_1 \% v \% res$ $SH_3 = \textbf{projOut}($ $\quad \textbf{powUnion}(SH_1 /\!/ v) - 1, res)$ $SH_4 = (SH_1/res) * (SH_3 /\!/ v)$ $SH' = \begin{cases} \bot \text{ if } \textbf{mustBeNull}(SH_1, v) \\ SH_2 + SH_3 + SH_4 \text{ else} \end{cases}$

$\mathcal{SC}^I_\pi[\![\texttt{if } v{=}{=}\texttt{null } com_1 \texttt{ else } com_2]\!](SH)$
$SH_1 = \mathcal{SC}^I_\pi[\![com_1]\!](\textbf{projOut}(SH, v))$ $SH_2 = \mathcal{SC}^I_\pi[\![com_2]\!](SH)$ $SH' = \begin{cases} SH_1 \text{ if } \textbf{mustBeNull}(SH, v) \\ SH_1 + SH_2 \text{ if } \textbf{mayBeNull}(\tau, v) \\ SH_2 \text{ else} \end{cases}$
$\mathcal{SC}^I_\pi[\![\texttt{if } v{=}{=}w\ com_1 \texttt{ else } com_2]\!](SH)$
$SH_1 = \mathcal{SC}^I_\pi[\![com_1]\!](SH)$ $SH_2 = \mathcal{SC}^I_\pi[\![com_2]\!](SH)$ $SH' = \begin{cases} SH_1 \text{ if } \textbf{mustAlias}(SH, v, w) \\ SH_1 + SH_2 \text{ if } \textbf{mayAlias}(SH, v, w) \\ SH_2 \text{ else} \end{cases}$
$\mathcal{SC}^I_\pi[\![com_1\,;\,com_2]\!](SH)$
$SH' = \mathcal{SC}^I_\pi[\![com_2]\!](\mathcal{SC}^I_\pi[\![com_1]\!](SH))$

Fig. 9. Abstract semantics for the commands

This native implementation will prove to be fundamental for the scalability of the analysis (Sect. 5).

Example 6. We show how the native algorithm computes **powUnion**(v_0v_2). Fig. 8 contains the initial ZBDD representing v_0v_2 (left). To compute **powUnion** for the original ZBDD, we first recursively compute **powUnion** for the node labeled v_2. When **powUnion** is applied to the node labeled v_2, which represents the set v_2, we have $R_0 = P_0 = 0$ and $R_1 = P_1 = 1$. The result is a node labeled v_2 with zero successor $R_0 + R_1 = 1$ and one successor $1 + R_1 = 1 + 1 = 1$, shown in the center of the figure. This ZBDD represents the powerset of v_2, namely $1+v_2$. We will call this ZBDD N. When we compute **powUnion** of the original ZBDD, $R_0 = P_0 = 0$, and $R_1 = N$. This step generates a node with value v_0, zero successor $R_0 + R_1 = 0 + N = N$, and one successor $1 + R_1 = 1 + N = N$. Because both nodes are identical (*reduction rule* applied within $Getnode$), we can delete one of them and change both edges of v_0 to lead to just one N, as shown in the right ZBDD in Fig. 8. The resulting graph represents $1 + v_0 + v_0v_2 + v_2$.

The command semantics (Fig. 9) is described in terms of the operators listed before. We only add a new predicate, used when checking if two variables might be aliases: **mayAlias**$(SH, v, w) = (\overline{\textbf{mustAlias}}(SH, v, w)$ and $SH/(v * w) \neq 0)$. The following example shows how the field store from Example 3 would be calculated using ZBDDs.

Example 7. Assume we start evaluating v_0 `.f=` $expr$ in an abstract set sharing $SH_1 = v_0v_1 + v_0v_2 + res$. Because all the sharings in SH_1 contain v_0 or res, $SH_2 = 0$. The union of the powersets of $SH_1//v_0 = v_0v_1+v_0v_2$ is calculated in a very similar fashion to the last example, and results in a set sharing $1+v_0+v_0v_1+v_0v_2+v_1+v_2$. Therefore, $SH_3 =$ **projOut**$(v_0+v_0v_1+v_0v_2+v_1+v_2, res) = v_0+v_0v_1+v_0v_2+v_1+v_2$. The last component of the result is $SH_4 = (SH_1/res) * (SH_3//v_0) = 1 * (SH_3//v_0) = v_0 + v_0v_1+v_0v_2$. The result is $SH' = 0+SH_3+SH_4 = SH_3 = v_0+v_0v_1+v_0v_2+v_1+v_2$, which is the same result obtained in the set example.

5 Experiments

To evaluate the scalability (in terms of memory usage and running time) of the ZBDD approach, we compared it to an alternative representation for set sharings based on sets of bitsets. Bitsets are a fast, light representation compared to other ways of representing a set sharing. In a bitset, each bit b_i indicates if the variable v_i is in the sharing ($b_i = 1$) or not ($b_i = 0$). Our first implementation used the Java library where a `BitSet` is an array of double words. However, our first experiments showed that this approach does not scale beyond set sharings with more than a few thousand elements. For this reason, we replaced the library implementation by a lightweight version, which only requires a single word to represent each sharing. This effectively limits the number of variables to be not more than 32 for the bitset approach, which is reasonable when confronted with powerset operations. In all the experiments we assume that the number of variables n is bounded by 32, but note that the ZBDD implementation scales well for larger set sharings, and could handle bigger values of n. Our ZBDD implementation of set sharing is based on the JDD library [21].

Several characteristics of set sharings influence the memory usage and the performance of the data structure representing them. Although the number of variables n seems to be important, our two representations are independent of this parameter. In the case of the bitsets, because we use 32 bits to store every sharing, independently of the number of variables. In the case of ZBDDs, only the statistical distribution of the sharings (i.e., their sparsity) influences the number of nodes required to represent the information, and therefore the memory usage and performance of the ZBDD. For the same reason, the behavior of the two data structures is independent of the *sharing density* of SH, i.e., the proportion of the number of sharings over the maximum possible: $SH_d = |SH|/2^n$.

The most decisive factor is the number of sharings $|SH|$. Because we allocate a new bitset every time a new sharing is added, the performance of the set of bitsets approach is inversely proportional to $|SH|$. In the case of ZBDDs we also have to take into account the *variable density*. This metric is the average number of variables per sharing: $v_d = \frac{1}{n*|SH|} * \sum_{S \in SH} |S|$. A small variable density is synonymous with a sparse set sharing, and therefore we can expect the ZBDD to perform inversely proportional to the

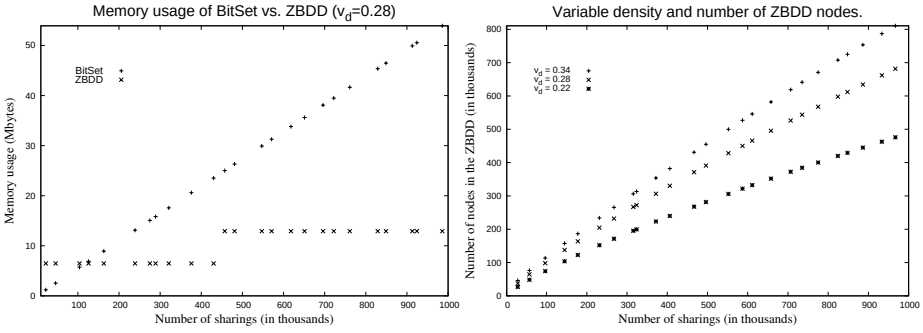


Fig. 10. Memory usage experiments. Over 25 runs.

metric. We now examine how the number of sharings and the variable density relate to memory consumption and execution times in our experiments.

Memory Usage: We generated random set sharings and measured the space requirements for the Java objects backing the set of bitsets and ZBDD as reported by a profiler [12]. The different memory usages are shown on the left of Fig. 10. The plot shows that the ZBDD scales better than the bitset solution. The differences are more significant (a factor of 5) for large values of $|SH|$. A set of bitsets uses 56 bytes per sharing, less than the 80 required by a set of the JDK 1.5 `BitSet` class. At one million sharings, the set of bitsets requires more than 56Mb, while the same information occupies 12Mb in the ZBDD version ($v_d = 0.28$). The staircase behavior of the ZBDD memory usage function is due to the *capacity* of the array storing the node list (ZBDDs are represented as arrays in JDD), which doubles when the load exceeds a certain threshold.

In the leftmost graph in Fig. 10 we did not take into account the effect of variable density. The other plot in that figure demonstrates how ZBDDs benefit from sparse variable distributions. This time we do not show the number of Kbytes in the y-axis, but rather the number of nodes in the binary decision diagram. As expected, sparse sharings require fewer nodes than those that are more dense in terms of v_d. In the experiments, the number of nodes goes down by an average 38.2% from $v_d = 0.34$ to $v_d = 0.22$.

Speed: We measured the number of milliseconds required to compute the semantics of the most significant operations (variable load/store, and field load/store), given a random initial set sharing. We disabled the JDD cache for the experiments. All the measurements were done on a Pentium M 1.73Ghz with 1Gb of RAM. The virtual machine was Sun's JVM 1.5.0 running on Ubuntu 6.06. The results are in Figs. 11 and 12.

The time required to simulate a variable load presents a similar, linear behavior in both cases; the bitset version is 14.6% faster in the average. Although not reflected in Fig. 11, the native operation **setResEqto** takes half the time of the equivalent composition of ZBDD operations (see Sect. 4). For the variable store, both running times are roughly linear in the number of sharings. However, the lack of a native ZBDD implementation results in running times noticeably slower than those of the set of bitsets. It

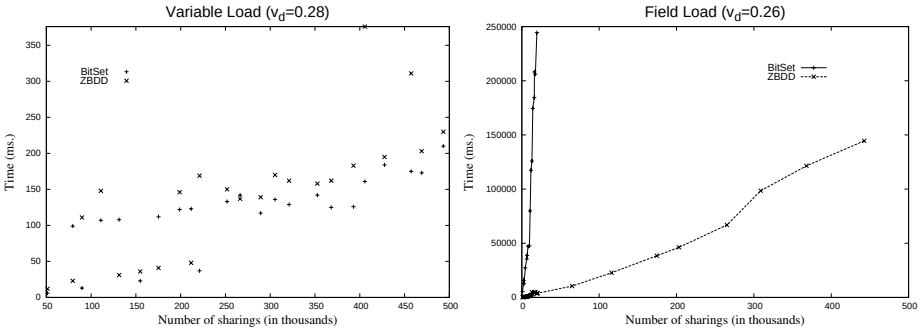


Fig. 11. Performance of a set of bitsets vs ZBDD (expressions). Over 25 runs.

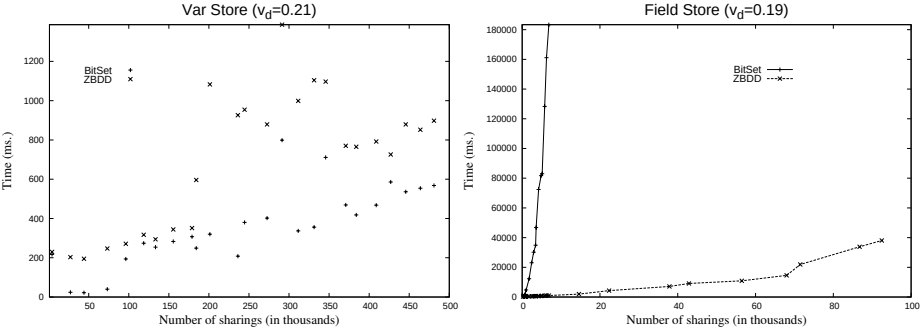


Fig. 12. Performance of a set of bitsets vs ZBDD (commands). Over 25 runs.

remains an open question whether a dedicated ZBDD algorithm can be devised for this command.

The powerset operation is a major obstacle for a feasible implementation of set sharing using the sets of bitsets. Both the field load and field store transfer functions depend on this operation. While the ZBDD **powUnion** algorithm requires reasonable times for calculating the union of many powersets, the bitset implementation presents exponential growth with respect to the number of sharings. For example, it needs half a minute to compute the output state for a field load in which the initial sharing has 5,000 elements. The ZBDD implementation finishes the same operation in less than 600ms. The field store (Fig. 12, right), which is a more complex operation, presents a similar pattern, although the running times are always significantly larger than for the field load.

6 Related Work

The ideas presented in this paper build on one hand on [16], where a first definition of a set sharing-based analysis for Java was introduced and shown to offer advantages in certain cases with respect to pair sharing-based analyses. We offer substantially improved definitions of the abstract semantics, a reduction in the number of components of an

abstract state, and in some cases (like the field load and store) more precise abstract operations. In addition, a significant difference with our previous work is of course the use of Zero-suppressed Decision Diagrams to efficiently implement the analysis domain. This is done without having to redesign any of the existing abstract operations. The experiments in [16] involved small set sharings (of at most 50 elements at a time) while in this paper we show how with ZBDDs we can scale up to thousands of sharings and still get reasonable times.

There has been extensive work in recent years on the use of BDDs [2,22,24,25] to represent (abstract) *points-to* information. In these abstractions, information is stored in the form of (v, a) pairs, where each such pair indicates that v may point to the allocation site a. As mentioned before, set sharing information can be interpreted as an *abstraction* of points-to information where instead of representing which exact objects can be pointed to by a variable, the domain captures only which sets of variables may point transitively to the same object. Thus, our analysis works at a different level since the set sharing encoding can result in some loss of precision, but offers the advantage of more compact representation.

ZBDDs were introduced by Minato [8] and applied to a great diversity of problems in model checking (e.g., [5,10,23]). More recently, Lhoták *et al.* have applied ZBDDs to the exploration of infinite state spaces [14] in the context of points-to analysis. The main differences between this work and [14] are one hand the abstraction used (set sharing vs. points-to pairs) and on the other that in the approach proposed the domain does not require relational information, i.e., we can use existing ZBDD libraries [20,21] directly in our implementation.

To the extent of our knowledge, this is the first work that relates set sharing analysis with ZBDDs or presents implementation results for the set-sharing domain using any type of binary decision diagram. In the logic programming realm, there has been a significant amount of work related to set sharing-based analysis for the automatic parallelization of Prolog programs (e.g., [11,17,18]). However, the abstract operations show significant differences with the ones required for an imperative/OO language. Furthermore, to the best of our knowledge, all existing implementations use lists of lists to represent set sharings. In [4] a connection between the set sharing domain and standard BDDs is suggested, but no implementation or experimental results are provided and there is no mention of ZBDDs. More recent work [19] for Java presents results for a BDD-based implementation of the less precise pair sharing domain [16]. Because in this case the abstraction is a set of pairs (and not a set of sets), the representation used is quite different from ours.

References

1. Bacon, D.F., Sweeney, P.F.: Fast static analysis of C++ virtual function calls. In: Proc. of OOPSLA 1996, SIGPLAN Notices, October 1996, vol. 31(10), pp. 324–341 (1996)
2. Berndl, M., Lhoták, O., Qian, F., Hendren, L., Umanee, N.: Points-to Analysis Using BDDs. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, pp. 103–114. ACM Press, New York (2003)
3. Bryant, R.E.: Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. ACM Comput. Surv. 24(3), 293–318 (1992)

4. Codish, M., Søndergaard, H., Stuckey, P.J.: Sharing and groundness dependencies in logic programs. ACM Transactions on Programming Languages and Systems 21(5), 948–976 (1999)
5. Coudert, O.: Solving Graph Optimization Problems with ZBDDs (1997)
6. Cousot, P., Cousot, R.: Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Fourth ACM Symposium on Principles of Programming Languages, pp. 238–252 (1977)
7. Dean, J., Grove, D., Chambers, C.: Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In: Olthoff, W. (ed.) ECOOP 1995. LNCS, vol. 952, pp. 77–101. Springer, Heidelberg (1995)
8. Minato, S.I.: Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. In: DAC, pp. 272–277 (1993)
9. Minato, S.I.: Binary Decision Diagrams and Applications for VLSICAD. Kluwer, Norwell (1996)
10. Minato, S.I.: Zero-suppressed BDDs and their Applications. STTT 3(2), 156–170 (2001)
11. Jacobs, D., Langen, A.: Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In: North American Conference on Logic Programming (1989)
12. JProfiler, `http://www.ej-technologies.com/products/jprofiler/`
13. Leroy, X.: Java bytecode verification: algorithms and formalizations. Journal of Automated Reasoning 30(3-4), 235–269 (2003)
14. Lhoták, O., Curial, S., Amaral, J.N.: Using ZBDDs in Points-to Analysis. In: Proceedings of the 20th International Workshop on Languages and Compilers for Parallel Computing (2007)
15. Meinel, C., Theobald, T.: Algorithms and Data Structures in VLSI Design. Springer, New York (1998)
16. Méndez-Lojo, M., Hermenegildo, M.: Precise Set Sharing Analysis for Java-style Programs. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 172–187. Springer, Heidelberg (2008)
17. Muthukumar, K., Hermenegildo, M.: Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In: 1989 North American Conference on Logic Programming, pp. 166–189. MIT Press, Cambridge (1989)
18. Muthukumar, K., Hermenegildo, M.: Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In: 1991 International Conference on Logic Programming, pp. 49–63. MIT Press, Cambridge (1991)
19. Payet, É., Spoto, F.: Magic-Sets Transformation for the Analysis of Java Bytecode. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 452–467. Springer, Heidelberg (2007)
20. Somenzi, F.: CUDD: CU Decision Diagram Package (2005), `http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html`
21. Vahidi, A.: JDD: A Pure Java BDD Library (2008), `http://javaddlib.sourceforge.net/jdd/index.html`
22. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: PLDI, pp. 131–144. ACM, New York (2004)
23. Yoneda, T., Hatori, H., Takahara, A., Minato, S.I.: BDDs vs. Zero-Suppressed BDDs: for CTL Symbolic Model Checking of petri nets. In: Srivas, M., Camilleri, A. (eds.) FMCAD 1996. LNCS, vol. 1166, pp. 435–449. Springer, Heidelberg (1996)
24. Zhu, J.: Symbolic Pointer Analysis. In: ICCAD, pp. 150–157 (2002)
25. Zhu, J., Calman, S.: Symbolic Pointer Analysis Revisited. In: PLDI, pp. 145–157 (2004)

A Complete Semantics for the Expressions and Commands

Contained in figures 13 and 14. In the case of the type component, the least upper bound is computed as $\tau_1 \sqcup \tau_2 = \{\, (v, \tau_1(v) \cup \tau_2(v)) \mid v \in Var \}$.

$$\mathcal{SE}^I_\pi[\![\texttt{null}]\!](SH,\tau)$$

$$SH' = SH$$
$$\tau' = \tau[res \mapsto \{\text{null}\}]$$

$$\mathcal{SE}^I_\pi[\![\texttt{new } k]\!](SH,\tau)$$

$$SH' = SH \cup \{\{res\}\}$$
$$\tau' = \tau[res \mapsto \{\text{k}\}]$$

$$\mathcal{SE}^I_\pi[\![v]\!](SH,\tau)$$

$$SH' = (\{\{res\}\} \uplus SH_v) \cup SH_{-v}$$
$$\tau' = \tau[res \mapsto \tau(v)]$$

$$\mathcal{SE}^I_\pi[\![v\texttt{.f}]\!](SH,\tau)$$

$$SH' = \begin{cases} \bot \text{ if } \textbf{mustBeNull}(SH, v) \\ SH \cup (\{\{v, res\}\} \uplus \bigcup_{S \in SH_v} \mathcal{P}(S|_{-v})) \text{ else} \end{cases}$$
$$\tau_1 = \tau[v \mapsto (\tau(v) \setminus \{\text{null}\}), res \mapsto (\downarrow F(v.f) \cup \{\text{null}\})]$$

Fig. 13. Abstract semantics for the expressions as set operations

$$\mathcal{SC}^I_\pi[\![v\texttt{=}expr]\!](SH,\tau)$$

$$(SH_1, \tau_1) = \mathcal{SE}^I_\pi[\![expr]\!](SH,\tau)$$
$$SH_2 = SH_1|_{-v}$$
$$SH' = SH_2|^v_{res}$$
$$\tau' = \tau_1[v \mapsto \tau_1(res)] \setminus (res, \tau_1(res))$$

$$\mathcal{SC}^I_\pi[\![v\texttt{.f=}expr]\!](SH,\tau)$$

$$(SH_1, \tau_1) = \mathcal{SE}^I_\pi[\![expr]\!](SH,\tau)$$
$$SH_2 = SH_{1_{-\{v,res\}}}$$
$$SH_3 = \bigcup_{S \in SH_{1_v}} \mathcal{P}(S) \setminus \{\{\}\}$$
$$SH_4 = SH_{1_{res}} \uplus SH_{3_v}$$
$$SH' = \begin{cases} \bot \text{ if } \textbf{mustBeNull}(SH_1, v) \\ SH_2 \cup (SH_3 \cup SH_4)|_{-res} \text{ else} \end{cases}$$
$$\tau' = \tau_1[v \mapsto (\tau_1(v) \setminus \{\text{null}\})] \setminus (res, \tau_1(res))$$

$$\mathcal{SC}^I_\pi[\![\texttt{if } v\texttt{==null } com_1 \texttt{ else } com_2]\!](SH,\tau)$$

$$SH_1 = SH|_{-v}$$
$$\tau_1 = \tau[v \mapsto \{\text{null}\}]$$
$$\sigma_1 = \mathcal{SC}^I_\pi[\![com_1]\!](SH_1, \tau_1)$$
$$\tau_2 = \tau[v \mapsto (\tau(v) \setminus \{\text{null}\})]$$
$$\sigma_2 = \mathcal{SC}^I_\pi[\![com_2]\!](SH, \tau_2)$$
$$(SH', \tau') = \begin{cases} \sigma_1 \text{ if } \textbf{mustBeNull}(SH, v) \\ \sigma_1 \sqcup \sigma_2 \text{ if } \textbf{mayBeNull}(\tau, v) \\ \sigma_2 \text{ else} \end{cases}$$

$$\mathcal{SC}^I_\pi[\![\texttt{if } v\texttt{==}w\ com_1 \texttt{ else } com_2]\!](SH,\tau)$$

$$\sigma_1 = \mathcal{SC}^I_\pi[\![com_1]\!](SH,\tau)$$
$$\sigma_2 = \mathcal{SC}^I_\pi[\![com_2]\!](SH,\tau)$$
$$(SH', \tau') = \begin{cases} \sigma_1 \text{ if } \textbf{mustAlias}(SH, v, w) \\ \sigma_1 \sqcup \sigma_2 \text{ if } \textbf{mayAlias}(SH, v, w) \\ \sigma_2 \text{ else} \end{cases}$$

$$\mathcal{SC}^I_\pi[\![com_1\texttt{;}\ com_2]\!](SH,\tau)$$

$$(SH', \tau') = \mathcal{SC}^I_\pi[\![com_2]\!](\mathcal{SC}^I_\pi[\![com_1]\!](SH,\tau))$$

Fig. 14. Abstract semantics for the commands

B PowUnion: Correctness Proof

Proof. **powUnion**(SH) correctly computes $\bigcup_{S \in SH} \mathcal{P}(S)$:
powUnion$(ZBDD(a, P_0, P_1)) = $ **powUnion**$(P_0 + a * P_1) = $ **powUnion**$(P_o) +$ **powUnion**$(a * P_1) = \bigcup_{S \in P_0} \mathcal{P}(S) \cup \bigcup_{S \in P_1} (\mathcal{P}(S \cup \{a\}) = \bigcup_{S \in P_0} \mathcal{P}(S) \cup \{\{a\}\} \cup \bigcup_{S \in P_1} (\mathcal{P}(S) \uplus \{\{\}, \{a\}\}) = \bigcup_{S \in P_0} \mathcal{P}(S) \cup \bigcup_{S \in P_1} \mathcal{P}(S) \cup \{\{a\}\} \cup \bigcup_{S \in P_1} (\mathcal{P}(S) \uplus \{\{a\}\}) = \bigcup_{S \in P_0 \cup P_1} \mathcal{P}(S) \cup \{\{a\}\} \cup (\{\{a\}\} \uplus \bigcup_{S \in P_1} \mathcal{P}(S)) = ZBDD(a, $ **powUnion**$(P_0 + P_1), 1 +$ **powUnion**$(P_1))$.

Register Bank Assignment for Spatially Partitioned Processors

Behnam Robatmili, Katherine Coons, Doug Burger, and Kathryn S. McKinley

Department of Computer Science, The University of Texas at Austin
{beroy,coonske,dburger,mckinely}@cs.utexas.edu

Abstract. Demand for instruction level parallelism calls for increasing register bandwidth without increasing the number of register ports. Emerging architectures address this need by partitioning registers into multiple distributed banks, which offers a technology scalable substrate but a challenging compilation target. This paper introduces a register allocator for spatially partitioned architectures. The allocator performs bank assignment together with allocation. It minimizes spill code and optimizes bank selection based on a priority function. This algorithm is unique because it must reason about multiple competing resource constraints and dependencies exposed by these architectures. We demonstrate an algorithm that uses critical path estimation, delays from registers to consuming functional units, and hardware resource constraints. We evaluate the algorithm on TRIPS, a functional, partitioned, tiled processor with register banks distributed on top of a 4×4 grid of ALUs. These results show that the priority banking algorithm implements a number of policies that improve performance, performance is sensitive to bank assignment, and the compiler manages this resource well.

1 Introduction

Traditional architectures offer a single register file with uniform delay for reading from and writing to any architectural (physical) register. Register allocation assigns variables (virtual registers) to architectural registers when possible. In traditional graph coloring [4] and linear scan [14] algorithms, the only goal is to minimize the overhead of load and store instructions created by spilling variables to memory.

To address current technology scaling challenges, emerging architectures partition resources such as registers, caches, and ALUs. This approach provides the bandwidth needed for high ILP programs while increasing the resources available on the chip. Spatially partitioned processors have non-uniform register access times, which place more burden on the register allocator, requiring a more sophisticated algorithm that intertwines bank and register assignment. To optimize for the partitioned layout of these processors, the register allocator must consider the location of register banks and data caches, and the placement of instructions on ALUs in order to decide which register bank to use for each register. The delay in reading or writing a register depends on the length of

J.N. Amaral (Ed.): LCPC 2008, LNCS 5335, pp. 64–79, 2008.

the path between the register bank and the ALU that reads or writes the register. Minimizing the communication latencies between partitioned components requires changes to traditional register allocation heuristics.

This paper proposes a bank allocation method for spatially partitioned architectures and evaluates it on the TRIPS hardware. The algorithm uses an evaluation function that selects a bank such that register values arriving at the same instruction at the same time are close to each other. The evaluation function calculates a score for each bank based on previously assigned banks of dependent instructions and the processor's topological characteristics.

We customize this evaluation function for TRIPS, a spatially partitioned tiled processor with register banks distributed in a row above a 4×4 array of ALUs. TRIPS is an instantiation of EDGE ISA in which each instruction encodes its consumer instructions directly. This direct, data-flow communication among instructions eliminates the need for an operand bypass network. Another characteristic of EDGE ISAs is atomic block execution, which amortizes the overhead of branch prediction and instruction cache access over a group of instructions. The results show that significant swings in performance are possible, and the algorithm improves over bank oblivious allocation by an average of 6%.

2 Related Work

Conventional register allocation methods. Chaitain et al. [4] present a graph-coloring register allocator that uses an interference graph (IG) to encode overlap between live ranges. Nodes represent variable live ranges and an edge indicates that the variables connected by that edge are simultaneously alive at some program point. A graph coloring register allocator assigns architectural registers to nodes such that two connected nodes do not receive the same color. If the graph is not colorable by N (the number of available physical registers), then some nodes are removed and the variables are spilled to memory to to make it colorable. The goal of a graph coloring allocator is to achieve an N-colorable graph with minimal spills [1,3].

Traub et al. [18] designed linear scan allocators that greedily scan the program variables to find a register assignment. Instead of using an interference graph, they directly use the live interval, which is the collection of instructions in which the variable is live. With good heuristics for ordering variables, the compiler can achieve the same code quality as graph coloring in many cases, but significantly faster. In this study, we start with a linear scan register allocator and add bank assignment functionality to it.

Bank assignment for clustered processors. In clustered processors, functional units and register files are partitioned or replicated and then grouped into on-chip clusters [9,12]. Clusters are connected through an inter-cluster communication network [11]. In each cluster, reads and writes are sent to the local register file (local reads/writes) or to remote register files in another cluster through the inter-cluster communication network. The register allocator attempts to minimize the number of remote register accesses.

Ellis generated code for VLIW processors with partitioned register files with a partitioning method called BUG (bottom-up greedy) intertwined with instruction scheduling [8]. Hiser et al. later extended that work by abstracting machine-dependent details into node and edge weights in a graph called the *register component graph* (RCG) [10]. The unconnected nodes (virtual registers) are good candidates to be assigned to separate banks. The algorithm first creates *an ideal instruction schedule*, assuming a single multiported register file. It then partitions the virtual registers by evaluating the benefit of assigning a given virtual register to each of the register banks, choosing the bank with the most benefit. The cost model includes the necessary copy instructions for virtual registers used in more than one partition. The algorithm runs as a pre-process before instruction scheduling and register allocation, which then use the specified banks to schedule instructions and allocate registers in partitions.

In this paper we address bank assignment for a different type of architecture in which registers, the ALUs, and the L1 cache banks all are physically partitioned and connected together via a lightweight on-chip network. The TRIPS architecture supports block atomic execution and direct dataflow communication among instructions in a block. Unlike other approaches [2,12], register allocation must occur before scheduling in TRIPS. Because blocks have a fixed size, the compiler must insert spills before placing instructions in blocks and then schedule [6]. Cluster architectures have a fixed inter-cluster communication delay, whereas in TRIPS, the register access delay depends on the distance between the register and the ALU of the instruction reading or writing that register.

The method we propose is most similar to Hiser et al. [10], but generalizes the register component graph to consider the arrival time of the virtual registers to each instruction. The algorithm also considers physical layout of the processor grid when choosing the best bank. Whereas Hiser et al. perform bank allocation as a pre-process before register allocation, our algorithm combines bank and register assignment. Each bank allocation decision thus uses information from prior allocation decisions, including spilled registers.

3 Background

Spatially partitioned uniprocessor architectures allow higher frequency operation at lower power, while exposing greater concurrency and data bandwidth. For example, the Raw processor integrates a low-latency on-chip network into its processor pipelines in a single-chip multiprocessor [17]. It provides programmable network routers for static routing, in which the programmer treats the Raw tiles as elements of a distributed serial processor. The most important characteristic of a spatially distributed architecture is that the topology of instruction placement is exposed in the ISA and performance is greatly dependent on both the exact placement of instructions and the time spent routing data between instructions.

We investigate register banking on the TRIPS processor, which is a spatially partitioned processor that uses an EDGE ISA. In an EDGE ISA, the compiler groups instructions into large, fixed-size blocks similar to hyperblocks [13]. The

ISA employs predication to form large blocks. Within each TRIPS block, the compiler encodes the instructions in dataflow form. Each instruction specifies where to send its result. At runtime, an individual instruction executes when it receives all of its operands, and each TRIPS block is executed atomically.

3.1 Overview of TRIPS

Figure 1 shows a diagram of a TRIPS processor core composed of a 2-D grid of 16 execution tiles (ETs), 4 distributed register tiles (RTs), 4 distributed L1 data cache tiles (DTs), 5 instruction cache tiles and a global control tile. Each ET has an integer unit, a floating-point unit, and reservation stations for instructions. Each RT includes 32 registers, resulting in a total register file capacity of 128 registers. The TRIPS tiles communicate using a lightweight operand network that dynamically routes operands and load/store traffic through intermediate tiles in Y-X order.

A TRIPS program consists of a series of instruction blocks that are individually mapped onto the array of execution tiles and executed atomically [15]. The compiler statically specifies where instructions execute, i.e., on which ET. The hardware determines when they execute by dynamically issuing instructions after their operands become available. The architecture reads inputs from registers and delivers them into the ET array. Operands produced and consumed within the array are delivered directly to the target instruction. Direct instruction communication requires no register file accesses. Operand arrival triggers instruction execution, thus implementing a dataflow execution model. A TRIPS block may hold up to 128 computation instructions with up to 8 mapped to any given ET.

The TRIPS ISA imposes several constraints on the blocks generated by the compiler:

- The maximum block size is 128 instructions.
- The number of register reads and writes in a TRIPS block is 32 reads (eight reads per register tile) and 32 writes (eight writes per register tile).
- The total number of executed load and store instructions in a TRIPS block must not exceed 32.

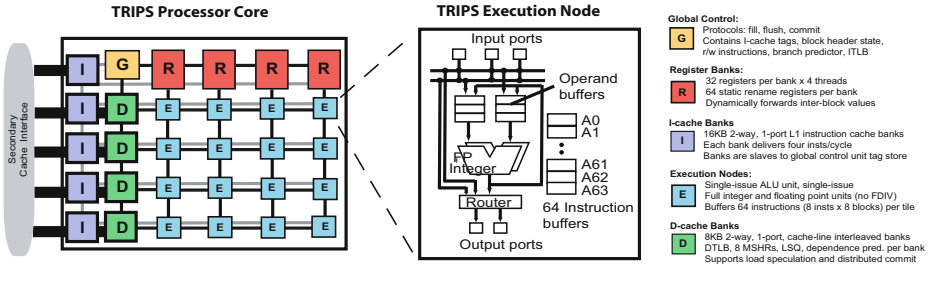


Fig. 1. A TRIPS Prototype Core

3.2 Compiling for TRIPS

To explain the interaction between the TRIPS register allocator and instruction scheduler, we describe different phases of the TRIPS compiler backend [15].

As shown in Figure 2, the first phase is block formation, in which the compiler combines basic blocks into a set of TRIPS blocks integrating if-conversion, predication, unrolling, tail duplication, and head duplication as necessary to form optimized blocks [13]. The compiler also performs scalar optimizations that merge redundant instructions and eliminate unnecessary predicates as an integrated step of block formation. During block formation, the compiler assumes an infinite virtual register set and uses a RISC-like intermediate form called TIL (TRIPS Intermediate Language).

After block formation, the compiler performs register allocation of virtual registers (variables) to physical registers. The variables defined and used inside a single block are not register allocated because EDGE instructions directly encode their consumer instructions. Therefore, the register allocator allocates only variables that are live-in or live-out across blocks. The allocator enforces the constraints on the TRIPS blocks regarding the number of reads and writes from each register tile. Allocation must occur prior to instruction scheduling because a spill could cause a block to violate the block size limit. In this case, the compiler performs reverse if-conversion, splits the block, and performs allocation again, until no spills violate the block constraints. We explain the register allocation algorithms in detail in the following sections.

The last phase in the TRIPS compiler backend is instruction scheduling, which outputs TRIPS Assembly Language (TASL). TASL fully specifies blocks and within blocks it encodes dataflow target form: each instruction has an identifier and each instruction may specify one or more instruction identifiers to which its result should be sent. The architecture maps instruction identifiers to execution tiles [6]. TASL is portable because the hardware can map instruction identifiers at run time based on various hardware topologies. The scheduler uses a cost function to choose an execution tile on which to place each instruction. This function considers features such as the communication among the dependent instructions, network delay, and network contention. It also considers the location

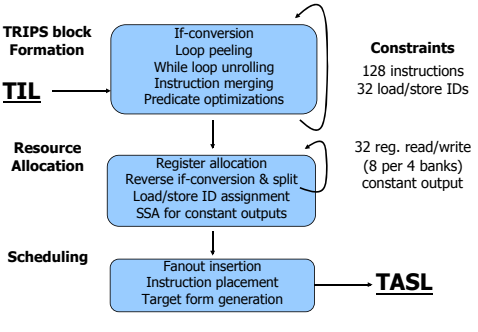


Fig. 2. TRIPS compiler overview

of the register bank (RT) of each register read or written by that instruction. The scheduler needs to know the register bank to which each variable is allocated to produce an efficient schedule.

3.3 Base Linear Scan Register Allocator

This allocator simply extends a linear scan algorithm. It performs a liveness analysis to compute the live range information at a block granularity. It sorts variables for allocation using a priority function similar to Chow and Hennessey's priority function [5]:

$$Priority_{DEF}(vr) = \sum_{i=LR(vr)} (Di * ST\_COST + Ui * LD\_COST) \qquad (1)$$

where binary values Di and Ui indicate whether the variable is defined or used in block B_i. ST_COST and LD_COST are the delays associated with store and load instructions, respectively. $LR(vr)$ returns a list of blocks in which variable vr is live. For each variable, the allocator considers all available physical registers, regardless of their register bank. For each register, the allocator tests for:

- **Live range conflicts:** The register live range must not conflict with the variable live range (i.e., the register has not already been assigned to another variable with an overlapping live range).
- **Block read/write conflicts:** The assignment must not violate the limit on the number of register reads or writes in the block (32 reads and 32 writes). Also, the assignment must not violate the limit on the number of bank accesses (8 reads and writes per bank) for all blocks that read or write the variable.

The allocator assigns the virtual register to a candidate register that meets these criteria and updates the live range of the physical register to encompass the live range of that variable. We configure this algorithm to perform bank-oblivious and round-robin assignments. Bank oblivious uses all the registers in the first bank, then the second, and so on. Round robin cycles through the banks as it assigns physical registers to variables in priority order.

If no physical register satisfies both the live range and bank conflict tests, the register allocator inserts spill loads and stores inside each block that uses or defines that virtual register. After a spill, register allocation is repeated to account for new live ranges generated by the spill code. An indirect effect of spilling on TRIPS that does not exist in conventional processors is that added loads and stores increase block sizes. If a block size exceeds the maximum (128 instructions for TRIPS), the block becomes invalid. The compiler forms valid blocks by splitting each invalid block into two blocks using reverse if-conversion [19], which creates new live ranges, and then performs register allocation again. To reduce the probability of splitting blocks, the allocator first identifies blocks that would overflow as a consequence of spilling and adds them to a list called $SIBLOCKS$

(spilling invalidate blocks). Using this list, the allocator increases the priority associated with registers used in those blocks:

$$Priority_{EDGE}(vr) = Priority_{DEF}(vr) + \alpha \sum_{i=LR(vr)\cap SIBLOCKS} \frac{SizeB_i}{128 - SizeB_i}. \quad (2)$$

where α is a fixed value greater than all possible values of $Priority_{EDGE}(vr)$, LR represents the live range of a virtual register, and $SizeB_i$ is the size of block B_i. Based on this function, any virtual register live in one of the blocks in $SIBLOCKS$ has higher priority than all virtual registers without this property.

4 Bank Assignment Algorithm for Spatially Partitioned Processors

This section explains the bank allocation algorithm for spatially partitioned processors. This algorithm, however, is not specific to the TRIPS processor and can be applied to other spatially partitioned processors. Therefore, this section explains the general algorithm, and the next section describes how we customize the algorithm for the TRIPS processor.

In spatially partitioned processors a lightweight network connects the register banks, ALUs, and data cache banks, which form a distributed 2-D substrate. A virtual register (variable) can be allocated to any of the register banks on the substrate, but the delay of accessing each register bank from an ALU on the substrate depends on the distance between the register and the ALU, as well as the contention in the network. For example, consider the sample substrate shown in Figure 3 with three ALUs ($A_{0...2}$) and three register banks ($B_{0...2}$) connected by the single delay-per-hop network shown with black lines. This figure shows two bank assignments for the variables $v_{0...3}$ where variables v_1 and v_2 are inputs to instruction i_0 in A_1 and variables v_0 and v_3 are inputs to instruction i_1 in A_2. The thick grey lines show the data transfers from register banks to the destination ALUs and the number beside each arrow indicates the arrival time of the corresponding register to that ALU. In the bank assignment on the left, the distances between variables v_0, v_1 and their destination instructions are three and two hops, respectively. However, the arrival of v_0 or v_1 will be delayed by

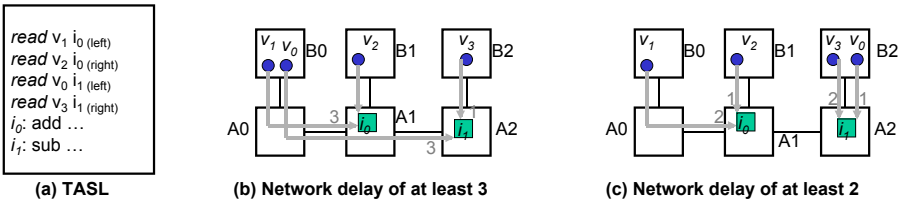


Fig. 3. Network delays resulting from different register bank assignments for a sample substrate

at least one cycle because both v_0 and v_1 use the same path, i.e., the path that connects register bank B_0 to A_0. Since the network only sends one value in each cycle, one of them will be delayed. Similarly, there is a network contention between variables v_0 and v_3 in the bank assignment on the right. This bank assignment, however, places dependent variables v_0 and v_3 in the same bank, resulting in fewer network hops, lower network congestion, and better overall timing for both instructions: a minimum network delay of 2 rather than 3.

The bank assignment algorithm first creates a graph called a register dependence graph (RDG). It then chooses an order in which to attempt to allocate virtual registers to architectural registers. For every virtual register in that ordered list, it uses a bank score evaluation function to calculate the benefit of placing the virtual register in each bank, and chooses the bank with the maximum score. The score for each bank is based on the banks of already allocated virtual registers and the weights of the edges of the RDG connecting that virtual register to other virtual registers. The register allocator next allocates the virtual register to one of the physical registers in that bank and the process continues.

4.1 Register Dependence Graph

The algorithm first builds a *register dependence graph* with nodes representing virtual registers and edges indicating dependences between virtual registers.

The weight on each edge between two virtual registers indicates the affinity between those two virtual registers. Lower values on an edge indicate that placing the virtual registers close together will improve the overall delay of the critical path. To create the RDG, the algorithm processes the blocks in the program one at a time. For each block, it estimates the execution time of instructions using an ideal schedule on the acyclic data flow graph (DFG) of that block. This ideal schedule assumes that all of the block's input registers arrive at the same time, and that there are no delays due to network contention. The algorithm traverses the DFG in a breadth-first order. For each instruction, it estimates the time its output virtual register is ready by adding the fixed execution delay associated with that instruction to the time when all its inputs are ready. Using this ideal schedule, the algorithm optimistically estimates when data from an input register will be available to its consumer instructions in the block DFG.

In a second pass, the algorithm traverses the DFG, keeping track of the variables that are ancestors of each instruction in the critical path. When an instruction has two different variables as ancestors, the algorithm places an edge between those variables with a weight equal to the difference between their estimated arrival times at that instruction. If an edge already exists between two virtual registers, the algorithm keeps track of the minimum weight for that edge. An edge with a low weight indicates that the two virtual registers should be placed close together.

Figure 4 provides sample intermediate code for a block, the block's DFG with the ideal estimated times, and the corresponding RDG. Variables a, b and c are the inputs to the block and must be kept in registers (other variables are temporary values within the block and are handled by the hardware). For this

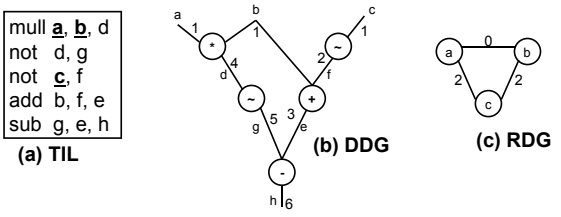


Fig. 4. (a) TIL block example, (b) Dataflow Dependence Graph (DDG) with ideal time estimates, and (c) Register Dependence Graph (RDG)

example, we assume that the execution time of a *mult* instruction is three cycles and the execution time of all other instructions is one cycle. The critical path of the block is the chain including the *mult, not,* and *sub* instructions. The chain of instructions coming from *a* and *b* intersect at the *mult* instruction at an estimated time of 1 cycle for both *a* and *b*. Because the *mult* instruction is on the critical path, we set the value on the link between *a* and *b* in the RDG to zero, meaning that the two virtual registers should be as close together as possible.

The chain of instructions originating at *b* intersects with the chain of instructions originating at *c* at the *add* instruction, and also at the sub instruction. The add instruction is not on the critical path, so this instruction is ignored. The sub instruction is on the critical path, however, so the weight of the edge between *b* and *c* is set to the difference between the arrival time of the data from each register at the *sub* instruction, $(5 - 3)$. Since we must perform scheduling after register allocation, the allocator assumes an ideal schedule to estimate latencies between instructions.

Each node in the RDG contains the following information for the corresponding virtual register:

- **Loop nesting depth:** If the virtual register is used or defined in more than one loop we select its maximum loop nesting depth.
- **Total number of instructions affected by the virtual register:** The number of instructions in the DDG which depend directly or transitively on this virtual register. For example, in Figure 4 the virtual registers *a* and *b* affect three and four instructions, respectively.

4.2 Bank Assignment

After building the RDG, the algorithm begins a combined bank assignment and register allocation phase. For a given virtual register, it first chooses the best register bank according to a heuristic function, and then tests if a physical register in that bank is available. If so, it assigns the virtual register to the physical register. Otherwise, the allocator tries to find an alternative register in another bank. Figure 5 shows the bank assignment algorithm. *PriorityOrder* determines the order in which the virtual registers are allocated. In classic register allocation studies, the priority for each virtual register is computed based on spill code

```
for each vr in PriorityOrder
    bestBank = 0
    bestScore = 0
    for each register bank b
        bankScore = CalculateBankScore(vr, b)
        if (bankScore > bestScore)
            bestScore = bankScore
            bestBank = b
        elsif (bankScore == bestScore)
            bestBank = TieBreak(vr, bestBank, b)
    reg = ChoosePhysicalRegisterFromBank(bestBank, vr)
    if (reg found)
        Replace vr with r in the code and update data for vr and bestBank
    else
        reg = ChoosePhysicalRegisterFromOtherBanks(bestBank, vr)
        if (reg found)
            Replace vr with r in the code and update data for vr and bestBank
        else
            Spill vr
```

Fig. 5. Bank Assignment Algorithm

```
CalculateBankScoreBasic(vr, bank)
return CalculateDependenceScore(vr, bank) - bank.numAssignedVR

CalculateDependenceScore(vr, bank)
score = 0
for each nvr RDG neighbor of vr assigned to NeighborBankSet(bank)
    score += (RDG-MAX-WEIGHT - RDG Weight(vr, nvr))
return score
```

Fig. 6. Basic Implementation of CalculateBankScore Function

overhead produced if that virtual register is spilled [4,5]. Because bank assignment is done in conjunction with register allocation, however, the priority order must also take into account the dependencies between virtual registers and their criticality. We define a simple priority function as follows:

$$Priority_{spatial}(vr) = 10^{LoopNestingDepth} + NumOfEdges(vr, RDG). \quad (3)$$

This function prioritizes virtual registers that have more dependencies with other virtual registers and affect more instructions in the DDG.

The bank allocation algorithm uses the *CalculateBankScore* cost function. Figure 6 shows a basic implementation of this function that uses the following components:

- *Dependence score*: A score based on the dependencies between the current virtual registers and the already allocated virtual registers. The function accumulates the weights of the RDG edges between the current virtual register and all virtual registers assigned to the current bank and its neighbor banks (referred to as *NeighborBankSet* in Figure 6).
- *Bank utilization penalty*: The number of registers already assigned to the bank. This component favors distributing virtual registers evenly across the banks to improve concurrency.

The algorithm uses a tie breaker function, *TieBreak*, to determine the bank when the scores of two banks for a given virtual register are identical. The definition of *NeighborBankSet* and *TieBreaker* functions depends on the physical layout and the characteristics of the processor grid. We explore implementations of these functions for TRIPS processor.

4.3 Customizing the Bank Score Evaluation Function for TRIPS

Because TRIPS has fewer register banks than execution tiles, we expect heavy traffic on the links connecting register banks to the execution tiles, and contention on those links affects performance. We implement a *TieBreaker* function to separate the load/store traffic, which flows from register banks to the data tiles, from the rest of the traffic, as shown in Figure 7. This function prioritizes register banks on the left (i.e., lower bank numbers) if there are critical load or store instructions dependent on the arrival time of the current virtual register. Otherwise, it prioritizes the register banks on the right to move the non-memory traffic out of the way of traffic to the data cache banks to avoid network contention.

```
TRIPS_TieBreaker(vr, bank1, bank2)
if (vr.affectedCriticalLoads + vr.affectedCriticalStores > 0)
    return min(bank1, bank2)
else
    return max(bank1, bank2)
```

Fig. 7. TieBreaker Function for TRIPS

5 Experimental Results

To evaluate performance we used the TRIPS hardware. The TRIPS chip is a custom 170 million transistor ASIC implemented in a 130nm technology. For these experiments, we ran the processor at 366MHz. The capacity of the L1 cache, L2 cache and main memory were 32 KB, 1 MB and 2GB, respectively. We collected cycle counts from the hardware performance counters using customized libraries and a runtime environment developed by the TRIPS team [20]. We used C and Fortran programs from the EEMBC benchmark suite with the iteration count set to 1000, and from the SPEC2000 benchmark suite [7,16].

For comparison, we adapted Hiser et al.'s algorithm (HCSB) to work with an EDGE ISA. We add a link in the RDG between two virtual registers if one is the input to a hyperblock, the other is an output from that same hyperblock, and there exists a dataflow path within the hyperblock from the input virtual register to the output virtual register. For example, consider the block in Figure 4. There will be edges between h and each of the inputs (a, b and c) in the RDG. The remaining operations from HCSB carry over without changes to an EDGE ISA.

We compare the following bank alloction algorithms on TRIPS:

- **Bank Oblivious:** The linear scan register allocation algorithm explained in Section 3.3 with no bank assignment mechanism. This algorithm uses all of the architectural registers in the current bank before using the registers in the next bank.
- **Round Robin:** The linear scan register allocation algorithm explained in Section 3.3 using round-robin bank allocation. This allocator chooses physical registers from banks in a round robin fashion.
- **HCSB:** Our implementation of Hiser et al. [10].
- **Spatial:** The bank allocation algorithm for spatially partitioned processors using the bank allocation priority function shown in Equation 3 and the basic bank score function shown in Figure 6.

Table 1 contains the number of static spill load and store instructions for each of the four allocators. Register allocation adds spill code to only 5 benchmarks out of the 39 EEMBC and SPEC benchmarks we evaluated. This low rate of spill code generation is partly because TRIPS has more registers than conventional architectures. In addition, the TRIPS compiler converts temporary values defined and used within a block to direct instruction communication that does not go through the register file, as it would on a conventional RISC processor.

For the benchmarks in Table 1, the spatial and HCSB bank allocators produce less spill code compared to the bank oblivious and round-robin allocators. The spatial and HCSB allocators prioritize variables based on the number of dependences according to the priority function shown in Equation 3. As a result, they allocate critical variables with higher numbers of dependences first. These dependences directly indicate the number of spill instructions. By prioritizing the variables with the most dependences first, if a variable later must be spilled, then it will likely have fewer dependences and thus require less spill code.

Table 1. Number of static spill load and store instructions. For the remaining benchmarks, none of the allocators generate any spill instructions.

Program	Benchmark suite	Bank oblivious	Round robin	HCSB	Spatial
a2time	EEMBC	111	111	30	31
applu	SPEC	528	514	365	382
apsi	SPEC	328	220	183	183
equake	SPEC	30	30	10	10
mgrid	SPEC	44	21	8	12

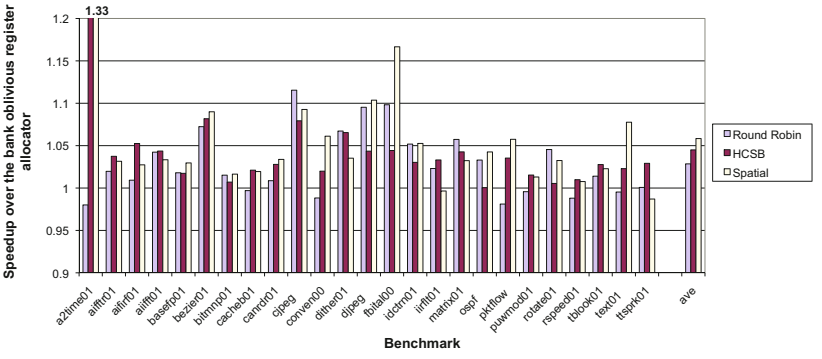


Fig. 8. The speedup achieved using different bank assignment algorithms for *EEMBC* on TRIPS compared to a bank-oblivious linear scan allocator

Figure 8 provides speedups using different bank assignment algorithms relative to the performance of the bank oblivious allocator. On average, the spatial bank assignment outperforms the other register allocators. The round-robin algorithm achieves a 3% performance improvement over the bank oblivious algorithm. This performance improvement results from a more balanced register distribution. HCSB improves performance over the round-robin bank allocation by placing dependent variables in nearby register banks. This algorithm, however, does not consider the arrival times of variables or the topology of the processor substrate. The spatial bank assignment algorithm places the virtual registers in the register banks according to their arrival times at the critical instructions. It also places registers used by critical load or store instructions to the register banks located close to the cache banks. On average, this bank assignment algorithm performs 6% better than the bank oblivious assignment.

For some programs, the spatial bank allocator performs significantly better than other allocators. The a2time benchmark has the largest speedup when using the spatial or HCSB allocators. Table 1 shows that a2time is the only EEMBC benchmark for which spill code is generated, and that the HCSB and spatial algorithms significantly reduce spilling for this benchmark.

In fbital, the spatial bank allocator achieves a high speedup over the bank oblivious allocator, and the round-robin allocator achieves the second best speedup. Figure 9 (a) illustrates the simplified version of the most frequently executed block of this program. In the critical path of this block (the grey lines in the figure), the computation chain starting from two virtual registers v_1 and v_2 ends by writing to variable v_2. Variables v_0 and v_1 are also inputs to a store memory operation. By separating memory and computation traffic, the spatial bank allocator places v_0, v_1 and v_2 in banks 0, 2, and 3, respectively. However, the HCSB bank allocator, which considers only register dependencies, places these dependent variables in banks 2, 1, and 1, respectively. Because of this allocation, the critical path suffers extra delays caused by the memory traffic of

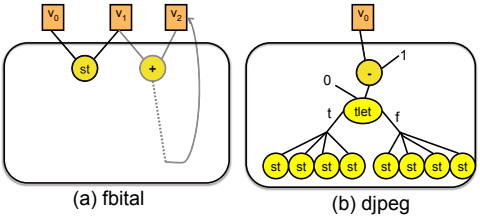


Fig. 9. The critical paths of fbital and djpeg EEMBC programs

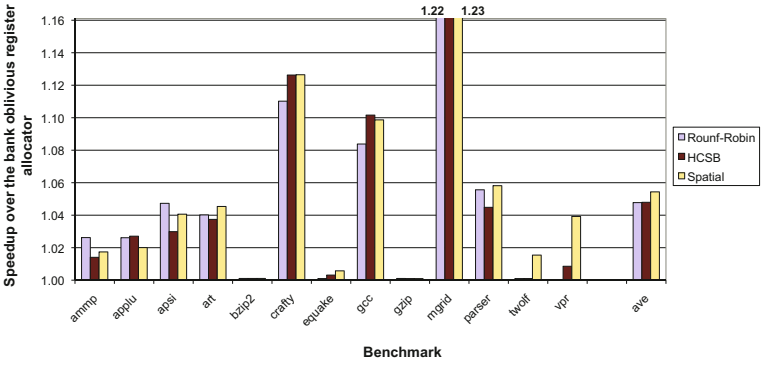


Fig. 10. The speedup achieved using different bank assignment methods for the *SPEC* benchmarks on TRIPS

the store instruction. Round robin randomly places v_1 and v_2 in banks 2 and 3, achieving better results than HCSB.

In the critical path of djpeg, a predicate condition is computed using an input variable, as shown in Figure 9 (b). Several parallel memory operations are executed on both predicate paths. The round-robin bank allocator places the critical variable in bank 0, which adds some delays to the predicate computations because of the high memory traffic. The HCSB allocator places that variable in bank 3, which is too far from the memory banks and adds some delays to the memory operations. Considering memory bank locations, the spatial bank allocator places that variable in bank 2, which results in the highest speedup over the bank oblivious allocator.

For some programs, round robin and HCSB perform better than spatial. Examples of such programs are aifirf01, aiifft01 and matrix01. We suspect that in these programs, the ideal schedule model used by the spatial bank assignment algorithm to generate the register dependence graph has inaccuracies caused by runtime resource constraints such as long latency cache misses.

Figure 10 illustrates the speedups using different bank assignment allocators relative to the performance achieved using the bank oblivious allocator for the SPEC benchmarks. Register allocation does not affect the SPEC benchmarks as strongly as it affects the EEMBC benchmarks, most likely because the SPEC

benchmarks are limited by other factors such as memory latency or instruction cache pressure. As a result, benchmarks such as gzip, bzip, and equake are not strongly affected. HCSB and round robin perform similarly on average, while the spatial bank allocator performs slightly better. This speedup may be the result of separating memory traffic from computation traffic.

6 Conclusions

In spatially partition processors, the delay of accessing registers depends on the location of the register files and the ALUs on the grid. Consequently, we introduce a register allocator that avoids spilling critical registers and estimates operand arrival times for critical instructions to choose banks for dependent registers wisely. The allocator also considers the topological characteristics of the hardware. In TRIPS, the memory tiles are located on the left side of the grid. Considering the location of register files, the spatial allocator separates memory traffic from computation traffic when assigning banks to critical registers. In addition, this allocator places dependent critical registers close together, so that critical instructions receive their operands faster. The individual effects of proximity of dependent registers and separation of memory and computation traffic is still an open question and requires further research.

Acknowledgments

We thank Jon Gibson and Aaron Smith who implemented the round-robin allocator. We also thank the entire TRIPS hardware and compiler teams.

This work is supported by DARPA F33615-03-C-4106, NSF EIA-0303609, Intel, IBM, and an NSF Graduate Research Fellowship. Any opinions and conclusions are the authors' and do not necessarily reflect those of the sponsors.

References

1. Bernstein, D., Golumbic, M., Mansour, Y., Pinter, R., Goldin, D., Nahshon, I., Krawczyk, H.: Spill code minimization techniques for optimizing compliers. In: ACM SIGPLAN Symposium on Interpreters and Interpretive Techniques, pp. 258–263 (1989)
2. Brasier, T.S., Sweany, P.H., Beaty, S.J., Carr, S.: CRAIG: a practical framework for combining instruction scheduling and register assignment. In: Parallel Architectures and Compilation Techniques, pp. 11–18 (1995)
3. Briggs, P., Cooper, K.D., Torczon, L.: Improvements to graph coloring register allocation. In: ACM Transactions on Programming Languages and Systems, May 1994, vol. 16(3), pp. 428–455 (1994)
4. Chaitin, G.: Register allocation and spilling via graph coloring. In: ACM SIGPLAN Symposium on Compiler Construction, pp. 98–105 (1982)
5. Chow, F.C., Hennessy, J.L.: Priority-based coloring approach to register allocation. In: ACM Transactions on Programming Languages and Systems, vol. 12, pp. 501–536 (1990)

6. Coons, K., Chen, X., Kushwaha, S., Burger, D., McKinley, K.S.: A spatial path scheduling algorithm for edge architectures. In: ACM Conference on Architecture Support for Programming Languages and Operating Systems, pp. 129–140 (2006)
7. EEMBC. Embedded microprocessor benchmark consortium, http://www.eembc.org/
8. Ellis, J.: A Compiler for VLIW Architecture. PhD thesis, Yale University (1984)
9. Farkas, K.L., Chow, P., Jouppi, N.P., Vranesic, Z.: The multicluster architecture: Reducing processor cycle time through partitioning. In: ACM/IEEE Symposium on Microarchitecture, pp. 327–356 (1997)
10. Hiser, J., Carr, S., Sweany, P., Beaty, S.J.: Register assignment for software pipelining with partitioned register banks. In: International Parallel and Distributed Processing Symposium, pp. 211–217 (2000)
11. Janssen, J., Corporaal, H.: Partitioned register files for TTAs. In: ACM/IEEE Symposium on Micorarchitecture, December 1995, pp. 301–312 (1995)
12. Kailas, K., Ebcioglu, K., Agrawala, A.: Cars: A new code generation framework for clustered ILP processors. In: Conference on High Performance Computer Architecture, pp. 133–143 (2001)
13. Maher, B., Smith, A., Burger, D., McKinley, K.S.: Merging head and tail duplication for convergent hyperblock formation. In: ACM/IEEE International Symposium on Microarchitecture, pp. 65–76 (2006)
14. Poletto, M., Sarkar, V.: Linear scan register allocation. In: ACM Transactions on Programming Languages and Systems, Spetember 1999, vol. 21, pp. 895–913 (1999)
15. Smith, A., Burrill, J., Gibson, J., Maher, B., Nethercote, N., Yoder, B., Burger, D.C., McKinley, K.S.: Compiling for EDGE architectures. In: International Conference on Code Generation and Optimization, pp. 185–195 (2006)
16. SPEC2000CPU. The standard performance evaluation corporation (SPEC), http://www.spec.org/
17. Taylor, M.B., Agarwal, A.: Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ILP and streams. In: ACM SIGARCH International Symposium on Computer Architecture, pp. 2–13 (2004)
18. Traub, O., Holloway, G., Smith, M.D.: Quality and speed in linear-scan register allocation. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, June 1998, pp. 895–913 (1998)
19. Warter, N.J.: Reverse if-conversion. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 290–299 (1993)
20. Yoder, B., Burrill, J., McDonald, R., Bush, K.B., Coons, K., Gebhart, M., Govindan, S., Maher, B., Nagarajan, R., Robatmili, B., Sankaralingam, K., Sharif, S., Smith, A.: Software infrastructure and tools for the TRIPS prototype. In: Third Annual Workshop on Modeling, Benchmarking and Simulation (2007)

Smashing: Folding Space to Tile through Time

Nissa Osheim, Michelle Mills Strout, Dave Rostron, and Sanjay Rajopadhye

Colorado State University, Fort Collins CO 80523, USA

Abstract. Partial differential equation solvers spend most of their computation time performing nearest neighbor (stencil) computations on grids that model spatial domains. Tiling is an effective performance optimization for improving the data locality and enabling course-grain parallelization for such computations. However, when the domains are periodic, tiling through time is not directly applicable due to wrap-around dependencies. It is possible to tile within the spatial domain, but tiling across time (i.e. time skewing) is not legal since no constant skewing can render all loops fully permutable. We introduce a technique called smashing that maps a periodic domain to computer memory without creating any wrap-around dependencies. For a periodic cylinder domain where time skewing improves performance, the performance of smashing is comparable to another method, circular skewing, which also handles the periodicity of a cylinder. Unlike circular skewing, smashing can remove wrap-around dependencies for an icosahedron model of earth's atmosphere.

1 Introduction

Many computational science applications iterate over a discretized spatial domain to model its change over time or to converge to a steady-state solution for unknowns within the discretized space. Regular computations are those where the discretization of the simulation space is done with a one, two, or three-dimensional grid whose values can be stored in an array of the same dimensionality. This paper focuses on the issues that arise when the discretized domain is periodic. The wrap-around dependencies that result from periodicity make it difficult to use a well known program optimization called time skewing. This paper presents a technique called smashing that folds the data space so that all dependencies in the corresponding iteration space are uniform, thereby enabling time skewing. Although other techniques such as circular skewing enable time skewing for some periodic domains, smashing applies more generally and exhibits comparable overhead.

In a periodic domain, some of the points on the grid boundary are simulation space neighbors to points on a different boundary in the discrete grid. Fig. 1 shows the code and the iteration space for a computation that iterates over a ring. The ring has been unrolled and discretized into a one-dimensional array. Notice the long, or wrap-around, dependencies going from iteration point $(1, 1)$

J.N. Amaral (Ed.): LCPC 2008, LNCS 5335, pp. 80–93, 2008.

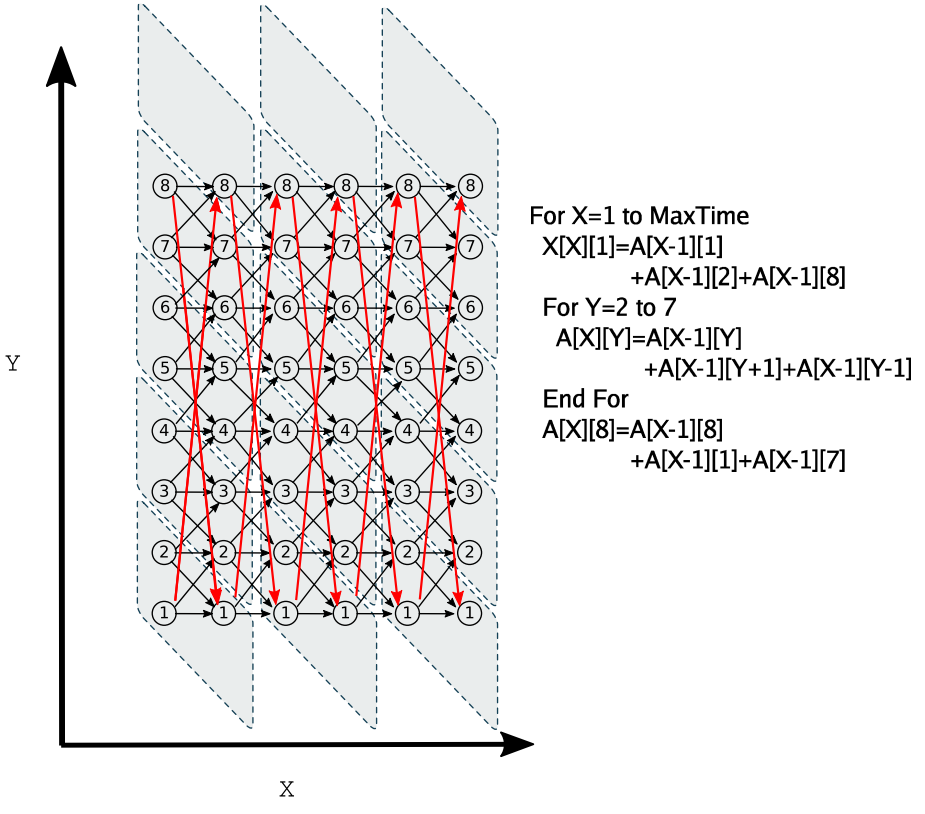


Fig. 1. Iteration space of modeling a ring through time. The ring has been cut and straightened. The vertical axis represents the points on the ring. The horizontal axis represents time. Points are dependent on their neighbors in the previous time step. The dependencies are shown with arrows. Because the top and bottom points are neighbors, there are long dependencies from the top to the bottom and the bottom to the top. The code has not been tiled.

to $(2, 8)$ and $(1, 8)$ to $(2, 1)$. [1] Periodic domains arise from modeling hollow objects or shells. Since they are hollow, shells can be represented in one less dimension. Typically, the object is cut and unrolled to remove one dimension. For example, ring is cut at one point and unrolled to form a line. A line is easy to map into data and iteration domains, but the top and bottom points of the line must still be treated as neighbors in simulation space. A cylinder or torus can be represented with a plane.

Shells that result in periodic domains are frequently used to model physical phenomena such as the earth's atmosphere and surface. For example, the earth's atmosphere has been modeled with an icosahedron [11] and a cubed-sphere [1].

[1] The code in Fig. 1 has been written using single assignment so that the dependencies are clear, but actual implementations only use two one-dimensional arrays.

Fig. 2 shows how an icosahedron is used to discretize one layer of the atmosphere and how it can be unrolled into a set of two-dimensional grids.

Since stencil computations over periodic domains occur in many important applications, there is significant interest in improving their performance through data locality improvements and parallelization. Stencil computations can be modeled as an iteration space where each point in the space represents one iteration of the loop. This space can then be tiled, breaking up the computations in an attempt to group those that use the same data together thereby increase data locality. Tiling as a performance optimization has been studied extensively and has been shown to make code more efficient [6, 12, 15, 16].

Tiling these domains through time is difficult because the traditional strategy of performing a uniform time skewing does not remove the cycle of dependencies between tiles. They have non-uniform dependencies in nearly opposite directions. Fig. 1 shows the example of trying to tile a ring as it is modeled through time. The vertical axis represents the points on the ring. The horizontal axis represents time. Because the top and bottom points are neighbors, there are long dependencies

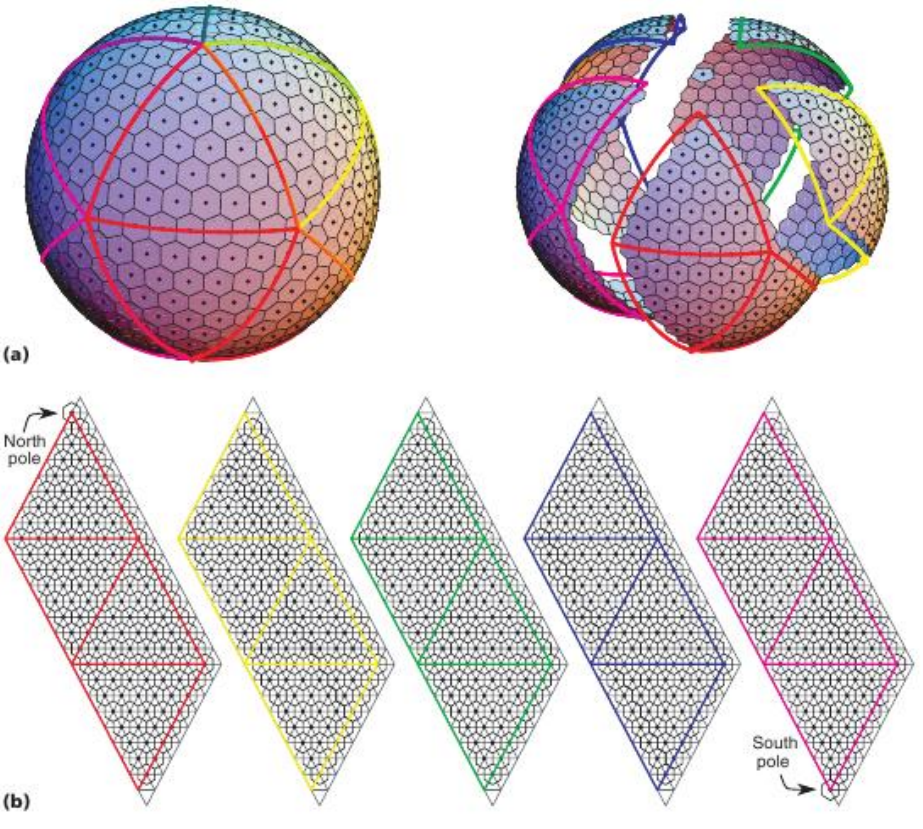
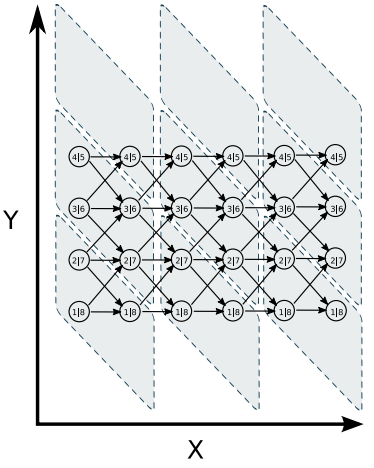


Fig. 2. Icosahedron cut into five parallelograms. Figure courtesy of Randal, Ringer, Heikes, Jones, and Baumgardner [11].




```
For X=1 to MaxTime
  A[X][1][0]=A[X-1][1][0]+A[X-1][2][0]+A[X-1][1][1]
  A[X][1][1]=A[X-1][1][1]+A[X-1][2][1]+A[X-1][1][0]
  For Y=2 to 3
    A[X][Y][0]=A[X-1][Y][0]+A[X-1][Y+1][0]+A[X-1][Y-1][0]
    A[X][Y][1]=A[X-1][Y][1]+A[X-1][Y+1][1]+A[X-1][Y-1][1]
  End For
  A[X][4][0]=A[X-1][4][0]+A[X-1][4][1]+A[X-1][3][0]
  A[X][4][1]=A[X-1][4][1]+A[X-1][4][0]+A[X-1][3][1]
End For
```

Fig. 3. Iteration space that results from the smashed ring data space. Each node corresponds to two points from the original computation. Notice that the tiles have no cyclic dependencies. Although tiling is legal, the displayed code has not been tiled.

from the top to the bottom and the bottom to the top. Any two-dimensional tiling scheme based using only unimodular skewing as a preprocessing step will create tiles with circular dependencies. There is no way to legally schedule tiles with circular dependencies, thus space can not be tiled through time.

In this paper, we resolve these difficulties through a new technique called smashing that removes all the non-uniform neighbor relationships from the periodic domains that result from rings, cylinders, tori, and icosahedra. Essentially, it is a "data allocation" technique rather than a loop/iteration transformation. In effect, we allocate the data by defining a map from the simulation space to memory in such a way that all dependencies, including the periodic ones are strictly uniform. Once the neighbor relationships in the data space are all uniform, the dependencies in the resulting iteration space are also uniform and therefore amenable to unimodular skewing to enabling tiling through time, or time skewing. Fig. 3 shows the iteration space that results from the smashed ring data space. Notice that all of the dependencies are uniform.

Section 2 reviews the motivation for time skewing and previous techniques for enabling time skewing in the presence of periodic domains. Section 3 presents the smashing technique and shows how it works for the ring, cylinder, torus, and icosahedron. Section 4 shows that the overhead of circular skewing and smashing on a cylinder are both reasonable and comparable. Section 5 concludes.

2 Related Work

Related work includes research on applying time skewing, or tiling through time, to stencil computations and techniques that enable tiling through time despite wrap-around dependencies due to periodicity.

Tiling through time to improve data locality in non-periodic, stencil computations has been studied and shown to improve performance in numerous contexts. Wolf and Lam [15] showed how the uni-modular transformations skewing, reversal, and permutation could be applied to perfectly nested stencil loops such as SOR to enable tiling over time. Basetti et al. [3] use the term "temporal blocking" for the way they do multiple time steps by having more than one layer of ghostcells in a structured mesh. Douglas et al. [4] modify the smoother in structured and unstructured multigrid implementations so that pieces of multiple time steps that access similar data locations are performed to improve data locality. Ahmed et al. [2] are able to tile across time for computations with imperfectly nested loops such as those that occur in the Jocobi stencil computation. They do this by embedding all of the iterations in the computation into a single product space and performing tiling within that space. Sellappa and Chatterjee [13] perform a temporal blocking across the time steps in Red-Black Gauss-Seidel. Wonnacott [18] shows how to modify the storage mapping and time skew the computation to ensure scalable locality, where scalable locality indicates that a constant tile size may be selected so that the computation experiences data locality no matter how large the problem size parameters grow.

Some previous research has also looked at tiling across time when there were periodic boundaries in the simulation space.

Jin et al. [7] present at technique called recursive prismatic time skewing. They calculate a reverse skewing factor to deal with periodic boundaries. The reverse skewing factor is used to prevent the computation of boundary iteration points at the start of the spatial domain that need computational results from previous iterations of points later in the spatial domain. The code generated computes these boundary iteration points as soon as the dependencies allow. Their technique is applicable to any (hyper) rectangular domain where the periodic boundaries are each pair of parallel sides in the (hyper) rectangle. Therefore, this technique applies to the ring, cylinder, and torus, but not the cubed sphere or icosahedron.

Periodic domains have also been tiled using rhombus shaped tiles [5, 8, 10]. These tiles overlap at their bases, causing two tiles to compute some duplicate results. Each tile computes all the data it will need for subsequent time steps itself. The iteration space does not need to be skewed, and the overlapping rhombus tiles can start execution at the same time, thus resulting in no start-up cost [10]. These benefits can be enough to overcome the extra time spent recomputing a portion of the iterations. Overlapping tiles can also be used to handle wrap-around dependencies that are introduced due to periodic boundaries [5, 8]. The presented techniques could feasibly be extended to handle the ring, cylinder, and torus domains. More complex periodic boundaries such as the cubed sphere and the icosahedron would require special logic to handle the varying directions of periodicity they exhibit.

Song and Li [14] used the circular skewing technique introduced by Wolfe [17] to make all of the long dependencies due to periodic domains point in the same direction and therefore enable tiling through time. They formalized circular

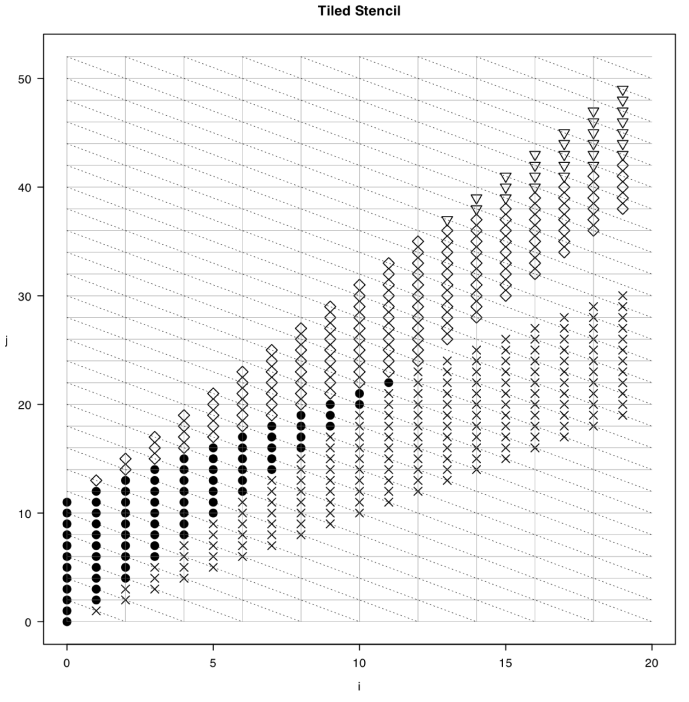


Fig. 4. Example of circular skewing on a ring. The horizontal axis is the time dimension. The vertical axis represents the spatial domain of the ring. Initially, the wrap-around causes non-uniform dependencies between the top and the bottom of this space. The iteration points marked with x are the problem points that do not allow tiling through time. The x points are translated by N (size of the ring) up to the diamonds. At each time step, the number of points that need to be translated increases because more points from the previous time step are translated. The triangles represent the points that need to be translated multiple times. The background grid represents 2x2 tiles.

skewing only in the case where the spatial domain was one-dimensional. Circular skewing plus unimodular skewing enables tiling through time. Fig. 4 shows that a skewed iteration space (lower points) can be modified with circular skewing to ensure that the resulting loop is fully permutable and therefore tileable. It removes the negative non-uniform dependencies and creates positive non-uniform dependencies. With all non-uniform dependencies going in the same direction, it is possible to tile legally. It is possible to extend circular skewing to the cylinder and torus domains, but the technique does not extend to the cubed sphere or icosahedron.

Smashing differs from previous work that enables tiling through time despite periodic dependencies due to the fact that it transforms the data space to prevent such periodic dependencies instead of transforming the iteration space to deal with them after the fact. As such, after smashing, techniques such as overlapping

tiles [5, 8, 10], which enable load-balanced parallelism, are applicable to the resulting iteration space.

3 Smashing

Smashing is a new storage mapping method that prevents wrap-around dependencies due to periodicity, which occurs when modeling hollow objects. Given a hollow object, one dimension needs to be removed to prevent iterating over the empty space inside the object. Normally this dimension is removed by cutting and unrolling. This creates wrap-around or non-uniform neighbors from one side of the cut to the other. Smashing treats the data domain more like the object it represents and smashes or flattens it to remove the extra dimension. This can also be done by proceeding with the unrolling as is typical and then folding the unrolled data space. For example, the unrolled ring creates a line, this line is folded in half to create the smashed ring. Whether the data transformation is done by smashing or folding, the end result is the same data space. It has multiple layers but no wrap-around. In the ring example, the single fold causes two layers. These layers can be stored as separate arrays or as a single two dimensional array where the inner dimension has a size of two.

To explain how this removes the wrap-around dependencies in the iteration space, we use the concept of neighbors in the data space. Neighbors are points that are near each other in simulation space. The stencil defines the neighbors of interest. In the data space a neighboring relationship is uniform if the neighbors are a constant distance from one another. Non-uniform neighbors in the data space cause the non-uniform dependencies in the iteration space. Smashing creates a data space with only uniform neighboring relationships. A data space with uniform neighbors can be mapped to an iteration space, skewed, and tiled using the same methods that work on non-periodic domains. The remaining subsections show the details of how smashing creates a uniform neighbor relation for a ring, torus, and icosahedron.

3.1 Ring

Here we smash the data space of the ring and show how it removes the non-uniform neighbor relations. The data space of the unrolled ring is a one dimensional array ranging from 0 to N-1, where N is the number of discrete points in the ring. We specify the neighbors in this data space as a piecewise affine function. For this example, we assume a stencil that uses the left and right adjacent points.

$$\text{Right}(i) \;=\; \begin{cases} i = N-1 : (0) \\ i < N-1 : (i+1) \end{cases}$$

$$\text{Left}(i) \;=\; \begin{cases} i = 0 : (N-1) \\ i > 0 : (i-1) \end{cases}$$

The above neighbor functions contain non-uniformity. For the $Right()$ neighbor function, when $i = N - 1$, its right neighbor is 0. The distance between them is $N - 1$ and is thus dependent on the number of points in the ring and non-uniform. Likewise, when $i = 0$ its left neighbor is $N - 1$. Smashing folds this data space into a two-dimensional data space with the following transformation:

$$Smash(i) = \begin{cases} i < N/2 : (i, 0) \\ i \geq N/2 : (N - 1 - i, 1) \end{cases}$$

We assume that N is even.

This creates a two dimensional array where the second dimension has two possible values. After this transformation, the neighbor function in the transformed data space is as follows:

$$Right(i, k) = \begin{cases} i = N - 1,\ k = 0 : (i, k + 1) \\ i < N - 1,\ k = 0 : (i + 1, k) \\ i = 0,\ k = 1 \quad\ : (i, k - 1) \\ i > 0,\ k = 1 \quad\ : (i - 1, k) \end{cases}$$

$$Left(i, k) = \begin{cases} i = 0,\ k = 0 \quad\ : (i, k + 1) \\ i > 0,\ k = 0 \quad\ : (i - 1, k) \\ i = N - 1,\ k = 1 : (i, k - 1) \\ i < N - 1,\ k = 1 : (i + 1, k) \end{cases}$$

Notice that all the neighbors are now constant offsets from the data space point (i, k).

3.2 Torus

A torus consists of a cylinder that has been rolled into a three-dimensional ring. The top and bottom of the cylinder are connected, creating an object that looks like a donut. To smash this we set the torus on edge and flatten it resulting in four rectangles set on top of each other. Fig. 5 visualizes this another way. Cut the torus vertically into two even halves. Straighten the two halves into cylinders. Cut each cylinder along the long edge into two even halves, and straighten these four halves into rectangles. The folding method results in the same four-layered rectangle.

The unrolled torus is a single rectangle with wrap-around from the north to south and from the east to west. The folding method folds this plane in half horizontally and again vertically resulting in four layered rectangles of one-fourth the size of the original. The points that were non-uniform neighbors on the edge of the unrolled rectangle before folding are now within three layers of each other on the edge of the folded rectangle.

The neighborhood of the unrolled torus is described with the following piecewise affine functions:

$$North(i, j) = \begin{cases} j = M - 1 : (i, 0) \\ j < M - 1 : (i, j + 1) \end{cases}$$

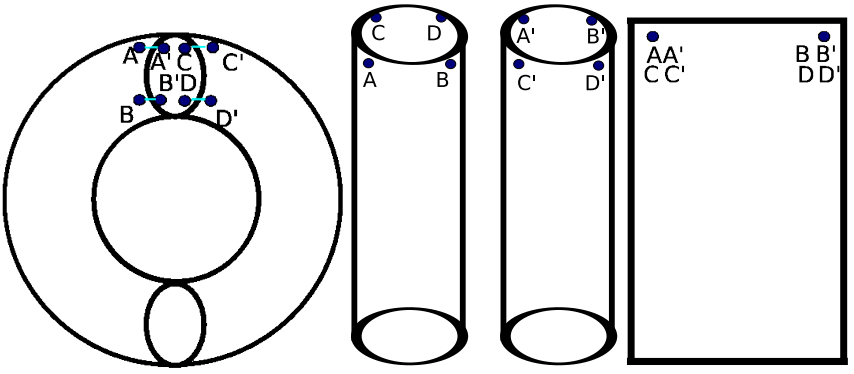


Fig. 5. Smashing a torus by cutting it into two cylinders

$$South(i,j) = \begin{cases} j = 0 : (i, M-1) \\ j > 0 : (i, j-1) \end{cases}$$

$$East(i,j) = \begin{cases} i = N-1 : (0, j) \\ i < N-1 : (i+1, j) \end{cases}$$

$$West(i,j) = \begin{cases} i = 0 : (N-1, j) \\ i > 0 : (i-1, j) \end{cases}$$

where N is the size of the domain in the i dimension, and M is the size in the j dimension.

Unrolling introduces non-uniform neighbors as it did in the ring example. The north neighbors of the points (i,M-1) are the points (i,0). These points are a distance of M-1 from each other. The points along the other edges also have non-uniform neighbors. Smashing folds the two-dimensional, unrolled data space into a three-dimensional data space with the following transformation:

$$Smash(i,j) = \begin{cases} i < N/2,\ j < M/2 : (i, j, 0) \\ i \geq N/2,\ j < M/2 : (N-1-i, j, 1) \\ i < N/2,\ j \geq M/2 : (i, M-1-j, 2) \\ i \geq N/2,\ j \geq M/2 : (N-1-i, M-1-j, 3) \end{cases}$$

N and M must be even.

The new neighbor functions are:

$$North(i,j,k) = \begin{cases} k < 2,\ j = M/2-1 : (i, j, k+2) \\ k \geq 2,\ j = 0 \quad\quad\ : (i, j, k-2) \\ k < 2,\ j < M/2-1 : (i, j+1, k) \\ k \geq 2,\ J > 0 \quad\quad\ : (i, j-1, k) \end{cases}$$

$$South(i,j,k) = \begin{cases} k<2,\ j=0 & : (i,j,k+2) \\ k\geq 2,\ j=M/2-1 & : (i,j,k-2) \\ k<2,\ j>0) & : (i,j-1,k) \\ k\geq 2,\ J<M/2-1 & : (i,j+1,k) \end{cases}$$

$$East(i,j,k) = \begin{cases} k=0\ or\ 2,\ i=N/2-1 & : (i,j,k+1) \\ k=1\ or\ 3,\ i=0 & : (i,j,k-1) \\ k=0\ or\ 2,\ i<N/2-1 & : (i,j+1,k) \\ k=1\ or\ 3,\ i>0 & : (i,j-1,k) \end{cases}$$

$$West(i,j,k) = \begin{cases} k=0\ or\ 2,\ i=0 & : (i,j,k+1) \\ k=1\ or\ 3,\ i=N/2-1 & : (i,j,k-1) \\ k=0\ or\ 2,\ i>0 & : (i,j-1,k) \\ k=1\ or\ 3,\ i<N/2-1 & : (i,j+1,k) \end{cases}$$

The smashed torus has uniform neighbor relations, and the resulting iteration space can be tiled without dealing with wrap-around dependencies.

3.3 Icosahedron Representation of the Earth

Modeling a sphere is difficult since a curve is hard to discretize. Randall et al. [11] use a geodesic grid to model the earth's atmosphere. They start with an icosahedron and repeatedly bisect the faces to create increasingly finer grids. We smash the icosahedron and remove all non-uniform neighbors. To do this, we start with the unrolled icosahedron specified by Randall et al. [11], which is made up of 20 triangles, or alternatively, of 10 rhombi (diamonds) of two triangles each, or even five parallelograms of four triangles each (see Fig. 2). The first and second parallelograms are reconnected where their third and second triangles, respectively, were connected in the original icosahedron (see Figures 7 and 8). The third, fourth, and fifth parallelograms are similarly reconnected. This creates a single polygon, but with non-uniform neighbors.

The non-uniform neighbors in the unrolled icosahedron can be made uniform by folding (see Figures in Appendix A). We make ten vertical folds along the ten main (i.e., longer) diagonals of the ten rhombi. The folds are made "accordion style" so that we end up with ten long parallelograms laid out on top of each other in ten layers (actually there are eleven layers – the first triangle, nine parallelograms, and a last triangle, but the triangles can be "joined" together to form a single parallelogram). The points that were non-uniform neighbors are now at the same point of the new equilateral, but on a different layer from their neighbor. To help visualize this, the reader is encouraged to cut out the Figures in Appendix A and do some origami. The new data space can now be skewed and tiled through time because all neighbors are an uniform distance from each other.

4 Experimental Results

To test the viability of smashing, we use a cylindrical domain and perform a nearest neighbor stencil computation over time. Smashing enables tiling through time, but introduces overhead in terms of accessing the data space. Our experiments show that the overhead for smashing on a cylinder is comparable with the overhead of circular skewing and that both result in an overall performance benefit because they enable tiling over time.

In the stencil computation test, the neighbors are the north, south, east, and west points, and the calculations (averaging the neighbors) are done using the values of the neighbors from the previous time step. We iterate over this domain with four separate methods, no tiling, tiling in two dimensions, tiling through time using circular loop skewing, tiling through time using smashing. The results for no tiling, circular loop skewing, and smashing are shown in fig. 6. Tiling in two dimensions did not improve the results over not tiling for this domain, therefore we do not include them in fig. 6.

We tried various tile sizes to find a reasonable tile size for each method. We then compare the methods using the tile size that is the best we found for each method. Our goal was not do determine what the optimal tile size is for the selected problem sizes, but instead to show that with a good tiling, the overhead of smashing competes with the less general circular skewing. In the time dimension the tiles sizes ranged from 2 to 64. In the space dimension we have square tiles ranging from 2 to 1024 on a side. We also tried not tiling the inner loop. Kamil et al. [9] recommend not tiling the inner loop because of the

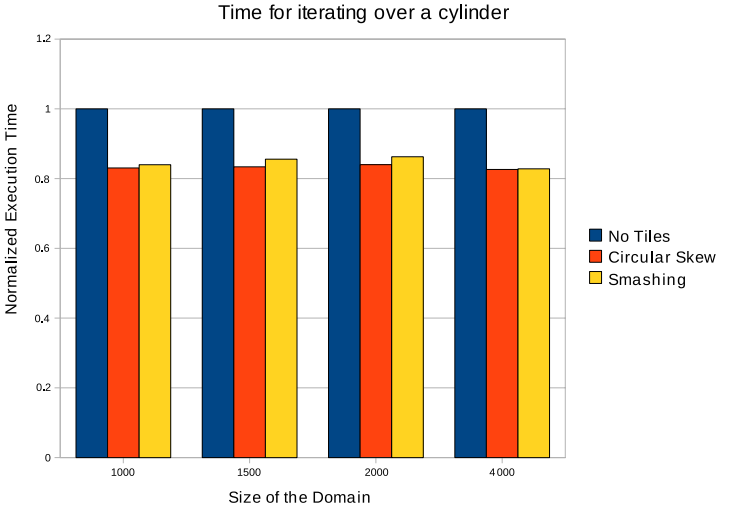


Fig. 6. Graph of the execution times for the cylinder using no tiling, circular skewing, and smashing. The iteration spaces are all three-dimensional. The graph has been normalized to the execution time of the no tiling method.

Table 1. Tile Sizes

Method	1000-2000	4000
Circular Skew	16x2x0	8x1x0
Smashing	8x2x0	64x64x64

prefetcher. We tried multiple tile sizes and not tiling the inner loop was the best strategy for all the methods except smashing on the 4000 size domain.

For our final results we ran the tile sizes that performed well for each method and domain twenty times and took an average to produce Fig. 6. With this wide variety of tile sizes, we can be reasonably sure that our choice of tile size is not distorting the results. Table 1 shows the tiles size used for each method and domain. The first dimension is time and the last is the inner loop. When the size of the last dimension is zero, it means we did not tile the inner loop. The columns show the problem size. We used the same tile size for problem the 1000, 1500, and 2000 problem sizes. They are in one column (1000-2000).

Tiling through time with both circular loop skewing and smashing shows a speedup over not tiling through time. The times are comparable between circular loop skewing and smashing. These results show that smashing does not introduce more overhead than less general techniques for enabling time skewing on computations with periodic domains.

5 Conclusion

We have proposed, analyzed, and implemented in a number of examples a data mapping technique called smashing. Smashing is an enabling transformation that allows a well known and useful tiling transformation—time skewing—to be applied to stencil computations for domains with periodic boundary conditions. Unlike previous techniques that address such periodic domains, smashing is not an iteration space transformation of a given piece of code. Rather, it can be viewed as a piecewise affine memory map—allocation of discretization points of the physical domain being modeled to memory locations that are viewed as a contiguous multidimensional array. Our main result is to show that for many practical cases smashing preserves the property that the neighbors of any point in the physical domain remain neighbors in the multidimensional memory. As a result, any iterative stencil computation on this data space automatically enjoys the uniform dependence properties that allow direct implementation of time skewing.

Smashing is easy to describe and implement, and as we have shown, provides performance comparable to circular skewing. It removes the wrap-around from the data space before it is converted to the iteration space and works for domains such as the cube and icosahedron that earlier techniques like circular skewing cannot handle.

An open question that we are investigating is a proof of generality: under what conditions can an arbitrary "hollow" physical object be smashed down to a contiguous multidimensional array while retaining the neighborhood property.

In addition we are developing tools to automatically generate code that incorporates the smashing transformation from high-level stencil specifications for hollow objects.

References

1. Adcroft, A., Campin, J.-M., Hill, C., Marshall, J.: Implementation of an atmosphere-ocean general circulation model on the expanded spherical cube. Monthly Weather Review, 2845–2863 (2004)
2. Ahmed, N., Mateev, N., Pingali, K.: Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In: Conference Proceedings of the 2000 International Conference on Supercomputing, Santa Fe, New Mexico, May 2000, pp. 141–152 (2000)
3. Bassetti, F., Davis, K., Quinlan, D.: Optimizing transformations of stencil operations for parallel object-oriented scientific frameworks on cache-based architectures. In: Caromel, D., Oldehoeft, R.R., Tholburn, M. (eds.) ISCOPE 1998. LNCS, vol. 1505, pp. 107–118. Springer, Heidelberg (1998)
4. Douglas, C.C., Hu, J., Kowarschik, M., Rüde, U., Weiß, C.: Cache Optimization for Structured and Unstructured Grid Multigrid. Electronic Transaction on Numerical Analysis, 21–40 (February 2000)
5. Frigo, M., Strumpen, V.: Cache oblivious stencil computations. In: Proceedings of the 19th Annual International Conference on Supercomputing (ICS), pp. 361–366. ACM, New York (2005)
6. Irigoin, F., Triolet, R.: Supernode partitioning. In: Proceedings of the 15th Annual ACM SIGPLAN Symposium on Priniciples of Programming Languages, pp. 319–329 (1988)
7. Jin, G., Mellor-Crummey, J., Fowler, R.: Increasing temporal locality with skewing and recursive blocking. In: High Performance Networking and Computing (SC), Denver, Colorodo, November 2001. ACM Press and IEEE Computer Society Press (2001)
8. Kamil, S., Datta, K., Williams, S., Oliker, L., Shalf, J., Yelick, K.: Implicit and explict optimizations for stencil computations. In: Memory Systems Performance and Correctness (2006)
9. Kamil, S., Husbands, P., Oliker, L., Shalf, J., Yelick, K.: Impact of modern memory subsystems on cache optimizations for stencil computations. In: Proceedings of the Workshop on Memory System Performance, pp. 36–43. ACM Press, New York (2005)
10. Krishnamoorthy, S., Baskaran, M., Bondhugula, U., Ramanujam, J., Rountev, A.: Effective automatic parallelization of stencil computations. In: Proceedings of Programming Languages Design and Implementation (PLDI), pp. 235–244. ACM, New York (2007)
11. Randall, D.A., Ringler, T.D., Heikes, R.P., Jones, P., Baumgardner, J.: Climate modeling with spherical geodesic grids. Computing in Science and Engineering 4(5), 32–41 (2002)
12. Schreiber, R., Dongarra, J.J.: Automatic blocking of nested loops. Technical Report UT-CS-90-108, Department of Computer Science, University of Tennessee (1990)
13. Sellappa, S., Chatterjee, S.: Cache-efficient multigrid algorithms. In: Proceedings of the 2001 International Conference on Computational Science, San Francisco, CA, USA, May 28-30, 2001. LNCS. Springer, Heidelberg (2001)

14. Song, Y., Li, Z.: New tiling techniques to improve cache temporal locality. ACM SIGPLAN Notices (PLDI) 34(5), 215–228 (1999)
15. Wolf, M.E., Lam, M.S.: A data locality optimizing algorithm. In: Programming Language Design and Implementation (1991)
16. Wolfe, M.J.: Iteration space tiling for memory hierachies. In: Proc. of the 3rd SIAM Conf. on Parallel Processing for Scientific Computing, pp. 357–361 (1987)
17. Wolfe, M.J.: High Performance Compilers for Parallel Computing. Addison-Wesley, Reading (1996)
18. Wonnacott, D.: Achieving scalable locality with time skewing. International Journal of Parallel Programming 30(3), 181–221 (2002)

Appendix A

Models of the icosahedron and the smashed icosahedron

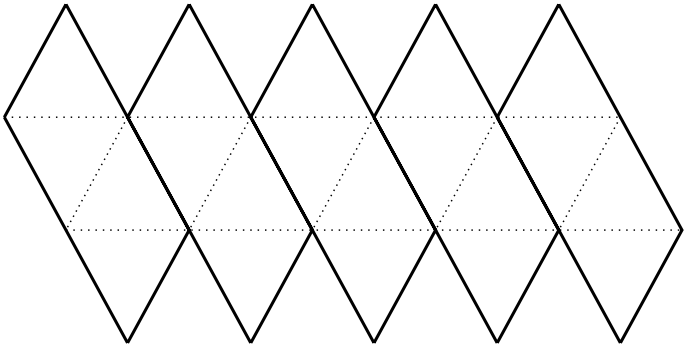


Fig. 7. Model of the icosahedron. Cut out the figure and fold in along the dotted lines. The edges will meet to form a icosahedron. Where the edges meet is where the non-uniform neighbors are when it is flattened out.

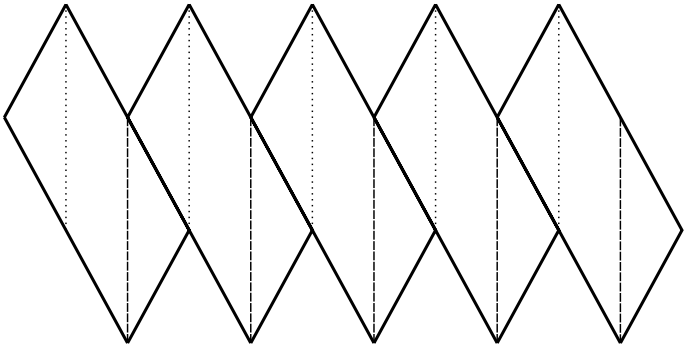


Fig. 8. Model of the smashed icosahedron. Cut out the figure, fold in along the dotted lines and out along the dashed lines. The result are ten layered parallelograms. The points that were non-uniform neighbors lie on top of each other.

Identification of Heap–Carried Data Dependence Via Explicit Store Heap Models*

Mark Marron[1], Darko Stefanovic[1], Deepak Kapur[1], and Manuel Hermenegildo[1,2]

[1] University of New Mexico
{marron,darko,kapur}@cs.unm.edu
[2] Technical University of Madrid and IMDEA-Software
herme@fi.upm.es

Abstract. Dependence information between program values is extensively used in many program optimization techniques. The ability to identify statements, calls and loop iterations that do not depend on each other enables many transformations which increase the instruction and thread-level parallelism in a program. When program variables contain complex data structures including arrays, records, and recursive data structures, the ability to precisely model data dependence based on heap structure remains a challenging problem.

This paper presents a technique for precisely tracking heap based data dependence in non-trivial Java programs via static analysis. Using an abstract interpretation framework, the approach extends a shape analysis technique based on an existing graph model of heaps, by integrating read/write history information and intelligent memoization. The method has been implemented and its effectiveness and utility are demonstrated by computing detailed dependence information for two benchmarks (Em3d and BH from the JOlden suite) and using this information to parallelize the benchmarks.

1 Introduction

The concept of data dependence between program statements is a fundamental tool for the reordering of program statements and the determination of invariant values in basic blocks, loops, or methods. Knowledge of data dependence allows the introduction of instruction–level parallelism and thread–level parallelism (both in loops and method invocations). In past work effective techniques for computing data dependence between scalar variables have been developed. However, the extension of this work to tracking memory–carried data dependence has been much less successful, in large part due to the lack of suitable heap analysis techniques to support them.

Previous work focused broadly on two approaches for identifying possible heap–carried data dependence, shape or points-to analysis as a proxy for data dependence [2, 4, 7, 14] wherein the identification of various acyclic structures and/or access path information is used to infer which expressions cannot access the same portion of the heap, and the explicit tracking of read/written locations [3, 8, 9] which model the set of locations that may be read/written at each program point. This work introduced several

* This work is supported in part by NSF grant 0540600.

J.N. Amaral (Ed.): LCPC 2008, LNCS 5335, pp. 94–108, 2008.

fundamental concepts involved in modeling heap carried data dependence. However experimental work with these approaches was limited to small numbers of micro-benchmarks or used coarse points-to style analysis.

This paper builds on the basic concepts developed in earlier work and makes several contributions which are critical to analyzing non-trivial programs. The first is a novel method for tracking read/write locations during the analysis. The approach presented in this paper only tracks a two program locations per object field (one read location and one write location) instead of a set of all possible read locations and a set of all possible write locations per field. This is sufficient to identify the most recent program point where each memory location may be used/modified while avoiding the additional space usage and computational cost of tracking a set of program locations per object field. The next contribution is a method to efficiently track read/write information through method boundaries, in particular how to ensure that the addition of *use–mod* information does not have a serious impact on the memoization of method body analysis results, which is critical to applying the technique to realistic programs.

Our analysis technique uses an explicit store model for the heap objects which allows us to easily track the identity of objects between program statements. This differs from some recent work on shape analysis, which uses logical models with implicit store representations [5, 15] that cannot be efficiently extended to track the properties of arbitrary heap locations. It also differs from approaches based on separation logic which restrict the program to regular recursive structures and limited sharing of objects on the heap in order to ensure termination [1, 6]. These features preclude the use of these approaches on many realistic application programs including the `em3d` and `bh` benchmarks, which we analyze as detailed case studies here.

2 Running Examples

We use examples in this paper to illustrate the various aspects of the analysis technique. The first is a small fragment created solely to illustrate the basics of the analysis. The second is a routine taken from `em3d`, one of the JOlden [10, 13] benchmarks.

The first example 1 creates 2 `Data` objects, each of which has a single integer field `val`, and puts them in a `Pair` object. If the conditional holds the `first` element of the pair is modified and then the swap method is called to interchange the `first` and `second` elements of the pair. This example is simple but relevant since in order to determine that the asserted property always holds the analysis needs to be able to track how pointer stores affect reachability relations in the heap, to identify where each heap location may be written, and do so across method invocations.

The second program fragment is a method taken from the `em3d` benchmark. This program builds a bipartite heap structure. Each call to `computeNewValue` takes a `ENode` object from one side of the bipartite graph and updates the `value` field of this node based on the value fields of `ENode` objects on the opposite side of the bipartite graph. This example demonstrates the importance of precisely resolving the heap structure so the dependence analysis can determine that the set of heap location where the `value` field is written is distinct from the locations that are read.

```
m1  void main() {
m2     Pair p = new Pair(new Data(5), new Data(10));
m3     if(*)
m4       p.first.val = 0;
m5     swap(p);
m6     assert(p.first.val != 0);
m7  }

s1  void swap(Pair p) {
s2     Data temp = p.first;
s3     p.first = p.second;
s4     p.second = temp;
s5  }
```

Fig. 1. Conditional Modify and Swap

```
c1  static void computeNewValue(ENode n) {
c2    for(int i = 0; i < n.fromCount; i++)
c3      n.value -= n.coeffs[i] * n.fromN[i].value;
c4  }
```

Fig. 2. `Compute` (From `em3d`)

3 Abstract Heap Domain

The underlying abstract heap domain that we extend is a graph in which each node represents a region of the heap (a set of objects or data structures) or a variable and each edge represents a set of pointers or a variable target. The nodes and edges are augmented with additional instrumentation predicates.

Types. Since each node in the graph represents a region of the heap (which may contain objects of many types) we use a set of type names for each node in the heap graph which contains the type of any object that may be in the region of the heap that is abstracted by the given node.

Linearity. To model the number of objects abstracted by a given node (or pointers by an edge) we use a *linearity* property which has 2 possible values 1, which indicates that the node (edge) concretizes to either 0 or 1 objects (pointers) and the value ω, which indicates that the node (edge) concretizes to any number of objects (pointers) in the range $[0,\infty)$.

Abstract Layout. To track the connectivity and shape of the region a node abstracts, the analysis uses *abstract layout* predicates *Singleton*, *List*, *Tree*, *MultiPath*, or *Cycle*. The *Singleton* predicate states that there are no pointers between any of the objects represented by an abstract node. The *List* predicate states that each object has at most one pointer to another object in the region. The other predicates correspond to the standard definitions for Trees, Dags, and Cycles in the literature.

Interference. The heap model uses two properties to track the potential that multiple pointers or variables can reach the same memory location in the region that a particular node represents. In this work the examples only require one of these properties (*interference*) so we omit the discussion of the other property (*connectivity*) and refer the interested reader to [12] for a more detailed description.

Each edge abstracts a set of pointers in the concrete program. The *interfere* property has three possible values, to track that some of the pointers may alias (*ap*), that none of the pointers alias but they may point into the same data structure (thus can interfere, *ip*), or that each of the pointers refers to a unique and disjoint data structure in the node that the edge ends at (they are disjoint and non-interfering, *np*).

Heap Representation. We represent abstract heaps pictorially as labeled, directed multigraphs. The variable nodes are labeled with the variable that they represent. The nodes representing the regions are represented as a record `[type, linearity, layout]` that tracks the instrumentation predicates.

The edges (which represent sets of pointers) in the figures are represented as records `[offset, linearity, interfere]`. The *offset* component indicates the offsets (labels) of the references that are abstracted by the edge. These labels may be any of the field identifiers that are used in the program or the special label, *?*, which is the label given to the summary field representing all the elements in a collection object `Vector`, `List`, or an array.

To simplify the figures we omit entries in the labels when they are the default domain value. The default values for the nodes are *layout* = *(S)ingleton* and *linearity* = 1. The default edge values are *linearity* = 1 and *interfere* = *np*. The variable edges always represent single references and the label is always implicitly the variable name.

3.1 Heap Structure Examples

Pair Example. Figure 3 shows the heap model (without any read/write information) that is computed as the result of executing the pair constructor in the first example program. The variable `p` points to a single object of type `Pair` (the *linearity* is 1 and the shape in *Singleton*, as described above this default information is omitted from the figure). The node representing the `Pair` object has 2 outgoing edges representing the two pointers stored in the `first` and `second` fields. The analysis determines that these edges each represent a single pointer (and since any edge representing a single pointer cannot have any interference the *interfere* property is *np*). Again the default

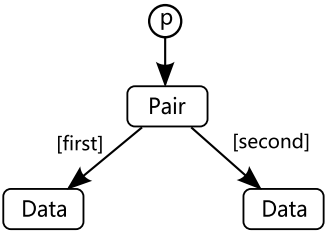


Fig. 3. `Pair` Allocation, Structure Only

properties of linearity 1 and non-interference are omitted from the figure. Finally, the model shows that the `first` and `second` pointers each refer to a single `Data` object.

Em3d Example. The state of the heap at the entry to the `computeNewValue` method in the program `em3d` is shown in Figure 4 (again without any read/write information). The `em3d` program computes electro-magnetic field values in a 3–dimensional space by constructing a list of `ENode` objects, each representing an electric field value and a second list of `ENode` objects, which represent a magnetic field values. To compute how the electric/magnetic field value for a given `ENode` object is updated at each step the `computeNewValue` method uses an array of `ENode` objects from the opposite field and performs a convolution of these field values and a scaling vector, updating the current field value with the result.

Figure 4 shows the heap structure computed for the `computeNewValue` method. We have placed dashed lines around the structures that represent the magnetic field (in blue if color is available) and the electric field (in green). Variable `g` points to a single object of type `BiGrph`, which is the data structure that encapsulates all the objects of interest. The `BiGrph` object has 2 fields, the `hNodes` field pointing to a linked list of `ENode` objects that make up the magnetic field and, the `eNodes` field pointing to a linked list of `ENode` objects that make up the electric field.

Looking at the structures in the magnetic field we see the edge labeled *[?, ω]* which represents all the pointers stored in the linked list. Since the linearity is ω we know the edge may represent multiple pointers but each of these pointers must point to a unique `ENode` object (the default interference value of non-interfering *np* is omitted). The figure also shows that the magnetic field is represented by many `ENode` objects (the node labeled *[ENode, ω]*) each of which has a pointer to a unique array of `floats`

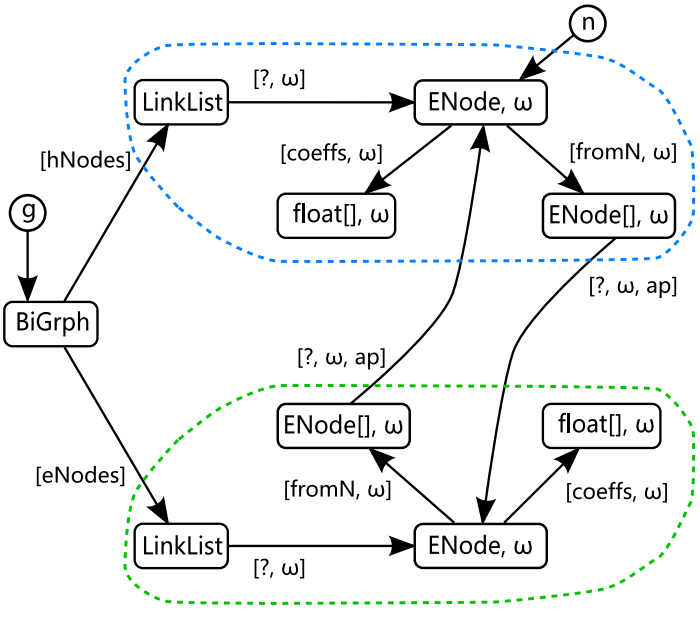


Fig. 4. `computeNewValue`, Structure Only

(the edge labeled `coeffs`) and an array of `ENode` objects (the edge labeled `fromN`) which are used as the set of nodes from the opposite field. The edge that represents the pointers stored in this array is labeled *[?, ω, ap]*, which indicates that it represents all the pointers stored in the array and, that the pointers it represents may alias (*ap*).

4 Data Dependence Extensions

To track the read/write histories of objects on the heap we extend the model presented in Section 3 with information to track the identity of the objects represented by a given node, and for each field in the object we track the *most recent* program location (statement or control flow structure) where a read/write of that field *may* have occurred.

In order to ensure that the initial shape analysis when augmented with the read/write domain remains efficient it is critical to minimize the amount of additional information that is added to the heap model. The key observation is that for most optimization applications the shape analysis only needs to provide precise information about the *most recent* program location at which each field *may* have been read or written. Thus, the analysis does not need to track every possible program location where a field may have been read/written, and this significantly reduces the computational requirements.

4.1 Intermediate Representation

Before we introduce the domain extensions we need to specify how program locations are represented. To simplify the analysis the Java programs are transformed into a structured mid-level intermediate language (called MIL). The partial grammar below provides a sample of the language constructs in the intermediate representation.

$$
\begin{array}{lll}
atom & ::= & var \mid literal \\
expr & ::= & atom \mid atom + atom \mid \text{new } type(atom, \ldots, atom) \mid var.f \\
 & & \mid var.m(atom, \ldots, atom) \mid var \text{ instanceof } type \mid \ldots \\
stmt & ::= & var\text{=}expr \mid var.f\text{=}atom \mid \text{break} \mid \ldots \\
contol & ::= & \text{if}(atom)\ block \text{ else } block \mid \text{while}(atom)\ block \mid \ldots \\
block & ::= & (stmt \mid control)*
\end{array}
$$

The language has method invocations, conditional constructs (`if`, `switch`), exception handling (`try-throw-catch`) and looping statements (`for`, `do`, `while`). The state modification and expressions cover the standard range of program operations (load, store and assign along with logical, arithmetic and comparison operators). We associate with each statement and each control flow structure a program location ℓ.

4.2 Extended Domain

Read–Write Locations. Each node may represent a number of objects of different types $(\tau_1 \ldots \tau_m)$ and each type may have many fields $(f^1_{\tau_i} \ldots f^n_{\tau_i})$. For each of these fields we keep two program locations (ℓ), the last time the field *may* have been read (ℓ_r) and the last time the field *may* have been written (ℓ_w).

Node Identity. In order to efficiently analyze method invocations we memoize the input and return abstract states and reuse them as possible. In order to prevent spurious

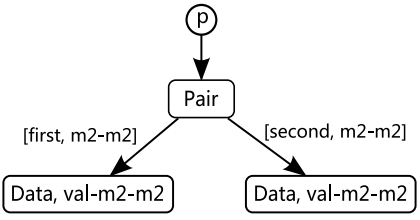


Fig. 5. `Pair` with *use-mod*

inequalities between the read/write program locations (that refer to the locations in the caller scope) in the memoized models we replace them with a generic *modified outside* value. To allow us to match the identities of the objects in the input state with their position in the output state we add a unique identity tag (a value in $\mathbb{N}$) to each node that is passed into a method call.

In our extended domain each node in the heap is now represented as a tuple `[type, linearity, layout, scalar-fields, identity]`. The entries `type`, `layout` and `count` are as described in Section 3. The `scalar-fields` entry is a list of `field-readloc-writeloc` entries, one for each scalar field, where `readloc` and `writeloc` are either a program location ℓ or the special entry 0 (*modified outside*). The `identity` entry is a *set* of identity tags or is omitted entirely if the node does not have a identity tag associated with it (or for clarity if it is not relevant to the example).

To track the read write information for the pointer fields we extend each edge label to `[offset, linearity, interfere, readloc-writeloc]` where `readloc` and `writeloc` are defined the same as for the scalar fields in the nodes. Again, for clarity, we omit `readloc-writeloc` information if it is irrelevant to the example.

Figure 5 shows the model that is computed as the result of executing the pair constructor in the first example program. The pair is marked as having read and written the two pointer fields at initialization (the `m2-m2` entries on the `first` and `second` edges) and the identity tag is omitted (since this object was allocated in the current scope). The two `Data` objects which had their *val* fields initialized at program location *m2* have the entry `m2-m2` in their *scalar-fields* read/write entry.

4.3 Local Data Dependence

Now that we have extended the model with the required instrumentation properties we can define a set of dataflow operations to model the effects of program operations on the read/write information. The changes for load and store operations are simple, only requiring an update of the last read/write value for the target object to the current program location, thus we omit a detailed description of these operations.

Data Flow Domain. If $\mathscr{G}$ is the set of all possible heap graphs and $\mathscr{B} = \wp(var) \times \wp(var)$ (a simple domain to track which variables *must* be *true*, the first element of the pair, and which *must* be false, the second element) then our abstract domain is $\mathscr{D} = \wp(\mathscr{G} \times \mathscr{B})$.

Given an element in the abstract domain, $\sigma \in \mathscr{D}$, we assume the abstract semantics are defined for expressions and statements. Thus, given an expression e, the abstract semantics of this expression on the abstract state σ are given by $\mathscr{S}[\![e]\!]\sigma$ and similarly for statement s, the abstract semantics are given by $\mathscr{S}[\![s]\!]\sigma$. Using the $\mathscr{B}$ component of the domain and a boolean condition b, we can filter an abstract state $\sigma = \{\theta_1, \ldots, \theta_k\}$ into two new abstract states $\sigma_{true} = \mathscr{S}[\![b]\!]_{true}(\sigma) = \{\, \theta_i \mid \text{b } \textit{may} \text{ be true in } \theta_i\}$ and $\sigma_{false} = \mathscr{S}[\![b]\!]_{false}(\sigma) = \{\, \theta_i \mid \text{b } \textit{may} \text{ be false in } \theta_i\}$.

Abstract Conditional Semantics. Using the above definitions we can write the standard definition for the `if` statement, $\mathscr{S}[\![if(b)\ block_t\ else\ block_f]\!]\sigma = \mathscr{S}[\![block_t]\!](\sigma_{true}) \cup \mathscr{S}[\![block_f]\!](\sigma_{false})$. However, using this definition of the semantics can result in exponential growth in the number of states that the analysis must deal with (since for most cases at the union of the abstract states that result from analyzing *true* and *false* branches will have many models that are identical except for a few *readloc–writeloc* entries).

To avoid this we replace all the *readloc–writeloc* entries that refer to program locations in the *true* or *false* branches of the conditional with the program location of the conditional before the union operation. Thus any differences that are solely due to *readloc–writeloc* entries are removed and exponential growth is avoided. Given $\sigma = \{\theta_1, \ldots, \theta_k\}$ and a *block* which contains statements/control structures at program locations $pl = \{\nu_1, \ldots, \nu_i\}$, we define the operator $\clubsuit(\sigma, block, \mu) = \left\{ \theta_i|_{pl}^{\mu} \;\middle|\; \theta_i \in \sigma \right\}$, which performs the required replacements in the heap graph models. With this definition the improved semantics for the conditional operation (at program location κ) are:

$$\mathscr{S}[\![if(b)\ block_t\ else\ block_f]\!]\sigma = \clubsuit(\mathscr{S}[\![block_t]\!](\sigma_{true}), block_t, \kappa) \cup \clubsuit(\mathscr{S}[\![block_f]\!](\sigma_{false}), block_f, \kappa)$$

Disjunctive Domain. To speed program analysis we employ a *partially disjunctive domain* [11] which we use to discard elements in the abstract states (θ_i) that contain redundant read/write information. This is done by defining an order on the program locations based on their control–flow order. In general this order is not total (e.g. statement locations in the *true* and *false* branches of an `if` statement). However, our replacement of locations inside nested control–flow structures with the program location of the structure that contains them ensures that we can always compare the program locations that appear in the *readloc–writeloc* entries.

Analyze Conditional Example. Figure 6(a) is the abstract heap that approximates the state of the program after the *true* branch ($\mathscr{S}[\![block_t]\!](\sigma_{true})$), where the first element of the pair had the `val` field written. In the node that represents the `Data` object that was written we updated the *writeloc* entry to program location *m4* (where the write occurred, marked in red if color is available). Figure 6(b) shows the result of $\clubsuit(\mathscr{S}[\![block_t]\!](\sigma_{true}), block_t, m3)$, where we replaced the *readloc–writeloc* locations that appear in the *true* branch with program location of the `if` statement (program location *m3*, shown in blue).

Figure 6(c) shows the abstract heap from the *false* branch where no write occurred ($\mathscr{S}[\![block_f]\!](\sigma_{false})$). The most recent *mod* location is unchanged (program location *m2*, where the object was initialized) in $\clubsuit(\mathscr{S}[\![block_f]\!](\sigma_{false}), block_f, m3)$ since program location *m2* is not nested in the conditional.

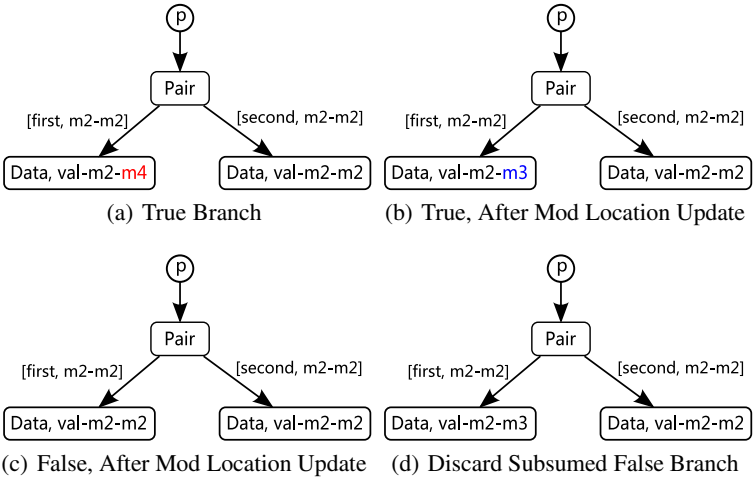


Fig. 6. Updating Read/Write Locations At Control Flow Join

Given our order relation on the *use-mod* sites we can simplify the models resulting from the *true* and *false* branches into a single model shown in Figure 6(d). Intuitively the *may use-mod* information from the *true* branch indicates that the memory location at `p.first.val` *may* have been written at location *m3* (the `if` statement) or at some previous point in the program, while the result of the *false* branch indicates that the memory location at `p.first.val` *may* have been written at location *m2*. Since the possibility that the object may be written at or before program location *m2* is implied by the statement that the object *may* be written at or before program location *m3* we can safely discard the model from the *false* branch.

Abstract Loop Semantics. The semantics of a looping statement `while` at program location κ can be expressed in terms of accumulating all possible exit states. To do this we define the state of the heap at the loop test for the i^{th} iteration of the loop as:

$$\sigma_i = \begin{cases} \sigma & \text{if } i = 0 \\ \mathscr{S}[\![block]\!](\mathscr{S}[\![b]\!]_{true}(\sigma_{i-1})) & \text{otherwise} \end{cases}$$

Then we can define the semantics of the loop analysis as the union of all the possible exits from the loop with the read/write program locations that occur within the loop body replaced by the program location of the loop (κ). Formally:

$$\mathscr{S}[\![while(b)\ block]\!]\sigma = \bigcup \{ \clubsuit(\mathscr{S}[\![b]\!]_{false}(\sigma_i), block, \kappa) \mid i \in \mathbb{N} \}$$

4.4 Interprocedural Data Dependence

In order to efficiently handle large programs we memoize results of analyzing each method. At method call sites, if we were to naively compare the memoized heap models with the current call state the method specific *readloc–writeloc* entries we embed in the model would create many spurious inequalities. As an example consider the `swap`

function from our running example. The `swap` method could be called from multiple locations in a program and at each of these call sites the `Pair` object may have a different *readloc–writeloc* entries for the `first` and `second` fields. If comparision is done in a naive manner these differences will result in spurious mismatches with memoized analysis values, forcing the method to be re-analyzed for each call.

To avoid this problem we anonymize the *readloc–writeloc* locations before attempting to find a match in the memo table. However, when doing this anonymization we need to ensure that we can figure out which locations in the result heap *may* have been read/written in the call and which *must* not have been read/written (and thus have the same *readloc–writeloc* entry as before the call).

Call Example. The anonymization and remapping operations are conceptually simple but without some intuition into how they function the definitions are difficult to follow. Thus, we first examine how the `swap` call is handled in the pair example. Figure 7 shows the steps that are taken to analyze the call at program location *m5* assuming that the memo table contains Subfigures 7(a) and 7(b) as a memoized result.

Figures 7(a) and 7(b) show that during the analysis of the `swap` method the analysis has determined the `first` and `second` fields have been read and written (the *readloc* and *writeloc* entries refer to program locations within the `swap` method, *s2*, *s3* and *s4*) but that the `val` fields are neither read nor written. The *readloc* and *writeloc* entries are the *modified–outside* value 0. Further, based on the identity tag sets we know that the object which was stored in the `first` field at the method entry (Figure 7(a)) and was given the identity tag 2 is stored in the `second` field at the method exit (Figure 7(b)). A similar situation holds for the object stored in the `second` field at the method entry, which was assigned the identity tag 3.

Figure 7(c) shows the state of the heap model at the call site (location *m5*) after we have added fresh tags (7, 8, and 9) to uniquely identify the nodes. After anonymizing the locations of the *readloc–writeloc* entries to the *modified–outside* value (0) we have the model shown in Figure 7(d), which is isomorphic (up to identity tags) to the model in our memo table, Figure 7(a).

During the anonymization we construct a map from the identity tags we added and the field identifiers to the *readloc–writeloc* entries in the caller scope that we are anonymizing. This gives us the map $ModM = \{(7,\texttt{first}) \rightarrow (m2,m2), (7,\texttt{second}) \rightarrow (m2,m2), (8,\texttt{val}) \rightarrow (m2,m3), (9,\texttt{val}) \rightarrow (m2,m2)\}$. Using the isomorphism from $\sigma_{in} \mapsto \sigma_{call}$ we have a map $\Pi = \{1 \rightarrow 7, 2 \rightarrow 8, 3 \rightarrow 9\}$.

Using these maps we transfer the read/write information from the call input to the memoized output, replacing any *readloc–writeloc* entries that refer to program locations in the callee body (`swap`) with the program location of the call site (program location *m5*) and replacing any occurrences of the *modified outside* value with the appropriate entry from *modM*. In Figure 7(b) the node with identify tag 2 has the *modified outside* value for the *readloc/writeloc* of the `val` field (`val-0-0`). To place the correct *readloc–writeloc* values into this node we look up the node that it maps to in the caller scope (via the Π map), which gives us the identity tag 8. Then we look up the caller scope *readloc–writeloc* information in the *modM* map, which gives us the read/write information for the field, `m2-m3`.

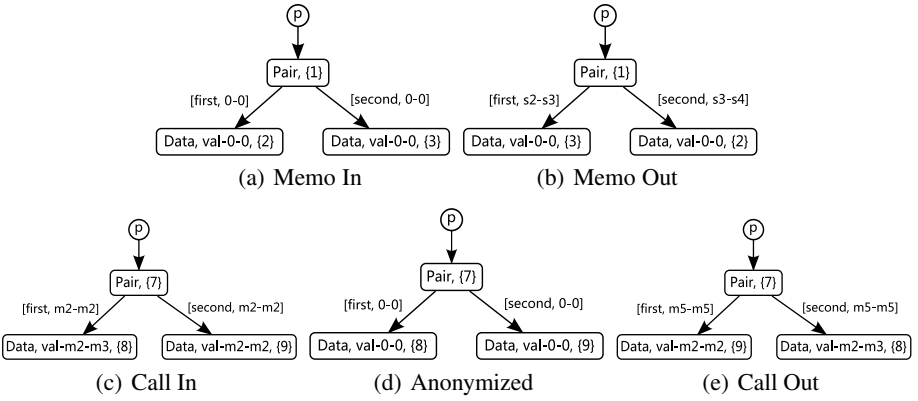


Fig. 7. Mapping Through Memoization

This remapping gives us the result in Figure 7(e), which shows that the object stored in the `second` field of the `Pair` object may have been written at program location *m3* but that the object stored in the `first` field has not been modified since initialization at program location *m2*. Thus, we can determine that the read from `p.first.val` is non-zero and the assertion will always succeed.

Dataflow Operations. For a method invocation at call site ℓ_{call} we give each node in the call state σ_{call} a unique tag $\kappa \in \mathbb{N}$, set the read/write location to the *modified outside* value and build a map $ModM : \mathbb{N} \times field \mapsto (\ell_r, \ell_w)$.

We then compare the anonymized version of σ_{call} with the entries in the memo table ignoring the read/write information. If a match $(\sigma_{in}, \sigma_{out})$ is found then there is a graph isomorphism $\Phi : \sigma_{in} \mapsto \sigma_{call}$. This isomorphism and the fact that the set of location tags in σ_{in} and σ_{out} are the same implicitly defines a map, $\Pi : \{\, \kappa \mid \kappa \textit{ a location tag } \in \sigma_{out}\} \mapsto \{\, \kappa' \mid \kappa' \textit{ a location tag } \in \sigma_{call}\}$. Using this map we can then compute the result of the call by replacing any *readloc–writeloc* values (ℓ_x) for the fields in each node n with:

$$(\ell'_x) = \begin{cases} \ell_{call}, \text{ if } \ell'_x \text{ is a location in the callee method} \\ \max(\{\, n'.\ell_x \mid \kappa \in n.\text{identity} \wedge n' \in \sigma_{call} \wedge \Pi(\kappa) \in n'.\text{identity}\}), \text{ otherwise} \end{cases}$$

5 Experimental Results

In this section we examine how the data dependence information can be used to perform thread level parallelization on variations of two of the more complex JOlden benchmarks, `em3d` and `bh` [10, 13]. To asses the performance of our approach we examine the analysis runtime on the JOlden suite, several of the SPECjvm98 benchmarks [16], and a logic formula manipulation program we developed as test case.

5.1 Case Studies

`Em3d`. The first application of the read/write dependence information we look at is performing thread–level parallelization of the `em3d` benchmark. In Figure 2 we show

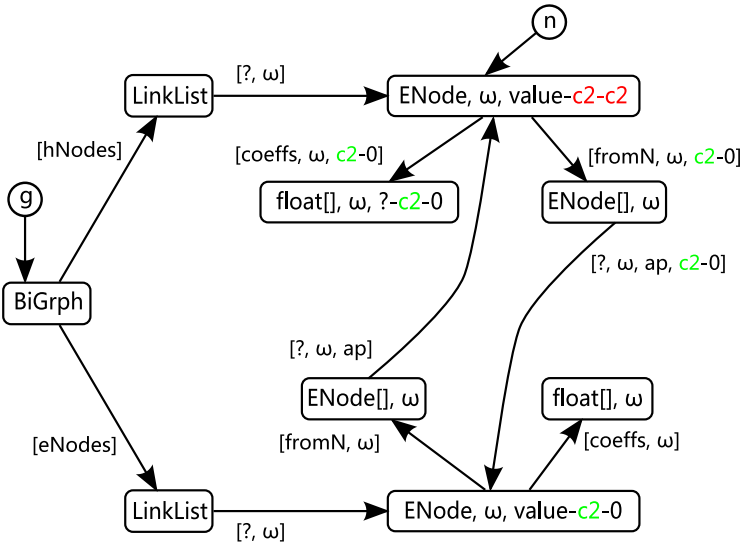


Fig. 8. Em3d With Read/Write Info

```
e1  for(int i = 0; i < this.hNodes.size(); ++i)
e2    computeNewValue((ENode) this.hNodes.get(i));
```

Fig. 9. Main Em3d Compute Loop

the code for updating the `value` field of a single `ENode` object. By applying our read/write analysis we obtain the model in Figure 8 at the end of the method body. We see that some object from the list of magnetic field nodes has had the `value` field both read and written in the loop, *readloc* = *c2* and *writeloc* = *c2* (marked in red if color is available), while there have been reads from the `coeffs` and `fromN` pointer fields, *readloc* = *c2* (marked in green), *writeloc* = 0. The pointers in the `fromN` array have also been read in order to access the `value` fields in the `ENode` objects in the opposite field, which have been read but not written (*readloc* = *c2*, *writeloc* = 0).

Using this information, the fact that each reference in the linked list (`LinkList`) of `ENode` objects refers to a unique object (the edge is *np*, the omitted default interference value) and the linear loop iteration, allows us to determine that each magnetic `ENode` object is written on a single iteration of the main update loop, program location *e2*, in Figure 9, which calls `computeNewValue`. Given this information it is valid to thread parallelize this loop (and to vectorize the loop in `computeNewValue`). Doing so results in a speedup of 3.21 on our quad-core test machine.

BH. Figure 10 shows the model that the analysis computes for the heap based read/write information in the `hackGravity` method of the *Barnes–Hutt* benchmark. For clarity we have simplified the heap structure in areas that are not relevant to this example.

The bh program performs a *fast–multipole* algorithm on the gravitational interaction between a set of bodies (the `Body` objects) and uses a space decomposition tree of

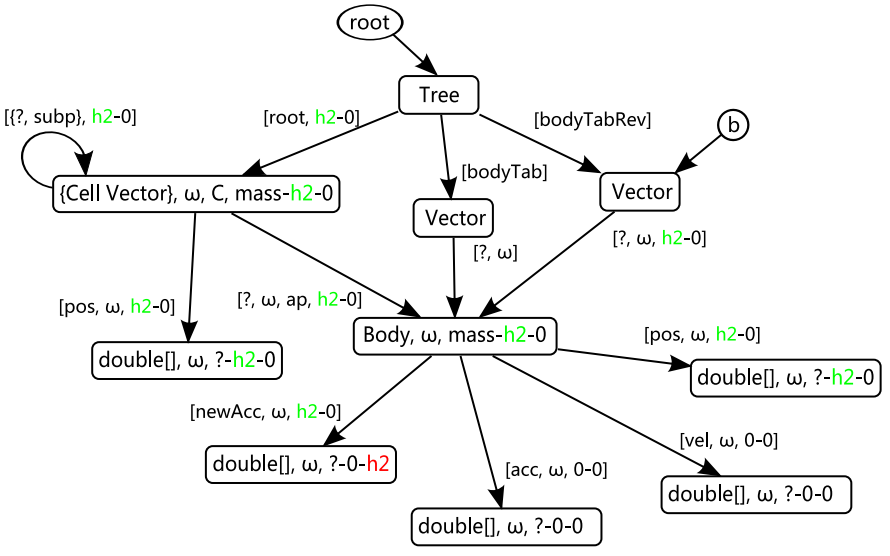


Fig. 10. BH With Read/Write Info

```
h1  Iterator b = this.bodyTabRev.iterator();
h2  while(b.hasNext())
h3    ((Body) b.next()).hackGravity(rsize, root);
```

Fig. 11. Main Update, Gravity Computation

`Cell` objects each of which has a `Vector` containing a subtree or a reference to the `Body` objects. The program also keeps two vectors for accessing the bodies, `bodyTab` and `bodyTabRev`. Figure 10 shows the state of the heap model after the loop body (Figure 11) that contains the majority of the computation in `bh`. This loop takes each `Body` object and walks the space decomposition tree (the `root` field) to determine a new acceleration value for the `Body` object (stored in the `newAcc` field).

Our analysis is not able to precisely resolve the construction of the space decomposition tree and conservatively assumes it may be a cyclic structure (shown by the `C` in the node representing the `Cell` objects). However, the analysis is able to determine that the `Cell` objects and the `Body` objects represent distinct regions in the program. This piece of information combined with the observation that the space decomposition tree is only read in the loop body (all the *readloc* entries set to *h2*, marked in green, and the *writeloc* entries set to 0), that the only part of the heap which is modified is never read (the `double[]` stored in the `newAcc` field, *writeloc* = *h2*, set to red), and that the collection being indexed over (the `Vector` referred to by the `bodyTabRev` field) does not have multiple references to the same object (the *?* edge is *np*, the omitted default interference value), is sufficient to ensure that there are no heap–carried dependence in this loop. Thus, we can safely thread–parallelize the loop body, achieving a factor of 2.98 speedup on our test machine.

5.2 Performance

The analysis algorithm was written in C++ and compiled using gcc 4.2. The analysis as well as the parallelization benchmarks were run on a 2.6 GHz Intel quad-core machine with 4 GB of RAM (although memory consumption never exceeded 60 MB).

The original Java programs are transformed into MIL programs and the required stub code is added to enable the analysis of the standard Java libraries (which requires from 200-600 lines depending on which libraries the benchmark uses). These MIL programs are then processed by the analyzer. A demo version of the analyzer and benchmarks can be obtained at [13].

Benchmark	LOC	Classes	Methods	Analysis Time	Shape	RW Dep
bisort	560	36	348	0.26s	Y	Y
mst	668	52	485	0.12s	Y	Y
tsp	910	42	429	0.15s	Y	Y
em3d	1103	56	488	0.31s	Y	Y
perimeter	1114	44	381	0.91s	P	N
health	1269	59	534	1.25s	Y	Y
voronoi	1324	58	549	1.80s	Y	Y
power	1752	57	520	0.36s	Y	Y
bh	2304	61	576	1.84s	P	Y
db	1985	68	562	1.42s	Y	Y
logic	3960	72	620	48.26s	P	Y
raytrace	5809	63	506	37.09s	Y	Y

Fig. 12. LOC is the size of the program after transformation to MIL (including library stub code that must be analyzed), Classes/Methods are the number of classes/methods in the program (including Java Libraries that are used). Shape reports the heap connectivity is correctly identified and RW Dep reports if the RW information is useful (as in Section 5.1).

We report Y(es) in the *Shape* column if the analysis correctly identified all the relevant the shape information of the heap structures in the program. P(artial) means the analysis was able to determine the precise shape for some of the data structures but that some properties were missed.

We report similar information for the utility of the RW information. Y(es) means the read/write information would be sufficient to introduce substantial thread level parallelism (as in Section 5.1) and provides the information required to enable significant instruction level parallelism optimizations (e.g. code motion to improve scheduling, elimination of redundant loads/stores or the identification of loop invariant values). We Report (N)o for only one of the benchmarks, perimeter, where the read/write information does not enable any thread level parallelism and only enables minor scheduling or load elimination opportunities.

Our experimental results show that the analysis is capable of efficiently computing very precise heap–carried dependence information over a range of benchmarks. In particular the ability to compute this information on the benchmarks bh, em3d, voronoi and raytrace is a significant advance in the state of the art for understanding the program heap. Computing precise shape and dependence information for these benchmarks

requires the analysis to precisely model recursive data structures, Java collections, non-trivial sharing between components of the heap and, in order to compute the dependence information, to precisely track the part of the heap each read/write affects.

The analysis presented in this paper is not only capable of accurately modeling all of these features but is able to do so efficiently (analyzing the smaller benchmarks takes less than 2s per benchmark and raytrace at 5809 LOC takes only 37s). Based on these results we believe that the analysis reported in this paper is robust enough to be generally useful in the optimization of smaller Java programs and we plan to continue work on scaling the analysis to handle larger programs with the same level of precision.

References

1. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P., Wies, T., Yang, H.: Shape analysis for composite data structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 178–192. Springer, Heidelberg (2007)
2. Cheng, B.-C., Mei, W., Hwu, W.: Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. ACM SIGPLAN Notices (2000)
3. Choi, J.-D., Burke, M., Carini, P.: Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In: POPL (1993)
4. Ghiya, R., Hendren, L.J., Zhu, Y.: Detecting parallelism in C programs with recursive data structures. In: CC (1998)
5. Gulwani, S., Tiwari, A.: An abstract domain for analyzing heap-manipulating low-level software. In: CAV (2007)
6. Guo, B., Vachharajani, N., August, D.: Shape analysis with inductive recursion synthesis. In: PLDI (2007)
7. Hendren, L.J., Nicolau, A.: Parallelizing programs with recursive data structures. IEEE TPDS 1(1) (1990)
8. Horwitz, S., Pfeiffer, P., Reps, T.W.: Dependence analysis for pointer variables. In: PLDI (1989)
9. Hummel, J., Hendren, L.J., Nicolau, A.: A general data dependence test for dynamic, pointer-based data structures. In: PLDI (1994)
10. Suite, J.: `http://www-ali.cs.umass.edu/DaCapo/benchmarks.html`
11. Manevich, R., Sagiv, S., Ramalingam, G., Field, J.: Partially disjunctive heap abstraction. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 265–279. Springer, Heidelberg (2004)
12. Marron, M., Kapur, D., Stefanovic, D., Hermenegildo, M.: A static heap analysis for shape and connectivity. In: Almási, G.S., Caşcaval, C., Wu, P. (eds.) KSEM 2006. LNCS, vol. 4382, pp. 345–363. Springer, Heidelberg (2007)
13. Modified Jolden and Demo (May 2008), `http://www.cs.unm.edu/~marron`
14. Rugina, R., Rinard, M.C.: Automatic parallelization of divide and conquer algorithms. In: PPOPP (1999)
15. Sagiv, S., Reps, T.W., Wilhelm, R.: Solving shape-analysis problems in languages with destructive updating. In: POPL (1996)
16. Standard Performance Evaluation Corporation. JVM98 Version 1.04 (August 1998), `http://www.spec.org/jvm98`

On the Scalability of an Automatically Parallelized Irregular Application

Martin Burtscher, Milind Kulkarni, Dimitrios Prountzos, and Keshav Pingali

Center for Grid and Distributed Computing
Institute for Computational Engineering and Sciences
The University of Texas at Austin
Austin, TX 78712
{burtscher,milind}@ices.utexas.edu,
{dprountz,pingali}@cs.utexas.edu

Abstract. Irregular applications, *i.e.*, programs that manipulate pointer-based data structures such as graphs and trees, constitute a challenging target for parallelization because the amount of parallelism is input dependent and changes dynamically. Traditional dependence analysis techniques are too conservative to expose this parallelism. Even manual parallelization is difficult, time consuming, and error prone. The Galois system parallelizes such applications using an optimistic approach that exploits higher-level semantics of abstract data types.

In this paper, we study the performance and scalability of a Galoised, that is, automatically parallelized, version of Delaunay mesh refinement (DR) on a shared-memory system with 128 CPUs. DR is an important irregular application that is used, *e.g.*, in graphics and finite-element codes. The parallelized program scales to 64 threads, where it reaches a speedup of 25.8. For large numbers of threads, the performance is hampered by the load imbalance and the nonuniform memory latency, both of which grow as the number of threads increases. While these two issues will have to be addressed in future work, we believe our results already show the Galois approach to be very promising.

Keywords: parallel programming, multicore processors, sparse graph algorithm, amorphous data-parallelism, optimistic execution, mesh refinement.

1 Introduction

Over the last three decades, the problem of automatic parallelization, *i.e.*, the mechanical transformation of sequential code into parallel code by identifying program regions that can execute concurrently, has been studied extensively. As a result, modern compilers are able to achieve very good parallel performance in certain application domains virtually without programmer guidance.

In particular, for applications that process arrays and matrices, which we refer to as "regular" applications, a multitude of techniques have been developed to prove independence between array accesses and to uncover, package, and schedule parallelism

J.N. Amaral (Ed.): LCPC 2008, LNCS 5335, pp. 109–123, 2008.

at various levels [6]. In this class of programs, data parallelism mainly manifests itself as FOR-ALL loops over integer intervals for which the iterations can be statically proven to be independent. We call this *crystalline data-parallelism*.

However, there exist a large number of important "irregular" applications that manipulate sparse graphs, which are much harder to parallelize. A characteristic example of this class of programs is Delaunay mesh refinement, a widely used computational geometry algorithm. For these applications, static parallelization techniques based on pointer [4] and shape analysis [5], [11], [16] are often insufficient because they must be correct for all possible inputs. However, the amount of parallelism in irregular applications is almost always data dependent and changes dynamically. We call this *amorphous data-parallelism*. For example, in Delaunay refinement, the parallelism is highly dependent on the shape of the input mesh. Thus, statically produced parallel schedules tend to be overly conservative for most inputs and unnecessarily serialize program execution.

In semi-static approaches, the computation is split into an inspector phase, which determines the dependences between units of work, and an executor phase, which uses this schedule to perform the computations concurrently [14]. Since the inspector is also executed at runtime, the input of the program is taken into account when producing the schedule. For Delaunay refinement, the usefulness of this approach is, however, limited because the mesh changes as the algorithm progresses. Hence, the inspector would have to be executed repeatedly, which is expensive because it involves expanding the cavities (*i.e.*, a substantial part of the work of a single iteration).

The most promising way to automatically parallelize irregular programs is employing dynamic approaches that speculatively parallelize the code at runtime. In this approach, portions of the application are executed in parallel assuming that dependences are not violated. The runtime system is responsible for detecting any such violations and for restoring the program to the correct state by aborting one of the conflicting computations and executing it later. If no dependence violation is detected, the speculative state is committed, thus becoming visible to the rest of the program.

In previous work, we introduced the Galois system [10], which we discuss in more detail below, to automatically and speculatively parallelize irregular code. While we believe our system to be practical, our previous studies [8], [9], [10] have not investigated the scalability beyond small multicore systems. The goal of this paper is to study the performance of an application that has been automatically parallelized by the Galois system on a large-scale shared-memory multiprocessor. This study not only provides insight into the effectiveness of our approach but also brings out important issues pertaining to the parallelization of irregular applications in a real-world setting.

The rest of the paper is organized as follows. Section 2 discusses the Delaunay mesh refinement algorithm in detail. Section 3 illustrates the Galois system. Section 4 presents the experimental methodology. Section 5 shows the results. Section 6 summarizes related work. Section 7 concludes the paper with a summary and future work.

2 Delaunay Mesh Refinement

Mesh generation is a vital component of many applications in graphics and the numerical solution of partial differential equations. The goal of mesh generation is to

represent a surface or a volume as a tessellation composed of simple shapes like triangles or tetrahedra.

Although many types of meshes are used in practice, Delaunay meshes are particularly important since they have a number of desirable mathematical properties [3]. The Delaunay triangulation of a set of points in the plane is the triangulation such that no point is inside the circumcircle of any triangle. This property is called the *empty circle* property. An example of such a mesh is given in Fig. 1.

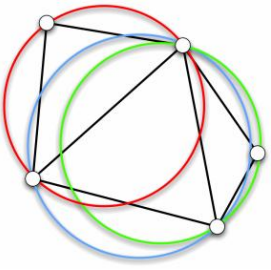


Fig. 1. Delaunay mesh (the circumcircles of the triangles contain no mesh points)

In practice, the Delaunay property alone is not sufficient, and it is necessary to impose quality constraints governing the shape and size of the triangles. For a given Delaunay mesh, this is accomplished by *iterative mesh refinement*, which successively fixes "bad" triangles (triangles that do not satisfy the quality constraints) by adding new points to the mesh and re-triangulating it. Fig. 2 illustrates this process.

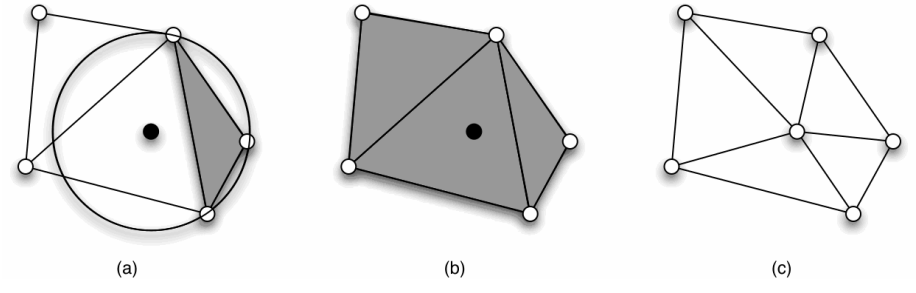


Fig. 2. Delaunay mesh refinement steps

The shaded triangle in Fig. 2(a) is assumed to be bad. To fix it, a new point is added at the center of this triangle's circumcircle. Adding this point may invalidate the empty circle property of some neighboring triangles. Hence, all affected triangles need to be determined. This region is called the *cavity* of the bad triangle and is shaded in Fig. 2(b). In this example, all triangles belong to the cavity, but in larger meshes, a cavity usually only covers a small fraction of the mesh. In the final step, the cavity is re-triangulated as shown in Fig. 2(c). Re-triangulating a cavity may generate new bad triangles, but it can be proven that this iterative refinement process will ultimately terminate and produce a guaranteed-quality mesh [3]. Different orders of

processing bad triangles may lead to different meshes, but all such meshes satisfy the quality constraints.

Fig. 3 provides pseudocode for mesh refinement. The input is a Delaunay mesh in which some triangles may be bad, and the output is a refined mesh in which all triangles satisfy the quality constraints. There are two key data structures used in this algorithm. One is a worklist containing the bad triangles in the mesh. The other is a graph representing the mesh structure where the nodes correspond to the triangles and the edges denote triangle adjacencies. The two-dimensional algorithm works as follows.

1. Find all the bad triangles in the mesh and put them into the worklist [line 3]. Then repeat the following steps until the list of bad triangles is empty [line 4].
2. Pick a triangle from the list [line 5]. The processing of other bad triangles may have removed this triangle from the mesh. If so, there is nothing to do [line 6].
3. Compute the cavity of the bad triangle as follows. Find the circumcenter of the triangle, add this new point to the mesh and determine the triangles that no longer satisfy the empty circle property because of this new point [lines 7 and 8].
4. Re-triangulate the cavity [line 9].
5. Replace the triangles in the cavity with the new triangles (*i.e.*, remove the old triangles from the mesh and add in the newly calculated triangles) [line 10].
6. Because the newly created triangles are not guaranteed to meet the quality constraints, any newly created bad triangles must be added to the worklist [line 11].

```
 1: Mesh mesh = ...;  // read in initial mesh
 2: WorkList wl;
 3: wl.add(mesh.badTriangles());
 4: while (wl.size() != 0) {
 5:   Triangle t = wl.get();  // get bad triangle
 6:   if (t no longer in mesh) continue;
 7:   Cavity c = new Cavity(t);
 8:   c.expand();
 9:   c.retriangulate();
10:   mesh.update(c);
11:   wl.add(c.badTriangles());
12: }
```

Fig. 3. Pseudocode of the 2D mesh refinement algorithm

2.1 Opportunities for Exploiting Amorphous Data-Parallelism

The natural unit of work for parallel execution in Delaunay mesh refinement is the processing of a bad triangle. Because a cavity is typically a small neighborhood of a bad triangle, the cavities of two bad triangles that are far apart in the mesh often do not overlap and can therefore be processed concurrently.

An example of processing several triangles in parallel is given in Fig. 4. The left mesh is the original mesh, and the right mesh represents its refinement. In the left mesh, the *black* triangles are the bad triangles while the *dark grey* triangles are the other triangles in the cavities. In the right mesh, the *black* points mark the newly added points and the *light grey* triangles denote the newly created triangles. Clearly, all the cavities in Fig. 4 can be refined in parallel without conflicts. Thus, Delaunay mesh refinement is an example of a worklist algorithm where the units of work may be independent.

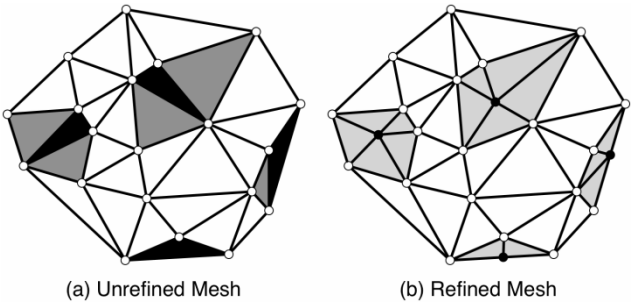


Fig. 4. Processing triangles in parallel

3 The Galois Model

The Galois programming model [10] is a concurrent, object-based, shared-memory model that is designed to be implemented as an extension to an object-oriented language. An application parallelized under this model consists of two components: the client code, which is written by the user of the system and has easily-understood sequential semantics, and the library and runtime code, which encapsulates all the complexity of parallel execution.

3.1 Client Code

The programming model provides two language constructs, called optimistic set iterators, that allow the user to implicitly express amorphous data-parallelism. The well-defined sequential semantics of these set iterators makes it easier to understand, write, and debug client code.

- ***Set iterator***: `for each e in Set S do B(e)`
  The loop body B(e) is executed for each element e of set S. Since the elements of a set are not ordered, this construct denotes that, in a serial execution of the loop, the iterations can be executed in any order. There may be dependences between the iterations, as is the case with Delaunay mesh refinement, but any serial order of executing iterations is permitted. Iterations may dynamically add elements to S.
- ***Ordered-set iterator***: `for each e in OrderedSet S do B(e)`
  This construct denotes a partially-ordered iterator over S. Contrary to the Set iterator, the execution order must respect the partial order imposed by the OrderedSet S.

3.2 The Galois Runtime and Class Libraries

The runtime system speculatively executes iterations of set iterators in parallel, thereby taking advantage of potential amorphous data-parallelism in the application. To guarantee that the parallel execution preserves the sequential semantics of the iterators, the system must ensure that concurrent accesses of and method invocations on shared objects are properly coordinated.

One way to detect conflicts in the Galois system is through *commutativity checks*. Intuitively, two iterations that concurrently access a shared object do not conflict if they call commuting methods on the object. Note that commutativity conditions are a property of the abstract data type of the object and are therefore only dependent on the public interface of the type and not on the concrete implementation of that interface. Thus, the implementation can be changed without affecting the commutativity conditions. These conditions are specified as annotations in the class definition [10].

Alternatively, conflict detection can be performed on *partitioned* data structures [9]. For example, a Delaunay mesh can easily be partitioned. Whenever two iterations touch the same partition of a shared data structure, a conflict is raised. Even though this scheme is less precise than commutativity checking, it is simpler and has a lower overhead, which is why we use it in this study.

The Galois system also supports overdecomposition. The basic idea of overdecomposition is to partition the data into more partitions than there are cores in the machine so that multiple partitions are mapped to each core. When a thread accesses a partition of a data structure, it owns all elements in that partition, and the other threads are not allowed to access them. Assigning multiple partitions to a core increases the probability that a thread can continue to perform useful work even if other threads have temporarily locked some of its partitions [9].

Whenever a conflict is detected, one or more iterations must be rolled back, *i.e.*, a series of *undo actions* are executed by the runtime system. For each method of a shared object type, the class implementor must provide another method that performs a "semantic undo". For example, the undo of *add(x)* in a set is *remove(x)*. As each iteration executes, the system records the undo actions corresponding to the methods that get called and uses them to perform a rollback if a conflict occurs.

4 Methodology

To study the scalability and other performance aspects of our Galoised version of Delaunay mesh refinement, we performed experiments on a Sun E25K server running SunOS 5.9. The system contains sixteen CPU boards with four dual-core 1.05 GHz UltraSPARC IV processors. The 128 CPUs share 512 GB of main memory. Each core has a 64 kB four-way set-associative L1 data cache and a unified 8 MB L2 cache.

We use Sun's Java compiler version 1.6.0_02 and the HotSpot 64-bit server virtual machine version 1.6.0-b105. Because HotSpot dynamically compiles frequently executed bytecode into native machine code, we repeat each experiment nine times in the same VM and report results for the median as well as the fastest run. To prevent other jobs from interfering with our measurements, we always reserve all 128 CPUs regardless of how many threads we create. Furthermore, to minimize the interference by the garbage collector, we use a 400 GB heap and force a garbage collection by calling System.gc() five times before executing the measured code section.

All measurements are obtained through source code instrumentation; that is, we read the timer and the CPU performance counters before and after the measured code section, compute the difference, and write the result to the standard output. We use the Java Native Interface and C code we wrote to access the performance counters.

We evaluate Delaunay mesh refinement on three random inputs. The small input contains 100,770 triangles of which 47,768 are initially bad. The middle input has 219,998 triangles of which 104,229 are initially bad. The large input consists of 549,998 triangles of which 261,100 are initially bad.

All measurements in this study refer to the refinement algorithm only. In particular, reading the input, building the initial graph, and partitioning the initial graph are excluded as we have not yet Galoised these components of the code.

5 Results

5.1 Speedup

Fig. 5 shows the speedup of the parallel code on the three inputs for various thread counts relative to the fastest run of our sequential implementation. The solid lines display the best and the dashed lines the median speedups. The overdecomposition factor is 32. There was no garbage collection during the execution of the timed code. The sequential refinement code takes 31.96, 76.16, and 195.1 seconds, respectively, for the small, medium, and large input.

The automatically parallelized code scales to 64 or 128 threads, depending on the input. Scaling is good (over 50% efficiency) up to 32 threads, where the speedup is roughly 20 for all three inputs. The large input scales to 128 threads, where it reaches

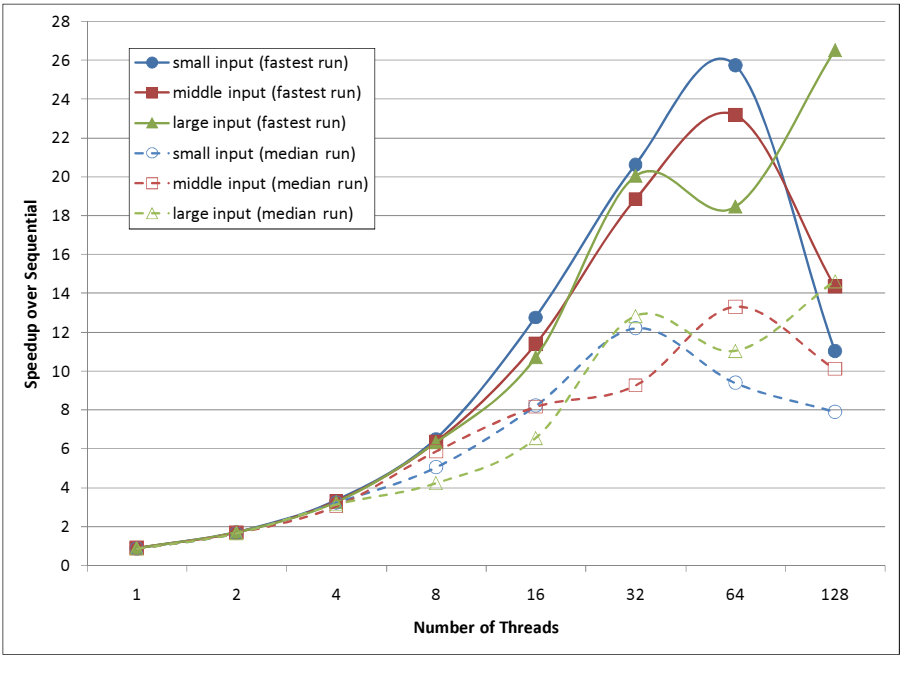


Fig. 5. Speedup over the fastest sequential run

a speedup of a 26.5, the highest we observed. The performance drop with 64 threads is due to a high CPI (*cf.* Section 5.2), which we believe is caused by unfortunate partitioning that results in a large amount of communication.

The median runtimes start to diverge from the fastest runtimes at eight threads because of slow communication between CPU boards. Recall that our machine comprises sixteen boards with eight processors each. Thus, experiments with eight or fewer threads may incur only intra-board communication whereas experiments with sixteen or more threads necessarily incur inter-board communication, which is slow. Hence, as long as the operating system executes all threads on CPUs of the same board, the runtimes of the nine experiments vary only slightly and the median is very close to the fastest runtime. However, as soon as multiple boards are involved, which may already happen with eight worker threads because of other JVM threads, it greatly matters whether threads that exchange data are assigned to the same board or not. Because some assignments result in better locality than others, the random allocation of threads to cores by the operating system yields different runtimes for each experiment, as is reflected by the discrepancy of up to a factor of 2.7 between the median and the best speedups. Due to this high variability, we believe the results reported in this paper for large numbers of threads show the correct trends but the absolute values might be unreliable.

The parallel code with one thread is 13% slower than our sequential implementation. This result reflects the overhead introduced by Galois, which includes the time it takes to start and terminate worker threads, the cost of checking for runtime conflicts (even though no conflicts can occur with just one thread), and the expense of recording the undo information.

In summary, the scaling is surprisingly good for an automatically parallelized irregular application. The efficiency is better than 66.6% up to 16 threads for the fastest runs. Above 16 threads, it drops quickly due to load imbalance and memory latency.

5.2 Memory Access Latency

To confirm the negative impact of the inter-board communication, we measured the average number of cycles it takes to execute an instruction. Fig. 6 shows the results. Because the same code is executed and because most instructions that do not access the memory have a fixed latency, we attribute any increase in the number of cycles per instruction (CPI) to slower memory accesses. Direct measurements of the L2 cache stall cycles corroborate our results but only capture part of the memory latency.

The CPI (and therefore the memory latency) starts to greatly increase above 16 threads, exposing the nonuniformity in the memory access time. With large numbers of threads, the CPI is up to 2.44 times as high as the CPI with a single thread. Since instructions that touch the memory represent only a fraction of the executed instructions, the average slowdown per memory access is, of course, even higher. Thus, the slow inter-board communication speed is one of the primary performance hurdles in the Galoised refinement code for large numbers of threads on our system.

5.3 Load Balance

Fig. 7 illustrates the fraction of time that the concurrent threads have to wait, on average, for the slowest thread to finish. For example, in the 128-thread run with the small

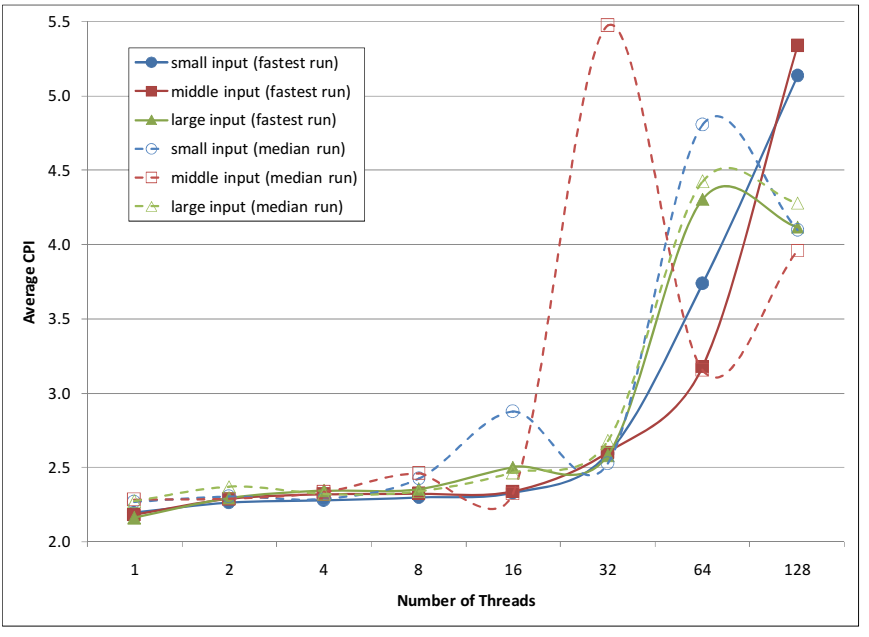


Fig. 6. Average cycles per instruction (CPI)

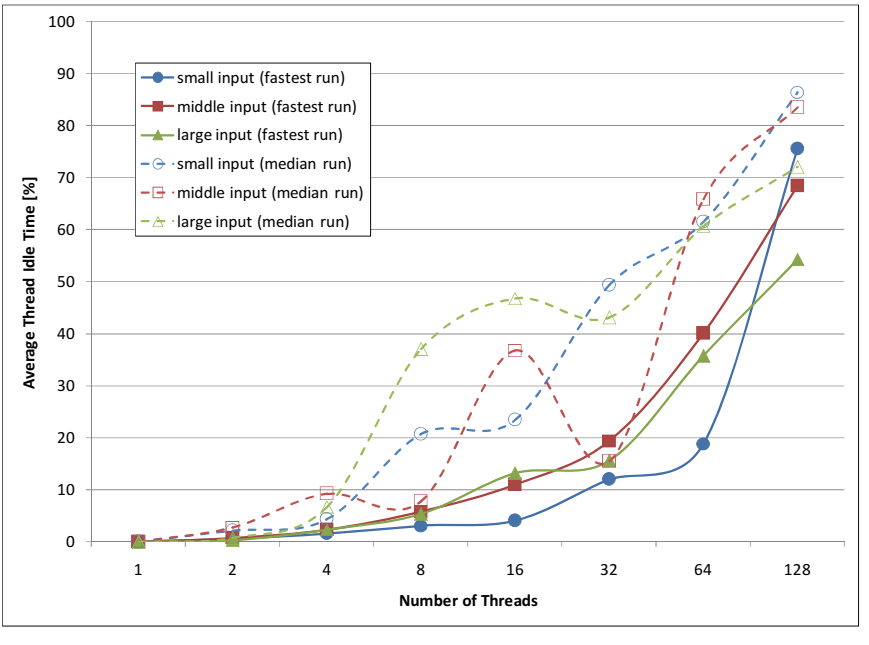


Fig. 7. Average thread waiting time as a percentage of the total runtime

input, the threads idle 75.5% of the time, on average. Note that there is no task stealing and new work generated by a thread is always handled by that same thread.

We observe very little load imbalance (under six percent idle time) up to eight threads. Above eight threads, the imbalance starts to become significant and for 128 threads, on average over half of the time the threads are idling for all three inputs. The load imbalance grows with the number of threads because larger thread counts result in less work per thread, which increases the likelihood of imbalance problems. Thus, load imbalance is the second main reason preventing the automatically parallelized code from scaling well to 128 CPUs. Note that, on the one hand, our random inputs are quite homogenous and thus probably more balanced than real inputs, meaning that load imbalance may be an even bigger problem in practice. On the other hand, we use a somewhat naive work partitioner based on recursive subdivision, and employing a more sophisticated partitioner may improve the load balance.

5.4 Aborted Speculations

Some of the speculative refinements fail because their cavity extends to a partition of the graph that is locked by another thread. Fig. 8 depicts the fraction of the attempted refinements that had to be aborted (and retried later). The overdecomposition factor is 32. In the median run with the large input, 18.8% of the refinements were aborted.

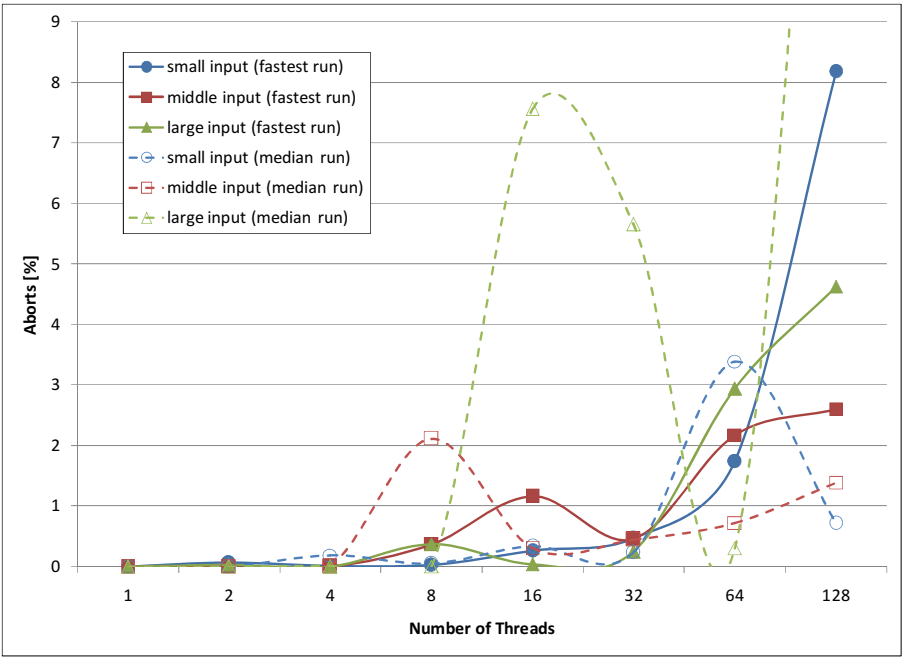


Fig. 8. Percentage of attempted refinements that were aborted due to speculation conflicts

There are relatively few aborts, as one might expect with a large overdecomposition factor. With the fastest runs, we see almost no aborts up to four threads. Larger numbers of threads cause more aborts for two reasons. First, the higher latency of the aforementioned inter-board communications slows down some of the refinements, meaning that they take longer and are therefore more likely to conflict with other concurrent refinements. This is probably also the reason why the (slower) median runs sometimes have much higher abort ratios than the fastest runs. Second, increasing the thread count increases the number of partitions but makes them smaller. Hence, the chance of a cavity overlapping multiple partitions increases.

Nevertheless, the observed abort ratios are too low to severely impact the scalability. In fact, aborts are detected quite early, namely during the cavity expansion and before the actual refinement work. As a result, they only have a small effect on the runtime (*cf.* Section 5.6).

5.5 Overdecomposition

Fig. 9 illustrates the impact of the overdecomposition factor on the speedup over the sequential code. For improved readability, we only show results for the fastest runs with the middle input.

As one might expect, one partition per thread results in poor performance. Two partitions per thread are necessary and sufficient for good performance up to eight

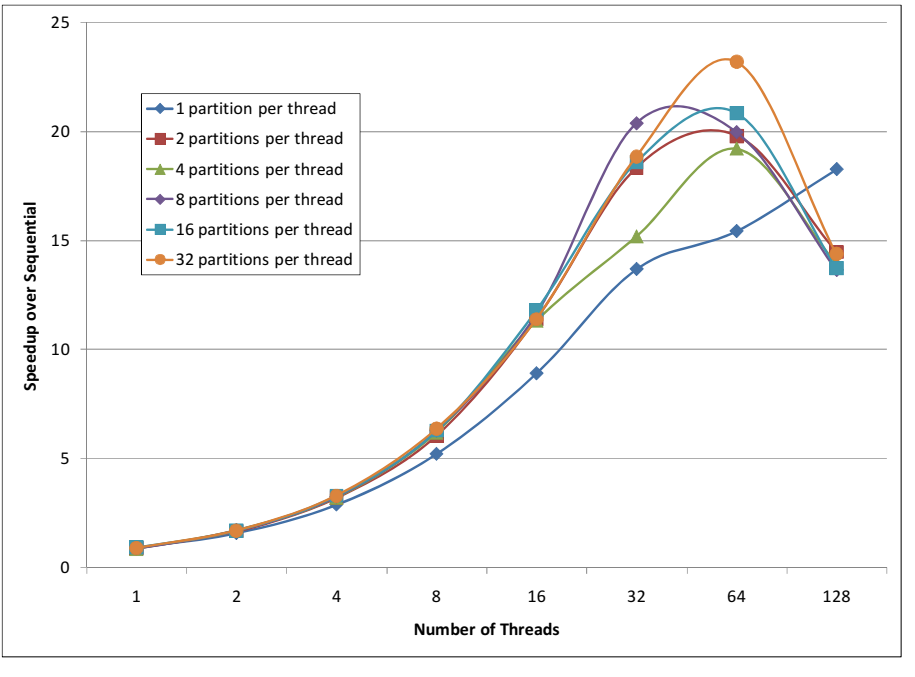


Fig. 9. Speedup of the fastest runs and middle input for different overdecomposition factors

threads. For larger numbers of threads, higher overdecomposition factors tend to help because they lower the misspeculation rate of the optimistic execution. However, beyond a certain level of overdecomposition, there is little benefit in using smaller partitions. In fact, the locking overhead increases as the partitions become smaller because the cavities are more likely to span multiple partitions, which necessitates the acquisition of multiple locks. The odd behavior with one partition per thread and 128 threads may be an artifact of the aforementioned high variability in our measurements with large thread counts.

5.6 Result Summary

Fig. 10 summarizes the results from the previous subsections by accumulating the runtime of all threads. The figure shows numbers for the fastest run with the middle input. The results for the other two inputs are qualitatively similar. The runtimes are relative to the sequential runtime. Each bar is broken down into five categories. They are, from bottom to top, 1) the runtime of the sequential code, 2) the single-thread overhead, *i.e.*, the runtime of the parallel code with just one thread, 3) the aborted work due to misspeculations, 4) the memory latency as computed in Section 5.2, and 5) the time the threads idle while waiting for the slowest thread to finish.

Because Fig. 10 sums up the runtime across all threads, the total runtime would be the same regardless of the number of threads if the parallel implementation scaled perfectly. However, as we noted in the previous subsections, there are factors that

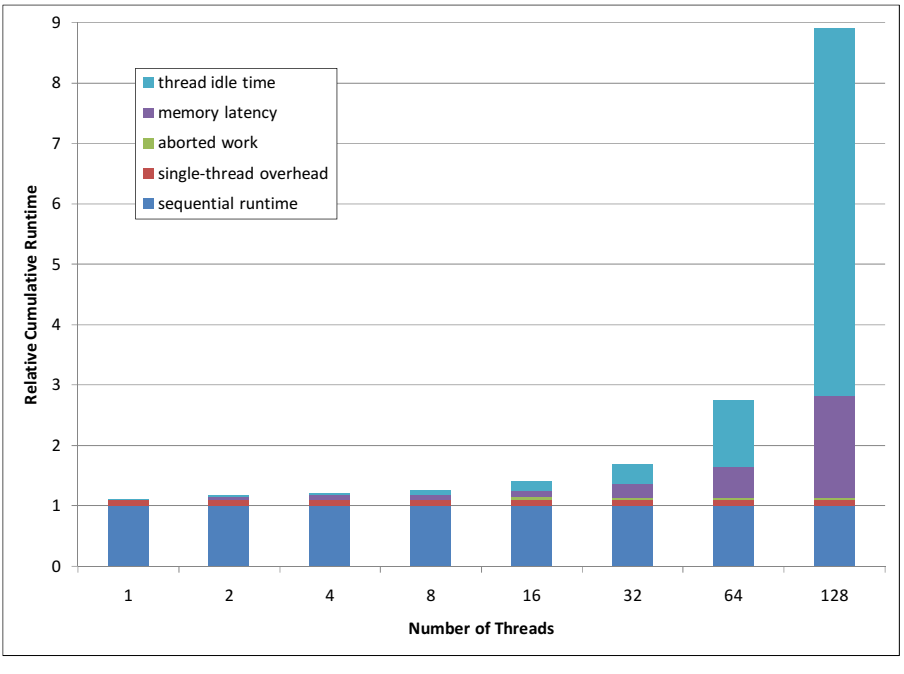


Fig. 10. Accumulative runtime breakdown for the fastest run with the middle input

hamper the scalability of Delaunay mesh refinement. Up to four threads, the overheads of the Galois system remain low, and hence the total runtime stays roughly constant. However, beyond four threads we see that the load imbalance and the memory latency begin to increase rapidly. Both of these overheads are the result of processors being unable to perform useful work (either because of waiting for memory or for slower threads). For large numbers of threads, the load imbalance represents the biggest performance bottleneck followed by the memory latency. The aborted work and the parallelization (single-thread) overhead are minor in comparison.

6 Related Work

Hand-written parallel implementations of Delaunay mesh refinement exist for 2D [1] and 3D [2] meshes. While both implementations eschew optimistic parallelization, they are not amenable to automatic parallelization (an automatic approach could not generate these particular parallel implementations from the sequential algorithm) for several reasons. In the 2D code, the mesh is partitioned among multiple processors, and each partition is processed relatively independently. However, the standard Delaunay algorithm is augmented with special handling for the boundaries between partitions. Thus, this approach is effectively a new algorithm for parallel Delaunay refinement rather than a straightforward parallelization of the sequential algorithm. In the 3D code, the mathematical properties of the Delaunay algorithm were examined and the authors developed a distance metric establishing the greatest possible size of a cavity in the mesh. Thus, regions sufficiently far apart can be processed in parallel. This approach to parallelization requires specific algorithmic knowledge and, again, is therefore not a straightforward parallelization of the sequential algorithm.

Other approaches to automatic parallelization of irregular programs include Thread Level Speculation (TLS) [7], [15]. This technique automatically parallelizes FOR loops in sequential programs using optimistic parallelization. However, because TLS focuses on parallelizing standard sequential programs, it cannot leverage key algorithmic semantics in the parallelization. Thus, the generated parallel programs must exactly match the sequential program, preventing TLS from, *e.g.*, reordering parallel computation to better exploit locality.

Another approach that has been studied extensively is Transactional Memory [12] (TM). One key distinction between the Galois approach and TM is that the latter is mainly concerned with *optimistic synchronization* as opposed to *optimistic parallelization*. In other words, the input program for a TM system has already been parallelized and the goal is to find an efficient and less error-prone way to synchronize the parallel tasks. In contrast, the main concern of the Galois model is to present the user with the right abstractions to express the amorphous data-parallelism in irregular codes as well as to provide an efficient implementation of those abstractions.

TLS and TM both detect speculative conflicts based on memory-level consistency. As we discuss elsewhere [10], tracking conflicts at such a low level may trigger false conflicts and thus disallow parallel execution that is actually safe. One way to overcome this problem in the case of Transactional Memory is to use open nested transactions [13]. This approach, however, complicates the semantics of the program and, as a result, increases the effort required by the programmer.

7 Conclusions and Future Work

This paper studies the scalability of an important "irregular" sparse graph application, Delaunay mesh refinement, which has been automatically parallelized using the Galois system. Our measurements on a 128-CPU shared-memory computer identified the load imbalance and the long nonuniform memory latency (due to inter-board communication) to be the primary bottlenecks to scaling for large numbers of threads. Speculation aborts and the overdecomposition factor have a relatively minor impact on the performance. Overall, the automatically parallelized code scales to 64 or 128 threads, depending on the input, and achieves a speedup of 26.5 over the sequential code.

While this work only investigates a single application, it raises several issues that are known to be problematic in parallelization. Thus, we believe Delaunay mesh refinement to be a representative amorphous data-parallel program worth studying and our findings to be more generally applicable. For instance, future multicore systems will likely also have nonuniform memory latency.

Addressing this issue is our primary target for future work. To minimize the inter-board communication, *i.e.*, the slowest memory accesses, we will hierarchically partition the work and pin it to CPUs such that the memory hierarchy (including the interconnection network) matches the hierarchy of the work partitions. To address the load imbalance, we will modify the work scheduler and add support for work or partition stealing. Other future work includes Galoising the sequential mesh partitioner, which currently takes longer to run than the parallel refinement code.

Acknowledgments

This work is supported in part by NSF grants 0833162, 0719966, 0702353, 0615240, 0541193, 0509324, 0509307, 0426787 and 0406380, as well as grants from IBM and Intel Corporation. Milind Kulkarni is supported by a DOE HPCS Fellowship.

References

1. Chernikov, A., Chrisochoides, N.: Parallel 2D Constrained Delaunay Mesh Generation. ACM Transactions on Mathematical Software 34(1) (2008)
2. Chernikov, A., Chrisochoides, N.: Three-dimensional Delaunay Refinement for Multi-core Processors. In: 22nd International Conference on Supercomputing, pp. 214–224 (2008)
3. Chew, L.P.: Guaranteed-quality Mesh Generation for Curved Surfaces. In: Ninth Annual Symposium on Computational Geometry (1993)
4. Ghiya, R., Hendren, L.J.: Putting pointer analysis to work. In: 25th Symposium on Principles of Programming Languages, pp. 121–133 (1998)
5. Hendren, L.J., Nicolau, A.: Parallelizing Programs with Recursive Data Structures. IEEE Transactions on Parallel and Distributed Systems 1(1), 35–47 (1990)
6. Allen, R.J., Kennedy, K.: Optimizing Compilers for Modern Architectures: a Dependence-based Approach. Morgan Kaufmann Publishers Inc., San Francisco (2002)
7. Krishnan, V., Torrellas, J.: A Chip-multiprocessor Architecture with Speculative Multithreading. IEEE Transactions on Computers 48(9) (1999)

8. Kulkarni, M., Carribault, P., Pingali, K., Ramanarayanan, G., Walter, B., Bala, K., Chew, L.P.: Scheduling Strategies for Optimistic Parallel Execution of Irregular Programs. In: Symposium on Parallelism in Algorithms and Architectures, pp. 217–228 (2008)
9. Kulkarni, M., Pingali, K., Ramanarayanan, G., Walter, B., Bala, K., Chew, L.P.: Optimistic Parallelism Benefits from Data Partitioning. In: International Conference on Architectural Support for Programming Languages and Operating Systems, vol. 36(1), pp. 233–243 (2008)
10. Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., Chew, L.P.: Optimistic Parallelism Requires Abstractions. In: Conference on Programming Language Design and Implementation, vol. 42(6), pp. 211–222 (2007)
11. Larus, J.R., Hilfinger, P.N.: Detecting Conflicts between Structure Accesses. In: Conference on Programming Language Design and Implementation (1988)
12. Larus, J., Rajwar, R.: Transactional Memory (Synthesis Lectures on Computer Architecture). Morgan & Claypool Publishers, San Francisco (2007)
13. Ni, Y., Menon, V.S., Adl-Tabatabai, A.R., Hosking, A.L., Hudson, R.L., Moss, J.E.B., Saha, B., Shpeisman, T.: Open Nesting in Software Transactional Memory. In: 12th Symposium on Principles and Practice of Parallel Programming, pp. 68–78 (2007)
14. Ponnusamy, R., Saltz, J., Choudhary, A.: Runtime Compilation Techniques for Data Partitioning and Communication Schedule Reuse. In: Conference on Supercomputing, pp. 361–370 (1993)
15. Rauchwerger, L., Padua, D.: The LRPD Test: Speculative Runtime Parallelization of Loops with Privatization and Reduction Parallelization. IEEE Transactions on Parallel Distributed Systems 10(2), 160–180 (1999)
16. Sagiv, M., Reps, T., Wilhelm, R.: Parametric Shape Analysis via 3-valued Logic. In: 26th Symposium on Principles of Programming Languages, pp. 105–118 (1999)

Statistically Analyzing Execution Variance for Soft Real-Time Applications

Tushar Kumar[1], Romain Cledat[2], Jaswanth Sreeram[2], and Santosh Pande[2]

[1] School of Electrical and Computer Engineering,
Georgia Institute of Technology, Atlanta, Georgia, USA
tushark@ece.gatech.edu
[2] College of Computing,
Georgia Institute of Technology, Atlanta, Georgia, USA
romain@cc.gatech.edu, jaswanth@cc.gatech.edu, santosh@cc.gatech.edu

Abstract. Certain high-performance applications like multimedia and gaming have performance requirements beyond reducing program execution time. These applications have repetitive components whose desired performance characteristics are more naturally expressed using soft real-time theory with its probabilistic guarantees. However, for large complex gaming and multimedia applications, programmers typically avoid real-time constructs as they significantly constrain how the programmer can express functionality. Instead, such applications are developed as monolithic programs using conventional languages like C/C++. Here the soft real-time behavior of the application becomes an emergent quality rather than being enforced by design. Programmers must then tweak parameters/algorithms until the application's soft real-time behavior becomes acceptable. There are currently no analysis techniques that directly extract the soft real-time execution characteristics of monolithic applications written without the use of real-time constructs. We introduce a domain-agnostic profiling methodology that identifies program execution-contexts whose variant behavior most significantly affects the soft real-time characteristics of the application.

1 Introduction

Important classes of high-performance applications, such as gaming and multimedia have performance requirements beyond minimizing program execution time. These applications have Quality-of-Service (QoS) requirements on repetitive application components, such as a live-video encoding application that attempts to maximally compress the input image stream while maintaining a sufficiently smooth frame-rate. Other examples include real-time object tracking and recognition kernels at the heart of many military and commercial applications.

Typically, games, streaming live-video encoders and video players attempt to maintain a reasonably high frame-rate for a smooth user experience. However, they frequently drop the frame-rate by a small amount and occasionally by a large amount if the computational requirements suddenly peak or compute resources get taken away. Therefore, the QoS requirements of such applications is

J.N. Amaral (Ed.): LCPC 2008, LNCS 5335, pp. 124–140, 2008.

best described using a combination of *i)* soft real-time theory with its probabilistic guarantees, and *ii)* the runtime sophistication of certain application-specific artifacts, such as the degree of compression achieved on a raw video stream, or the realism of Artificial Intelligence or physics modeling in games. During the design optimization stage, programmers like to tweak algorithms and parameter values in order to maximize the runtime sophistication of their application while minimally compromising the desired real-time characteristics. *In this paper, we exclusively focus on characterizing the soft real-time behavior of an application, in the absence of any knowledge about the application's functionality or its domain.* We limit our technique to informing the programmer about an application's soft real-time behavior, and leave it to the programmer to decide how best to tweak algorithms based on application and domain specific knowledge.

Monolithic Applications. Programmers often use specialized real-time languages and libraries to either guarantee that real-time requirements are met (hard real-time, for safety critical applications), or are met as close as possible (soft real-time). Such programming requires that the application be broken up into real-time constructs such as tasks, ordering dependencies be established between tasks, and completion deadlines (probabilistic or hard) be set on the tasks. However, when developing large applications like gaming and video, programmers typically eschew the benefits of formal real-time methods and languages, instead using conventional C/C++ development flows for their significant high-productivity advantages. The soft real-time behavior of the resulting monolithic application becomes entirely an *emergent quality* rather than being enforced by design. Programmers are then left to use ad-hoc means to understand what application components are responsible for undesirable soft real-time behavior.

In order to rectify the lack of suitable analysis tools for such applications, we propose a profile-driven methodology for characterizing the soft real-time behavior of *conventionally written monolithic* applications. The primary objective of our profiling methodology is the identification of application components that exhibit the *maximum variability* in their execution time. Such components are the ones most likely to affect the meeting of *implied* execution deadlines (such as desired frame-rates).

Application Structure. A soft real-time application typically processes a sequence of data items, such as a sequence of image-frames for MPEG video encoding. There are soft real-time requirements limiting the average execution time and variability in execution time for functions that process the data sequence. A programmer unfamiliar with the application stands to gain important insights about the application's design and functionality if the most significant functions processing the data sequence are pointed out. Our profile analysis framework automatically identifies those functions whose repetitious behavior most significantly contributes variance to their enclosing scopes. Consequently, the set of functions identified by the profile analysis can be expected to closely match the set of functions that process the data sequence. The primary intuition behind this reasoning is as follows: the application needs soft real-time requirements

to be enforced primarily because processing each data item, or parts of a data item, does not take constant time. Isovic, et al. [1] show that there is a significant amount of variation in decoding times for realistic video streams and argue that standard scheduling algorithms that assume average values and limited variation in frame decoding times will lead to poor video quality.

More generally, a soft real-time application may exhibit variability at many levels: from the highest level of processing a data item, to lower levels of processing pieces of the data item. Examples of this are image-frames at the highest level in video-encoding, and motion-estimation over 8-pixel × 8-pixel blocks of the image-frame. The execution-time of motion-estimation may vary dramatically from block-to-block even within the same image-frame, depending on how wide the motion-estimation searches to find a closely matching block. There is a large body of research showing how the search-space of motion-estimation can be limited based on the types of input video expected or the search-space dynamically adapted in order to more consistently achieve the desired frame-rate. Our profiling framework helps the programmer identify functions contributing significant variability at all levels of processing in the application, and empowers the programmer to make decisions about whether, where and to what extent algorithmic or configuration-parameter tweaking needs to be done, such as adjusting parameters that constrain the size of the motion-estimation search window.

1.1 Research Questions

We posed the following open research questions in order to drive the design of our profile analysis framework:

Question 1. **Component discovery** Can recurrent behavior identified during profiling of function calls be used to identify individual components of an application's soft real-time functionality?

Question 2. **Structure discovery** Can the identified recurrent behavior be used to reconstruct the soft real-time structure of the application? The structure would be composed of components of soft real-time functionality.

Question 3. **Context-sensitivity discovery** Can the context-dependent variability in the behavior of soft real-time components be detected? The behavior of a component may differ significantly depending on where it is invoked.

Question 4. **Generality** How reliably can behavior discovered during profiling be expected to generalize to future runs of the application on arbitrary inputs?

1.2 Contributions

We make the following specific contributions in this paper.

– We describe a tractable approach for succinctly capturing the behavior of millions of profile events in terms of tens of soft real-time components. The discovered components are functions that introduce significant variability

to the application's real-time behavior, and hence are most important to be brought to the attention of a programmer interested in improving soft real-time behavior.
- We demonstrate that function call-chain segments capture the context sensitivity of a component's soft real-time behavior. We motivate how the length of call-chain segments gives them varying ability to differentiate between multiple contexts of execution of a component. We provide algorithms that choose the correct segment lengths in order to produce highly succinct profile results that differentiate only between those contexts where behavior is significantly different.
- We illustrate the use of specific statistical theory for constructing algorithms that find *patterns* of behavior (dominant components and corresponding execution-contexts). Due to probabilistic guarantees provided by the statistical theory, the produced patterns generalize well for describing the behavior of the application executing on arbitrary input data.

We validate the correctness of the identified components by profiling well-known multimedia applications. Extensive prior research exists about the soft real-time behavior of these applications. The components reported by our profiling methodology match closely with those described in prior research as the main causes of soft real-time variance in these applications.

Among the questions listed above, only the structure discovery question is not satisfactorily answered by our current methodology. Although the discovered components and their call-chain contexts do allow the programmer to infer the structure, this inference is not sufficiently precise and may not work in all circumstances. In Section 6 we provide insights on how our technique can be improved to accommodate structure discovery as well.

Overview. Section 2 introduces the profile representation constructed from the raw stream of profile events. Section 3 introduces the relevant statistical theory and describes the analysis performed on the constructed profile representation for detecting patterns. Section 4 provides experimental validation.

2 Profile Representation

We profile-instrument the application and use the generated profile events to construct a Calling Context Tree (`CCT`). Ammons, et al. [2] showed that a `CCT` representation succinctly captures the dynamic structure of the function calls executed by the application. It preserves the full call-chain context of invocation, and merges information along multiple identical contexts into a single context. This makes the `CCT` an ideal representation for investigating context-sensitive behavior.

We automatically profile-instrument a C/C++ application using the LLVM [3] compiler infrastructure. We execute the application on test inputs and use the generated sequence of profile-events to construct the `CCT` as described in [2].

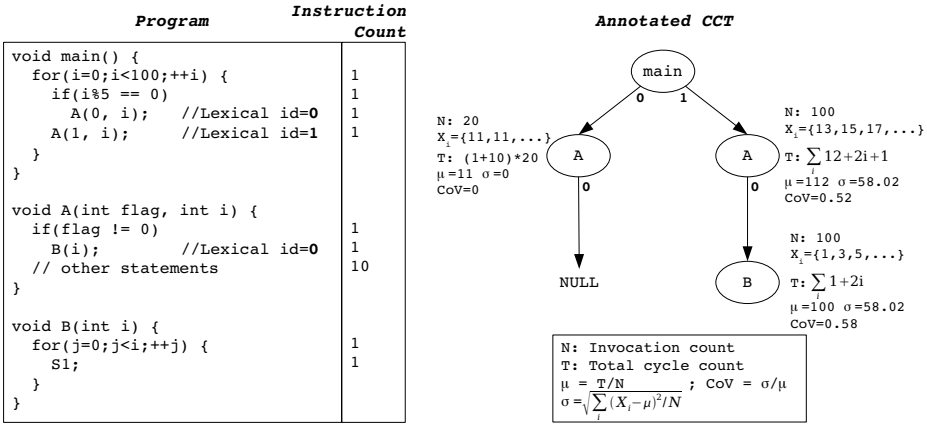


Fig. 1. Sample Program and CCT with Annotated Node Statistics

During `CCT` construction, we annotate statistics on each `CCT` node. These statistics are used by subsequent analysis for detecting patterns. Figure 1 shows an example program, the corresponding `CCT` and annotated node statistics. Function `A` was invoked from two call-sites within the parent function `main`. This leads to two children nodes for function `A`. Since function `B` was never invoked under the left `A` node, it only gets a *NULL* edge at its call-site in `A`. The next subsection describes the node annotations required for our variant call-context analysis.

2.1 Node Annotations

Each node is annotated with the following statistics about the execution of the function-call corresponding to it:

1. `invocation count` N`:` The number of times the corresponding function-call was invoked.
2. `mean` $\bar{X}$`:` The mean execution time across all invocations of the function-call corresponding to the node. *This includes the execution time of all children function-calls.*
3. `variance` σ^2`:` The statistical variance in the execution time of the function-call across all invocations. Variance is the square of the standard deviation σ. Using $\sigma^2 = E(X^2) - \bar{X}^2$, a single pass over the profile data constructs the `CCT` and computes all node annotations including variance.

2.2 Measuring Execution Time

We need to use some notion of elapsed time to time-stamp each function entry and exit event. Ideally, we would like to use wall-clock time with the application running on the target platform. Initially, we need to profile-instrument each function entry and exit since we don't know which ones will be significant for determining an application's variant behavior. However, this approach will suffer

such a large runtime overhead that wall-clock measurements will be rendered meaningless for capturing the application's real-time behavior.

In order to avoid introducing significant distortions to the application's real-time behavior, we chose to use dynamic-instruction-counts to estimate elapsed time. While measuring dynamic-instruction-count does not account for micro-architectural effects and system level stalls (such as cache misses), it does allow us to robustly compare execution times in the *order-of-magnitude* sense for function-call instances in the `CCT`. Our primary motivation is to determine which function call instances (`CCT` nodes) are highly variant with respect to their mean execution times, and which function call instances are orders-of-magnitude more variant than others. Time-stamping profile events with dynamic-instruction-counts suffices for this purpose, while at the same time remaining unaffected by the overheads of profile-instrumentation. The LLVM compiler pass inserts code to update a global counter for the dynamic-instruction-count. This counter is sampled when dumping profile events during program execution.

Once the pattern generation analysis has been performed using dynamic-instruction-counts to measure elapsed time, the function names that *cumulatively* (i.e., over all `CCT` instances of same function) consume significant execution time are known. In subsequent iterations of profiling, real wall-clock time can be used to dump profile events only for those functions that were previously seen to consume significant time cumulatively. This would dramatically lower the runtime overhead of profile-instrumentation since lower-level functions that do not affect the overall analysis would not get profile-instrumented at all. The resulting wall-clock measurements can then be expected to closely match the real-world execution timing of the application.

3 Detecting Patterns of Behavior

Once the `CCT` has been constructed and its node annotations calculated, the `CCT` is traversed in pre-order for analysis. Nodes whose total execution time constitutes a miniscule fraction (say, $< 0.02\%$) of the total execution time of the program and their children subtrees, are deemed as *insignificant*. All other nodes are deemed *significant*. Since `CCT` nodes subsume the execution time of their children nodes, once a node is found to be insignificant, the nodes in its children subtree are guaranteed to be insignificant as well.

Since insignificant nodes individually constitute a miniscule portion of the program's execution time, any patterns of behavior detected for them would quite likely provide very limited benefits in optimizing the design of the whole application. Therefore insignificant nodes are ignored from all further analysis. This dramatically reduces the part of the `CCT` that needs to be examined by any subsequent analysis, leading to considerable savings in analysis time. For each application, the programmer can experimentally tweak the cutoff percentage used to determine significant nodes. A good methodology for this would be to start with a relatively large cutoff threshold (say, 0.1%) and successively reduce it until the profiling results stabilize. Stabilization suggests that further inclusion of less significant nodes does not affect the analysis.

3.1 Tagging Nodes

We examine the annotations of nodes to determine if the corresponding nodes exhibit high-variance in execution-time within the context of the caller function (parent node). This is captured by the variance $P.\sigma^2$. We use Chebyshev's inequality [6] given below to determine meaningful thresholds to compare a node's variance. **Chebyshev's inequality** establishes conservative probability bounds on a given collection of data samples *while making no assumptions about the underlying probability distribution that generated the data.*

$$Pr(|X - \mu| \geq k\sigma) \leq \frac{1}{k^2} \tag{1}$$

In our experiments, we define a node to be high-variant if its execution time cannot be guaranteed to lie within 200% of its mean with atleast 96% probability. This implies $\frac{1}{k^2} = 1 - 0.96 = 0.04$ and $k\sigma = 2 \times \mu$. Therefore $\frac{\sigma}{\mu} \geq 0.4$ becomes the condition for high-variance. Consequently we use the Coefficient-of-Variability metric for classifying the variant nature of nodes: $CoV = \frac{\sigma}{\mu}$. The choice of the variance-window around the mean and the probability of samples falling within it can be tweaked by the user based on the method described above. As the programmer pushes the probability guarantee of samples falling within the $k\sigma$ variance window to 100%, $\frac{1}{k^2} \to 0$ and $k \to \infty$. This implies that the window $k\sigma \to \infty$ would trivially encompass all possible execution-times. Therefore, it is practical to keep the probability not too close to 100%, and for almost all soft real-time applications a probability guarantee of 96% would suffice, though this can be adjusted to the guarantees desired for any given application. Qualitatively, $k\sigma = 200\% \times \mu$ suggests a highly variant behavior as the execution-time can increase to over thrice the mean execution time (and reduce all the way down to 0). Based on how stringent the soft real-time requirements are for an application, the programmer can adjust the threshold that defines high-variant behavior.

Once the `CCT` is constructed from the profile data, it is pre-order traversed in linear time and individual nodes may be *tagged* as being *high-variant.* As mentioned earlier, the traversal is restricted to significant nodes.

3.2 Signature Generation for Patterns

The previous subsection described how significant nodes in the `CCT` were individually tagged if they exhibited statistical high-variance. The next step is to find *patterns of call-chains* whose presence on the call-stack can be used to predict the occurrence of the high-variance behavior found at the tagged nodes. For a given tagged node P, we restrict the call-chain pattern to be some contiguous segment of the call-chain that starts at `main` (the `CCT` root node) and ends at P.

The names of the sequence of function-calls in the call-chain segment become the detection pattern arising from the tagged node. This particular detection pattern might occur at other places in the significant part of the `CCT`. Quite possibly, the occurrence of this detection pattern elsewhere in the `CCT` does not match the statistical behavior, i.e., mean and CoV values, that were observed at

the tagged node. Therefore, our key criteria in generating the detection pattern is the following:

> All occurrences in the significant CCT of a detection pattern arising from a high-variance tagged node must exhibit the same statistical behavior that was observed at the tagged node.

Notice that this condition is trivially satisfied if we allow our detection pattern to extend all the way to `main` from the tagged node, since this pattern cannot occur anywhere else due to its full call-context being a unique path in the CCT. In many applications, patterns extending to `main` are likely to *generalize* very poorly to the *regression execution* of the application on arbitrary input data. Regression execution refers to the real-world-deployed execution of the application, as opposed to the *profile execution* of the application that produced the profile sequence used for constructing the CCT. In many applications, we expect the behavior of the function call at the top of the stack to be correlated with only the function-calls just below it in the call-stack. This short call-sequence would be expected to produce the same statistical behavior regardless of where it was invoked from within the program (i.e., regardless of what sits below it in the call-stack). In this paper we focus our attention on detecting just such call-sequences. We call these *Minimal Distinguishing Call Sequences* (MDC sequences) corresponding to any particular statistical behavior. These are the shortest length detection sequences whose occurrence predicts the behavior at the tagged node, with no false positive or false negative predictions in the CCT. A pattern with MDC is illustrated in Figure 2.

Given a tagged node P, Algorithm 1 produces the MDC sequence for P that is just long enough to distinguish the occurrence of P from the occurrence of any other significant node that has the same function-name as P but does not match the statistical behavior of P (the *other_set*). This is done by starting the MDC sequence with a call-chain consisting of just P, and then adding successive parent nodes of P to the call-chain until the MDC sequence becomes different from every one of the same length call-chains originating from nodes in the

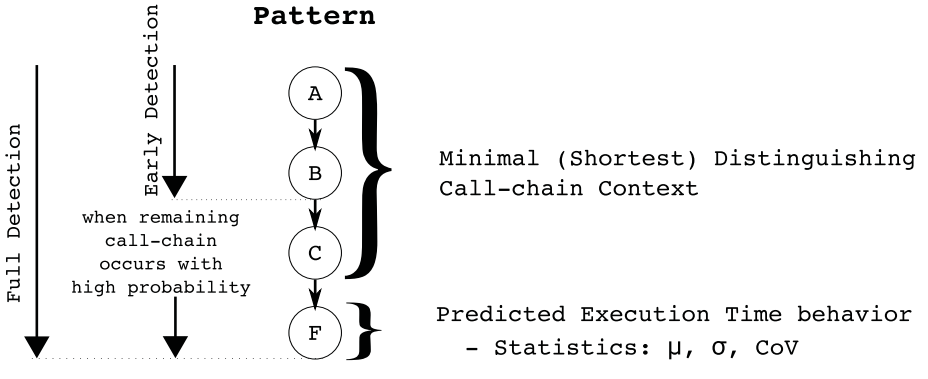


Fig. 2. Minimal Distinguishing Call-Context Pattern

other_set. Therefore, by construction, using steps 6 - 9 of Algorithm 1, the MDC sequence cannot occur at any CCT nodes that do not satisfy the statistics of P (matching mean and CoV). However, the same MDC sequence may still occur at multiple nodes in the CCT that *do* satisfy the statistics for P (at some nodes in the *match_set* in step 5). There is no need for P's MDC sequence to distinguish against these nodes as they all have the same statistics and correspond to the call of the same function as for P. Since all nodes in the *match_set* will have the same *other_set*, the algorithm is optimized to generate the *other_set* only once, and apply it for all nodes in the *match_set* even though only P was passed as input. The algorithm outputs the MDC sequence for each node in *match_set* (called the *Distinguishing Context* for P).

Algorithm 1. Minimal Distinguishing Call Sequence Generation

Input: CCT C, Tagged CCT Node P
Output: Distinguishing Context DC for P: set of pairs of form <MDC sequence, node of occurrence>

```
1  begin
2      DC ⟵ ∅;
3      func_name ⟵ Name of function corresponding to node P;
4      all_set ⟵
       get_all_significant_node_occurrences_of_function_in_CCT(func_name, C);
5      match_set ⟵ identify_all_nodes_with_matching_statistics(P, all_set);
6      other_set ⟵ all_set − match_set ; // identify nodes with same name
       whose statistics don't match P's
7      for each CCT node m in match_set do
8          MDC ⟵ [< func_name, lexical-id of m in its parent >] ; // initialize
           MDC as call-chain of length 1
9          Extend MDC sequence with parent nodes of m (and their lexical-ids) until
           the detection pattern MDC is different from call-chains of same length
           arising from every node in other_set;
10         DC ⟵ DC ∪ {< MDC, m >};
11 end
```

3.3 Grouping and Distinguishing between Similar Patterns

In the previous discussion, we assumed that the programmer desired to distinguish between tagged nodes whenever their statistics (mean, CoV) did not match exactly. However, exact matching of statistics may lead to very long detection patterns that generalize poorly to regression runs. For example, if multiple high-variant tagged nodes with very different means require long call-chains to distinguish between each of them, then it may be preferable to actually have a shorter call-chain pattern that does not distinguish between the tagged nodes.

Furthermore, if the same detection sequence occurs at multiple tagged nodes in the significant CCT and the nodes have matching statistics, we would like to combine the multiple occurrences of the detection sequence into a single detection

sequence. Such detection sequences are likely to generalize very well to the regression run of the application, and are therefore quite important to detect.

In order to address the preceding two concerns in a unified framework, we first use Algorithm 1 to generate short patterns using only the "broad-brush" notions of high-variance, without distinguishing between tagged nodes using their statistics (mean, CoV). Then we group patterns with identical call-contexts (arising from different tagged nodes) and use *pattern-similarity-trees* (`PST`) to start differentiating between them based on their statistics. The initial group forms the root of a `PST`. We apply a *Similarity-Measure* (`SM`) function on the group to see if it requires further differentiation. If the patterns in the group have widely different means or CoVs, and the programmer wants this to be a differentiating factor, then the similarity check with the appropriate `SM` will fail. In our experimental evaluation, we use an `SM` that checks if the corresponding means and CoVs of the two patterns being compared are within 10% of each other; the programmer can choose to plug in a different `SM`, say one that checks only on means.

Once the `SM` test fails on a group, all the patterns in the group are extended by one more parent function from their corresponding call-chains (tagged `CCT` nodes are kept associated with patterns they generate). This will cause the resulting longer patterns to start to differ from each other. Again identical longer patterns are grouped together as multiple children groups under the original group. This process of tree-subdivision is continued separately for each generated group until the `SM` function succeeds in all current leaf nodes. At this point, each of the leaf groups in the `PST` contain one or more identical patterns. The patterns across different leaf groups are however guaranteed to be different in some part of their prefixes. Patterns in different leaf groups may be of different lengths, even though the corresponding starting patterns in the root `PST` node were of the same length. All the identical patterns in the same leaf-node are collapsed into a single detection-pattern. For example, an `SM` function that differentiates on σ but not on means (or only weakly on means), will produce leaf nodes that contain patterns with a single σ but a collection of widely varying means.

3.4 Ranking Impact of Patterns

The previous steps produce numerous patterns (11 to 46 patterns for our benchmarks) characterizing the variability in the application at multiple levels. It is highly desirable to rank the patterns in order to focus the programmer's attention on the ones that are most likely to contribute variability to the program. For this purpose we introduce a metric that we call the *Variability Impact Metric* or `VIM`. The Chebyshev inequality introduced earlier points us towards a suitable definition for `VIM`. While the CoV $= \frac{\sigma}{\mu}$ indicates whether the variations are large with respect to the mean, the $k\sigma$ term in the Chebyshev inequality indicates the absolute amount of variability. The variability per invocation multiplied by the total invocation count of that pattern gives the total amount of variability contributed by the innermost function in the pattern to its immediate parent.

Therefore, we define VIM as follows, with N being the invocation count of the innermost function in the pattern:

$$\texttt{VIM} = k\sigma N \tag{2}$$

While this metric indicates how much variance is contributed by the innermost function F to its immediate parent C (referring to the pattern in Figure 2), it is not necessarily implied that the pattern's variance contribution propagates up the call-chain to A or B. For example, if B invokes C from inside a loop, then the VIM for C will measure the variance impact to iterations of the loop, not to B directly. In fact, it is possible that B is not variant at all if each iteration of C consumes correspondingly lower time if the loop-count is high, and vice-versa when the loop-count is low, leading to a constant execution-time for the loop across all invocations of B. A similar situation can occur without loops if B invokes C inside a very infrequently executed branch.

Despite the limitation described above, the profile analysis technique is excellent for *eliminating* unlikely contributors of variance. Therefore, the correct way to interpret the produced patterns is to think of them as *highly likely contributors* of variance. This immediately allows the programmer to narrowly focus on very limited parts of the application in order to identify the causes of violations to the soft real-time requirements. The programmer would of course have to examine relative invocation counts along a given pattern's call-chain to infer how far up the call-chain an innermost function is likely to be contributing variance.

4 Experimental Evaluation

The Statistical Profile Analysis tool has been written in Python. We did not use any high-performance numerical or scientific libraries (such as NumPy, SciPy) in the Python implementation. We profile instrumented a number of applications in the MediaBench II Video suite and a real-time object-recognition benchmark (mimas-findTux) from the Mimas Computer Vision application-suite [18]. We generate profile data (sequence of profile events) for each benchmark using the input data sets provided with the benchmark suites, or some larger external data sets if the profiles are too short. Specifically, we use two different input data-sets for each benchmark, referred to as D1 and D2.

We run profile analysis on D1 to create patterns and then use D2 for the regression run that we use to validate the statistics of the patterns found previously. The regression run simulates the application call-stack using the profile events. No CCT is constructed and no analysis is performed in the regression phase. We use a generic finite-state-machine sequence detector to detect the occurrence of the patterns at the top-of-the-stack. Such a sequence detector needs to check the call-stack for the possible occurrence of every pattern on every profile event. This is the cause of the significant slow-down seen in Table 1 in the pass times for the regression runs compared to the profile runs. *We would like to emphasize that the profile analysis time consists entirely of the time to read and parse the*

Table 1. Patterns Found in Benchmarks

Benchmark	Profiling on D1: Pat. Generation				Regression on D2			
	# of steps	*Pass time (seconds)*	*# of patterns*	*Pattern Set size*	*# of steps*	*Pass time (seconds)*	*Pattern Set size*	*Pat Set overlap*
H.263enc	30000000	397	9	7	60000000	1245	7	100%
H.263dec	25000000	341	30	5	60000000	2194	6	100%
findTux	30000000	402	60	3	40000000	2833	3	100%
mpeg2enc	30000000	387	44	5	60000000	2943	5	100%
mpeg2dec	30000000	402	20	5	60000000	1657	5	80%

profile file from disk. The actual time for all of the analysis combined (variance tagging, minimum call-context detection, etc) consumes a fraction of a second.

Table 1 shows the length of the D1 profile (in terms of number of `entry` / `exit` events) used to generate patterns, the number of high-variance patterns found, and the length of the D2 profile used during regression to simulate the real-world execution of an application. The *Pass Time* refers to the duration of time needed to complete profile analysis or regression.

Clearly during regression the input data set is different, which will lead to corresponding changes in the call-chains invoked, their frequencies and their variant behaviors. However, in our validation we strive to demonstrate that the patterns capture the statistical behavior of the application at a more fundamental level, which tends to remain relatively constant across different data-sets. In order to demonstrate this, we introduce the notion of a *Pattern Set* both for Profiling and Regression. We define the Pattern Set to consist of a subset of patterns that are found to be most impactful as measured by their Variability-Impact-Metric (VIM). Specifically, we limit the Pattern Set to only those patterns whose VIM is atleast 10% of the VIM of the pattern with the highest VIM. This is done separately for Profiling and Regression, leading to the construction of two potentially disjoint sets. Table 1 shows that in fact the Regression Pattern Set very closely mirrors the Profiling Pattern Set (*Pattern Set Overlap* column). This implies that the same set of patterns that were found to be most impactful during Profiling tend to remain most impactful during Regression. The Pattern Set spans an order-of-magnitude of the largest VIM values (i.e., 10×). We chose to define the Pattern Set as such because we expect the data-set induced variations to cause relative fluctuations due to changes in length of data (number of events) and type of data (for example, encoding video with constant background versus moving background, different frame-dimensions, etc). Despite these variations in characteristics of input data, the most impactful patterns found on D1 tend to remain most impactful on D2 as well, validating our intuition that our patterns capture variant behavior in a statistically sound manner. In mpeg2dec, the VIM of one pattern was just slightly smaller during regression causing it to be dropped from the Regression Pattern Set. Similarly, a pattern that had barely missed inclusion in the Profiling Pattern Set got included in the Regression Pattern Set. However, both these patterns have similar VIMs (in the order-of-magnitude of sense). Therefore, despite a Pattern Set Overlap of only 80%, this result also

shows that Profiling and Regression Pattern Sets match closely for mpeg2dec. In H.263dec, there was a pattern that barely missed inclusion in the Profiling Pattern Set, but got included in the Regression Pattern Set.

Figure 3 shows the distribution of the mean and CoV values for all the patterns discovered, on a per-benchmark basis. For each benchmark, the left-segment in the scatter-plot shows the distribution found during Profiling (on `D1`), and the right-segment shows during Regression (on `D2`). No `VIM` based distinction is made between patterns; the least varying pattern with low invocation-count is given a point just like the most impactful pattern. For all benchmarks the distributions between Profiling and Regression are very similar, except for a uniform linear shift and uniform scaling of one distribution with respect to the other. When we look at Figure 4 plotted using only patterns in the Profiling and Regression *Pattern Sets*, we again see a close similarity between Profiling and Regression distributions, indicating that the dominant patterns are fundamentally associated with the application behavior, regardless of data-sets. For example, encoding raw video with a larger image frame-size quadratically increases the mean and possibly the CoV of a motion-estimation pattern, but motion-estimation remains dominant independent of the image frame-size.

The following is representative across the benchmarks of the *compaction of information* achieved in going from raw profile data to the final profile results: 800MB to 1.3 GB of raw profile event data reduced to a `CCT` with 600 to 800 nodes, out of which 200 to 350 nodes were found significant, out of which 16 to 116 nodes were tagged high-variant, which were grouped down to 9 to 60 patterns with identical contexts and similar means and CoVs (using pattern similarity tree), finally out which 3 to 7 were dominant patterns (pattern set).

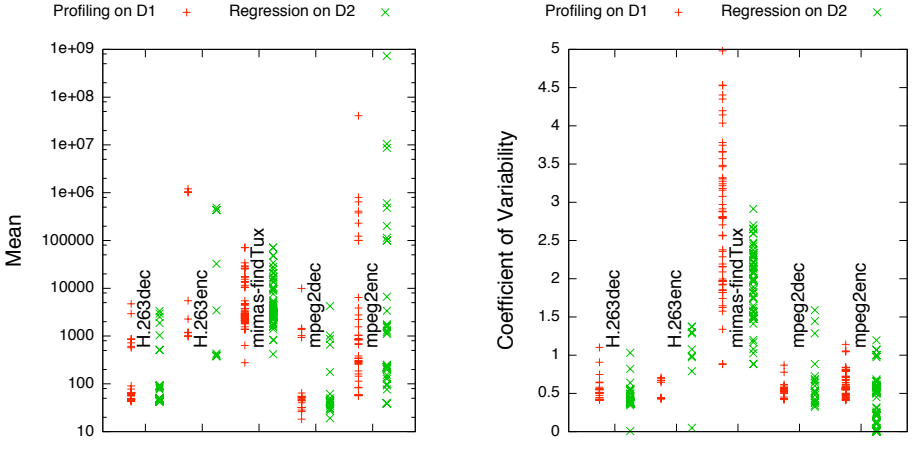


Fig. 3. Comparison of *mean* and *CoV* scatter-plots between Profiling `D1` and Regression `D2` using **all patterns**

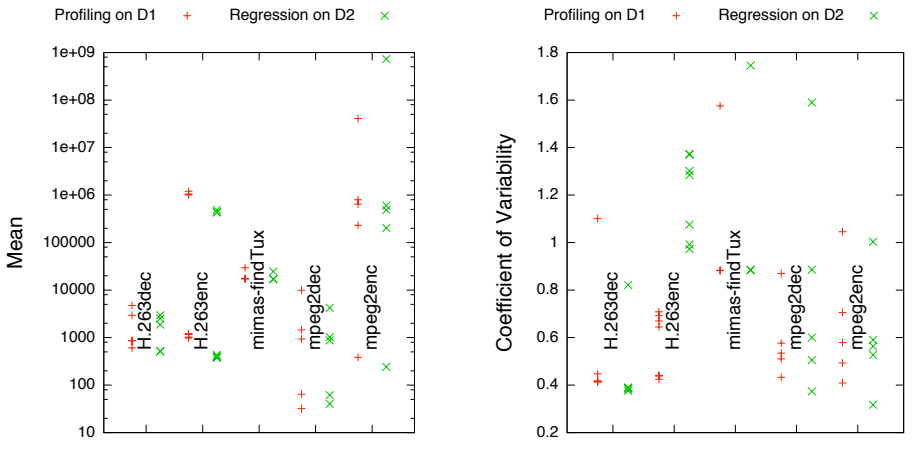


Fig. 4. Comparison of *mean* and *CoV* scatter-plots between Profiling `D1` and Regression `D2` using **Pattern Set**

4.1 Case Study: H.263enc

Figure 5 shows the Profiling Pattern Set for the H.263enc benchmark, sorted from the most impactful to the least. The `VIM` found for each pattern is shown for the Profiling and Regression phases. Function-names are shown in boxes and the edge-annotations give the `lexical-id` (lexical position) of the call-site of the callee (sink of arrow) within the body of the caller (source of arrow). The italicized number on top of each box gives the number of times the corresponding function was invoked as part of the pattern. A pattern's invocation-count corresponds to the invocation-count of the function in the left-most box. This is the innermost function of the pattern, and the entire pattern occurs only when the entire call-chain segment occurs on the stack. Therefore, the invocation-count of the innermost function is the invocation count of the pattern.

The patterns in Figure 5 were automatically discovered by the profile analysis framework with no guidance from the user, and no application or domain knowledge. Yet, these patterns closely mirror conventional wisdom about the parts of video-encoding applications that are the most important with regards to meeting or violating soft real-time requirements. Motion-estimation related macroblock search-spaces are known to be the most variant parts of video encoding [16], since the search space can be quite large and it is hard to know up front how quickly the search will terminate.

Note that the middle three patterns and the bottom three patterns are identical except for a difference in `lexical-ids`. In both cases, the multiple identical patterns have very similar statistical characteristics (`VIM`s and also from their positions in the scatter plots). These could have been combined into a single pattern in both cases, but our analysis framework distinguishes based on `lexical-ids` within patterns. The downside here is having three patterns where one would

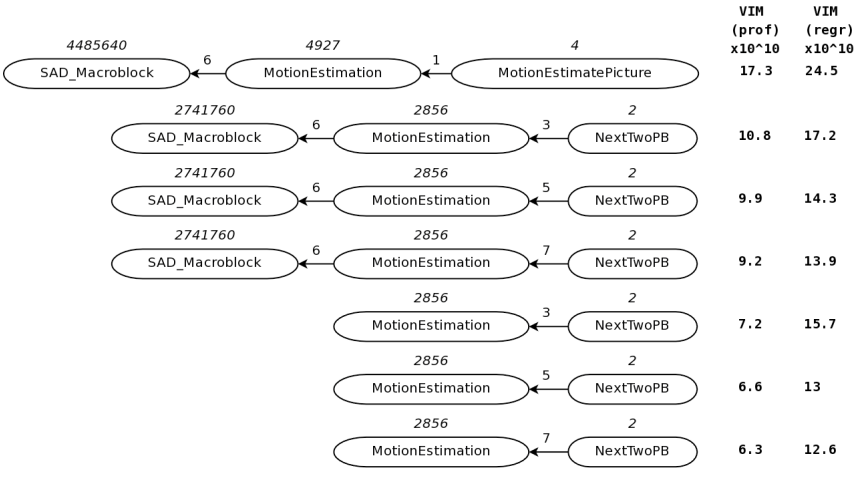


Fig. 5. Pattern Set for H.263enc

suffice, but in general this produces greater resolving power between identical call-chains whose behavior varies between call-sites, such as with mpeg2enc.

5 Related Work

Existing application profiling techniques look for program hot-spots and hot-paths [4,5,9]. These techniques attempt to find performance bottlenecks in an application, and do not attempt to identify variant behavior impacting an application's soft real-time characteristics.

Calder et al. have used statistical techniques to characterize large scale program behavior using few recurrent intervals of code [7] and to find phase change points in the dynamic execution of a program [8]. However, their work was not intended for mining soft real-time characteristics of an application, and cannot be adapted for such. In particular, they seek out intervals in [7] with closely matching set of dynamic basic-blocks, whereas we seek out call-contexts where the same function exhibits highly variant execution time.

Variability Characterization Curves (VCCs) and Approximate VCCs [10] have been used to characterize the variability in the workloads of multimedia applications. Such analysis techniques require domain-specific knowledge of the application before they can be applied. Similarly, there are custom techniques for improving the QoS of each type of application, such as by Roitzsch et al [15] that develop a higher-level representation model of a generic MPEG decoder, and based on this predict video decoding times with high accuracy. In contrast, our framework characterizes the variant behavior in the application in a completely domain-independent manner, with no assistance from the user.

For applications written using real-time constructs/formalisms such as tasks and deadlines, there is an established body of formal techniques [11,12] that analyze or ensure the real-time characteristics of the application. For monolithic applications written without the use of these abstractions, our framework is unique in its ability to characterize their soft real-time behavior.

Worst-Case-Execution-Time (WCET) [13] is an analysis methodology applicable to monolithic applications, and has been incorporated into commercial products such as from AbsInt [17]. However, for non-safety-critical, compute-intensive applications like gaming and video, knowledge of the *likely range* of real-time behavior is more important for driving design optimization than knowledge of worst case behavior. The likely range (detected by our technique) can be substantially removed from the worst case, thereby diminishing the value of characterizing the worst case behavior for such applications.

In contrast with prior work [14] on identifying variant behavior in monolithic applications, the techniques in this paper establish statistically robust probability bounds on variant behavior and produce concise results prioritized by their impact on soft real-time behavior.

6 Conclusion

In this paper we demonstrated that analyzing a profile sequence of time-stamped function entry and exit events can be used to *i)* identify the dominant soft real-time components of functionality in an application, *ii)* determine the context-sensitivity of the behavior of the identified components, and *iii)* concisely convey the components and their context sensitivity to the programmer using patterns consisting of minimal-length call-chain segments. Further, we established that the dominant patterns detected during profile analysis continue to remain dominant in regression runs of the application on input data sets that have different characteristics (differences in frame-dimensions, encoding format, degree of motion in input videos). This experimental validation coupled with a sound foundation of our algorithms in statistical theory suggests that our analysis detects fundamental aspects of an application. Lastly, we find that patterns identified by our profile analysis in well-known multimedia applications correspond closely with extensive prior research studying the causes of soft real-time variance in these applications. In conclusion, our technique concisely captures the true soft real-time characteristics of a monolithic application, for which existing real-time analysis techniques are inapplicable.

Future Work. Call-chain segments were found useful in defining contexts for components, but call-chains do not sufficiently capture which components contain other components, and more importantly, whether contained components contributed significant variance to any parent component. We are currently developing a concise hierarchical representation for capturing the cause-effect and containment structure between components.

References

1. Isovic, D., Fohler, G., Steffens, L.: Timing constraints of MPEG-2 decoding for high quality video: misconceptions and realistic assumptions. In: ECRTS 2003 (2003)
2. Ammons, G., Ball, T., Larus, J.R.: Exploiting hardware performance counters with flow and context sensitive profiling. In: PLDI 1997 (1997)
3. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: CGO 2004 (2004)
4. Duesterwald, E., Bala, V.: Software profiling for hot path prediction: less is more. SIGPLAN Not 35(11), 202–211 (2000)
5. Hall, R.J.: Call path profiling. In: ICSE 1992 (1992)
6. Freund, J.E., Walpole, R.E.: Mathematical statistics, 4th edn.
7. Sherwood, T., Perelman, E., Hamerly, G., Calder, B.: Automatically characterizing large scale program behavior. SIGOPS Oper. Syst. Rev. 36(5), 45–57 (2002)
8. Lau, J., Perelman, E., Calder, B.: Selecting Software Phase Markers with Code Structure Analysis. In: CGO 2006 (2006)
9. Arnold, M., Hind, M., Ryder, B.G.: Online feedback-directed optimization of Java. In: OOPSLA 2002 (2002)
10. Liu, Y., Chakraborty, S., Ooi, W.T.: Approximate VCCs: a new characterization of multimedia workloads for system-level MpSoC design. In: DAC 2005 (2005)
11. Lin, C., Brandt, S.A.: Improving Soft Real-Time Performance through Better Slack Reclaiming. In: RTSS 2005 (2005)
12. Wandele, E., Thiele, L.: Abstracting functionality for modular performance analysis of hard real-time systems. In: ASP-DAC 2005 (2005)
13. Gustafsson, J., Ermedahl, A., Sandberg, C., Lisper, B.: Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis Using Abstract Execution. In: RTSS 2006 (2006)
14. Kumar, T., Sreeram, J., Cledat, R., Pande, S.: A profile-driven statistical analysis framework for the design optimization of soft real-time applications. In: ESEC-FSE 2007 (2007)
15. Roitzsch, M., Pohlack, M.: Principles for the Prediction of Video Decoding Times Applied to MPEG-1/2 and MPEG-4 Part 2 Video. In: RTSS 2006 (2006)
16. Girod, B., Steinbach, E., Färber, F.: Performance of the H.263 Video Compression Standard. J. VLSI Signal Process. Syst. 17, 2–3 (1997)
17. `http://www.absint.com`
18. `http://www.shu.ac.uk/research/meri/mmvl/research/mimas/`

Minimum Lock Assignment: A Method for Exploiting Concurrency among Critical Sections

Yuan Zhang[1], Vugranam C. Sreedhar[2], Weirong Zhu[3,⋆], Vivek Sarkar[4], 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
vugranam@us.ibm.com
[3] Microsoft Corporation, Redmond, WA
weirong.zhu@microsoft.com
[4] Rice University, Houston, TX
vsarkar@rice.edu

Abstract. In this paper we propose a lock assignment technique to simplify the mutual exclusion enforcement in multithreaded programs. Programmers are allowed to annotate the regions of code that are expected to be mutually exclusive as `critical` sections, without using explicit locks. The compiler then automatically infers an assignment of the minimum number of locks to critical sections by solving the Minimum Lock Assignment (MLA) problem so as to enforce mutual exclusion without any loss of concurrency. We show that the MLA problem is NP-hard. We have proposed a heuristic to solve the MLA problem, and tested the optimality of the heuristic with the Integer Linear Programming (ILP) solver. We have also tested the efficiency of the heuristic using scientific applications, from which we obtain up to 30% performance gain with respect to the programs in which all critical sections are controlled by a single lock.

1 Introduction

Given that the processors in current and future computer systems are becoming multi- or many-core by default, it is important to address the performance and productivity issues in multithreaded programming. One of the major performance and productivity issues in multithreaded programming arises from enforcing the mutual exclusion (mutex for short) using lock/unlock operations. Programmers explicitly assign lock variables to control mutex regions, and the lock variables are acquired by the executing thread before the mutex region is executed, and are released after the execution of the mutex region completes. Explicitly managing multiple locks is error prone since it is easy for programmers to introduce data races and create deadlocks. Alternatively, programmers

⋆ The author participated this work when he was a graduate student in the University of Delaware.

J.N. Amaral (Ed.): LCPC 2008, LNCS 5335, pp. 141–155, 2008.

can use a single lock to control all mutex regions to avoid deadlocks and data races. However, they lose concurrency among mutex regions by unnecessarily serializing them.

In this paper, we propose a lock assignment technique to simplify the enforcement of mutual exclusion in multithreaded programs. We allow the programmers to annotate regions of code that are expected to be executed mutually exclusively as `critical` sections, without managing explicit locks. The compiler then automatically infers an assignment of multiple compiler-managed locks to critical sections (possibly multiple locks for one critical section) to preserve the mutual exclusion and also exploit the concurrency among critical sections.

A naive lock assignment approach associates one lock to each shared memory location, and the lock set of a critical section is the set of locks assigned to memory locations it accesses. This approach, however, may use more locks than necessary, and introduce excessive overhead on lock acquisition and release. To control the locking overhead, we would use the minimum number of locks which is necessary to preserve the mutual exclusion and fully exploit the concurrency between critical sections. We formulate this lock assignment task as the **Minimum Lock Assignment** (MLA) problem:

Problem 1 (Minimum Lock Assignment). Given a multithreaded program with a set of critical sections, find the minimum number of distinct locks that are needed for controlling the critical sections such that

(a) Two critical sections are assigned disjoint sets of locks if (1) they are concurrent and (2) they do not access any common location, or if they access a common location then none of them writes to the common location.

(b) Two critical sections are assigned at least one common lock if (1) they are concurrent and (2) they access some common location and at least one of them writes to the common location.

Note that a critical section can be assigned a set of locks. The semantics of a lock set follows the strict two-phase locking policy [1].

The solution of the MLA problem consists of two main phases: the analysis phase and the lock assignment phase. In the analysis phase, the compiler reads the multithreaded program and statically determines whether a pair of critical sections are interfering. Two critical sections are interfering if they are concurrent and they access some common shared memory location(s), with at least one of them writes to the common location(s). In the lock assignment phase the compiler calculates the minimum number of locks to control critical sections according to the analysis result, and assigns one or more locks to each critical section. Besides, the runtime system guarantees that an execution is deadlock free by acquiring and releasing locks in a predetermined order. The analysis phase is solved by concurrency analysis, data set analysis and pointer analysis. However, due to the space limitation, in this paper we simply assume the analysis result is already calculated, and we only focus on the lock assignment phase. Readers can refer to [1,2] for more details on the analysis phase and deadlock

avoidance options. In the following discussion we refer to the MLA problem as the lock assignment phase exclusively, without any further clarification.

The rest of the paper is organized as follows. In Section 2 we introduce the concurrency graph as the main data structure to solve the MLA problem. In Section 3 we prove that MLA problem is related to the graph coloring problem and it is NP-hard. We then present a heuristic to solve the MLA problem. We also formulate the MLA problem as an Integer Linear Programming (ILP) problem. In Section 4 we evaluate our heuristic by comparing it with optimal solutions produced by the commercial ILP solver CPLEX. In 300 randomly generated testing cases we observe that our MLA heuristic is optimal for 83.3% of them. We also test the performance of our heuristic using a 10-way Sunfire machine on a set of Splash2 [3] benchmarks, and obtain up to 30.17% performance speedup with respect to programs in which all critical sections are controlled by a single lock. Related work is presented in Section 5, and finally we conclude in Section 6.

2 Concurrency Graph and Critical Sections

In this section, we introduce the *concurrency graph* to model the potential concurrency and interference among critical sections in a multithreaded program.

2.1 Concurrency Graph

Definition 1. A **Concurrency Graph** is an undirected graph $G = (V, E)$, in which: a vertex $v \in V$ denotes a textual critical section, and there is an edge $(u, v) \in E$ if instances of critical sections u and v may be concurrent.

In the above definition, if two instances of the critical section u are concurrent, we do not introduce a self-loop on u, since we will assign at least one lock to each critical section, and the mutual exclusion of u with respect to itself is self preserved. As an example, Figure 1(b) illustrates the concurrency graph for the program shown in Figure 1(a). The set of shared memory locations that are accessed within critical sections are also listed within curly braces in Figure 1(b).

Two concurrent critical sections are said to be **non-interfering** if either they do not access a common location or if they access a common location then

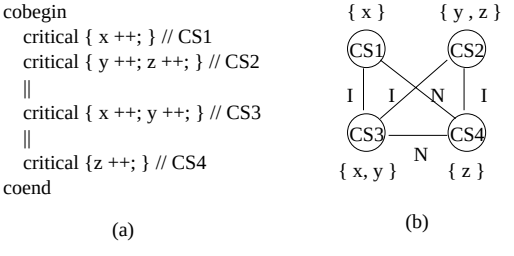


Fig. 1. (a) Example program (b) Concurrency graph

none of them writes to the common location. Two concurrent critical sections are **interfering** if they access some common location and at least one of them writes to the common location. We extend the concurrency graph defined in Definition 1 by labeling an edge (u, v) with label I when critical sections u and v are interfering, and with label N when u and v are non-interfering.

Note that a general concurrency graph may be a forest of connected graphs, and we analyze each connected component independently. In the following discussion, we simply assume that a concurrency graph G is a connected graph.

2.2 Non-interfering Concurrency Graphs

Consider a class of multithreaded programs P_n whose corresponding concurrency graph contains only non-interfering edges. Since all incident edges of a critical section are non-interfering, it cannot share any lock with its neighbors. This implies that whenever two critical sections are connected (concurrent), they require different locks. We can now rephrase the MLA problem (Problem 1) for non-interfering concurrency graphs as follows:

Problem 2. Given a program with a set V_n of non-interfering critical sections, find the minimum number of locks that can be assigned to critical sections such that if two different critical sections in V_n are concurrent then they get different locks.

The above problem is equivalent to the classical graph coloring problem — color the vertices (critical sections) of a graph using the minimum number of colors (locks) such that no two adjacent (concurrent) vertices (critical sections) are given the same color (lock). The MLA problem for this special class of programs is NP-hard[1].

2.3 Interfering Concurrency Graphs

Consider a class of programs P_i, for which the concurrency graph contains only interfering edges. In this case, two critical sections are either concurrent and interfering, or are not concurrent (not connected). If they are concurrent and interfering, they should share at least one common lock to preserve the mutual exclusion, which implies that they must be serialized. If they are not concurrent, they are already serialized. Therefore, in this interfering special case, there is no inherent concurrency, so we can use a single lock to control all critical sections without introducing any performance penalty.

2.4 Concurrency Graph Partition

In general cases, a concurrency graph contains both non-interfering edges and interfering edges. Given a concurrency graph $G = (V, E)$, let E_n denote the set

[1] For certain classes of graphs, such as the interval graphs, the graph coloring problem can be solved in polynomial time. However, the general concurrency graphs are not necessarily interval graphs.

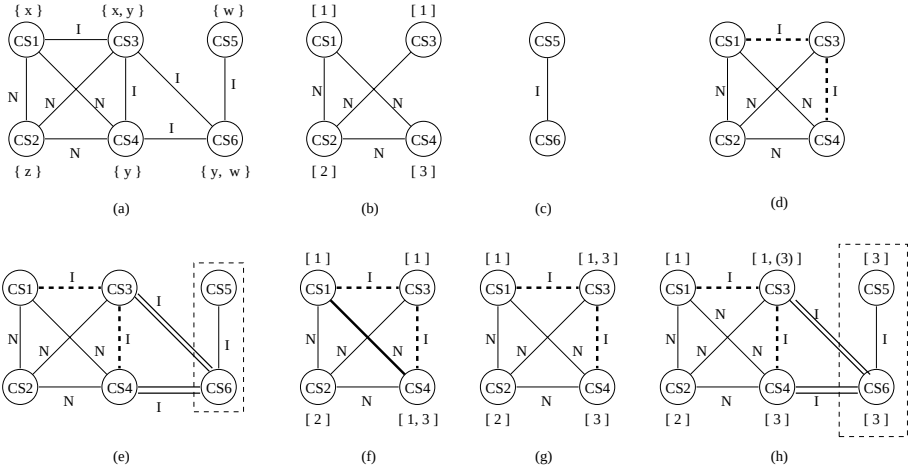


Fig. 2. (a) A general concurrency graph (b) The non-interfering subgraph G_n (c) The interfering subgraph G_i (d) The SNIG G_n^s (e) The crossing edges (double lines), serializing interfering edges (dotted lines), and the interfering subgraph (in dotted box) (f) A un-safe borrowing from CS_3 to CS_4 (g) A safe borrowing from CS_4 to CS_3 (h) Final lock assignment result

of non-interfering edges and E_i denote the set of interfering edges in G, such that $E = E_n \cup E_i$ and $E_n \cap E_i = \emptyset$. Let $G_n = (V_n, E_n)$ be the *non-interfering subgraph* induced by E_n, where $V_n \subseteq V$ such that a vertex $v_n \in V_n$ has at least one non-interfering edge incident on it. Figure 2(b) illustrates the non-interfering subgraph of Figure 2(a). Let $G_i = (V_i, E_i')$ be the *interfering subgraph* induced by vertices V_i, where $V_i = V - V_n$ and $E_i' \subseteq E_i$ is a set of interfering edges (u_i, v_i) such that $u_i, v_i \in V_i$. Figure 2(c) illustrates the interfering subgraph for Figure 2(a). Finally, let $E_i'' = E_i - E_i'$ be a set of interfering edges that are not in G_i. Some of interfering edges in E_i'' connect vertices of the non-interfering subgraph, for example, edges (CS_1, CS_3) and (CS_3, CS_4), as illustrated as bold dashed lines in Figure 2(d). We call such interfering edges that occur inside a non-interfering subgraph as *serializing* interfering edges E_s, because they could "serialize" the inherent concurrency that exists within non-interfering subgraph. The remaining interfering edges $E_{ci} = E_i'' - E_s$ are *crossing edges* between vertices in G_n and G_i. In the example shown in Figure 2(a), $E_{ci} = \{(CS_3, CS_6), (CS_4, CS_6)\}$, illustrated as double solid lines in (e). Besides the non-interfering subgraph G_n and the interfering subgraph G_i, we introduce the notion of the *serializing non-interference graph* (SNIG) as the non-interfering subgraph with serializing edges, $G_n^s = (V_n, E_n \cup E_s)$. Figure 2(d) illustrates an example of SNIG. SNIGs have some interesting properties that will influence the lock assignment.

2.5 Serializing Non-interference Graph

Let us consider a class of concurrency graphs called *Serializing Non-Interfering Graphs* (SNIGs). A SNIG consists of only non-interfering edges and serializing

interfering edges (as defined in the previous section). Serializing interfering edges constrain the inherent concurrency in a non-interfering concurrency graph. They also constrain the minimum number of locks required to color a SNIG.

The following observation states that sometimes it is impossible to color a SNIG if a vertex can be assigned only one color.

Observation 1. It is impossible to color an arbitrary SNIG with the following conflicting constraints:

1. Each vertex gets only one color,
2. If vertices u and v are connected by a non-interfering edge then they are given two different colors,
3. If two vertices u and v are connected by a serializing interfering edge then they are given the same color.

Consider the SNIG in Figure 3. Assume we satisfy all above constraints, then all critical sections get the same lock, because they are connected by serializing interfering edges (CS_1, CS_3), (CS_3, CS_4) and (CS_4, CS_2). However, the constraint (2) requires that CS_1 and CS_2 are given two different colors, a contradiction. Therefore Figure 3 cannot satisfy all three constraints.

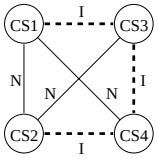


Fig. 3. Example SNIG for Observation 1

There are two ways to handle the above impossibility: relax constraint (1) in the above observation, or relax constraint (2). By relaxing constraint (1) we are allowed to assign multiple colors to each vertex. By relaxing constraint (2) we will reduce the concurrency. Constraint (3) must be satisfied since otherwise the mutual exclusion will be violated. In the MLA solution we will take the approach of assigning multiple locks so as to maximize the concurrency.

Let $C(x)$ be the set of colors that are assigned to a vertex u, the coloring problem on SNIG is stated as the following:

Problem 3. Given a SNIG $G_n^s = (V_n, E_n \cup E_s)$ find the minimum number of colors to color G_n^s such that:

(a) If two vertices u and v are connected by a non-interfering edge then $C(u) \cap C(v) = \emptyset$ and

(b) If two vertices u and v are connected by a serializing edge then $C(u) \cap C(v) \neq \emptyset$.

Let G be an arbitrary concurrency graph, and let G_n^s be the SNIG of G. We will show in Section 3.2 that the minimum number of locks required by G equals the minimum number of locks required by G_n^s.

3 Minimum Lock Assignment Solution

The MLA problem for arbitrary concurrency graphs is NP-hard because one special case - MLA problem for non-interfering concurrency graph - is NP-hard. In this section we present a heuristic approach for solving MLA. We also formulate the MLA problem as an Integer Linear Programming (ILP) problem, and in Section 4 we will use this ILP formulation to quantitatively evaluate our heuristic.

3.1 A Naive Solution

Assume all shared memory locations that a critical section accesses can be statically identified by compiler analysis, then a simple solution to the MLA problem is to assign a distinct lock to each shared memory location, and the lock set of a critical section is the set of locks assigned to memory locations it accesses. However, this approach may use more locks than necessary, and introduce more overhead of lock acquisition and release. We say the number of locks required in this simple solution, i.e., the total number of memory locations accessed in a program, denoted as $|M|$, is the *upper bound* (UB) of the optimal MLA solution.

3.2 MLA Heuristic

Our MLA heuristic consists of three main steps (see Figure 4):

Step 1: Assign locks to non-interfering subgraph G_n using graph coloring heuristic (Line 6).

Step 2: Ensure that the serializing interfering edges in SNIG are correctly handled (Line 7).

Step 3: Finally propagate the locks to the interfering subgraph G_i (Line 8).

The first step is straightforward. We use a heuristic graph coloring algorithm [4] to color G_n, and one possible solution for our example is shown in Figure 2(b).

Next, we must ensure that critical sections connected by serializing interfering edges in SNIGs are correctly serialized. The details of this step are given by the function `HandleSerializingEdges` in Figure 4. In Figure 2(d), CS_1, CS_3 and CS_4 are in G_n and each of them has obtained a lock from the graph coloring. Interfering critical sections CS_1 and CS_3 are automatically serialized by sharing lock 1, but CS_3 and CS_4 are not. A straightforward method to solve this is let one of them "borrow" the lock from the other. For a serializing interfering edge (u, v), we say vertex u borrows the lock from v, denoted as $borrow(u \leftarrow v)$, if u adds v's lock to its lock set, $Lock(u) = Lock(u) \cup Lock(v)$. Denote the set of locks from u's non-interfering neighbors as $NIN(u)$, $NIN(u) = \bigcup_{(u,w)\in G_n} Lock(w)$. Before the borrowing, u has a disjoint set of locks with all its non-interfering neighbors, i.e., $Lock(u) \cap NIN(u) = \emptyset$. This implies that the concurrency between u and its non-interfering neighbors is maximized. After the borrowing, we also require u not share any lock with its non-interfering neighbors. This is satisfied if $Lock(v) \cap$

```
LockAssignment(G)
1.   Initialize Lock(u) for all u ∈ V as empty
2.   Partition the graph G
3.   if G_n = φ
4.      assign a global lock to each critical section
5.   else
6.      HLB = GraphColoring(G_n)
7.      HandleSerializingEdges(G_n^s)
8.      LockPropagation(E_ci, G_i)
9.   end if
10.  if HLB > |M| then
11.     for each v ∈ V
12.        Lock(v) = ⋃_{i∈LS(v)} Lock(i)
13.     end for
14.  end if

HandleSerializingEdges (G_n^s)
15.  for each serializing interfering edges (u, v)
16.     if Lock(u) ∩ Lock(v) = ∅
17.        if borrow(u ← v) is safe
18.           Lock(u) = Lock(u) ∪ Lock(v)
19.        else if borrow(v ← u) is safe
20.           Lock(v) = Lock(v) ∪ Lock(u)
21.        else
22.           HLB = HLB + 1
23.           add a new lock to u and v's lock sets
24.        end if
25.     end if
26.  end for

LockPropagation(E_ci, G_i)
27.  for each (v_n, v_i) ∈ E_ci
28.     sequence = BreadthFirstSearch(G_i, v_i)
29.     Arbitrarily pick one lock l from v_n's lock set
30.     for each v in sequence
31.        Lock(v) = Lock(v) ∪ {l}
32.     end for
33.  end for
```

Fig. 4. Lock Assignment Heuristic

$NIN(u) = \emptyset$, that is, none of u's non-interfering neighbors has u's borrowed lock from v. In this case we say the borrowing is "safe", which means it does not reduce concurrency among non-interfering critical sections.

In our example in Figure 2, in order to enforce the mutual exclusion between CS_3 and CS_4, we first let CS_4 borrow the lock from CS_3, then $Lock(CS_4) = \{1, 3\}$. This is shown in Figure 2(f). However, this borrowing is not safe, because one of CS_4's non-interfering neighbor CS_1 would share lock 1 with it. Then we try the alternative way. We let CS_3 borrow the lock from CS_4. This is illustrated

in Figure 2(g). This borrowing is safe because $Lock(CS_4) \cap NIN(CS_3) = \emptyset$, where $NIN(CS_3) = \{2\}$. Note that if neither borrowing is safe, we will introduce a new lock and add it to both end vertices' lock sets. The procedure of lock borrowing is summarized in Figure 4.

The first two steps together color the SNIG G_n^s. Finally, in function `LockPropagation`, we propagate the SNIG lock assignment result to the interfering subgraphs G_i . The interfering subgraph G_i is connected to the non-interfering subgraph G_n through a set of crossing edges (v_n, v_i), where $v_n \in G_n$, and $v_i \in G_i$. Each (v_n, v_i) is an interfering edge, that means v_i should share at least one of v_n's lock obtained from the graph coloring. We say v_n "propagate" a lock to v_i. If v_i has more than one incident crossing edges, then it should inherit locks from all its neighbors in G_n. Subsequently, v_i propagates its lock set to its neighbors in G_i. This propagation continues until every vertex in G_i inherits locks from its neighbors. This procedure can be simply implemented as a set of breath-first searches, with each v_i at a crossing edge as the source vertex. The algorithm is shown in Figure 4. One propagation result of our example is shown in Figure 2(h). An important property of this lock propagation is that it does not introduce any new lock, therefore the number of locks required to color G_i cannot exceed the number of locks required to color the SNIG G_n^s.

The final lock assignment result is shown in Figure 2(h). We refer to the number of locks required to color G as the Heuristic Lock Bound (HLB). We have mentioned in the naive solution that the upper bound UB of the required locks is the number of shared memory locations accessed in the concurrency graph G. In some cases HLB might exceed UB, and we need to choose the smaller one from HLB and UB for lock assignment. The MLA heuristic algorithm is summarized in Figure 4.

The following theorems show that our MLA heuristic can preserve the mutual exclusion between critical sections without any loss of concurrency. They also show that lock assignment on an arbitrary concurrency graph G is optimal if the lock assignment on SNIG of G is optimal. Detailed proofs can be found in [5].

Theorem 1. *When the algorithm LockAssignment (G) terminates, any pair of interfering critical sections in G share at least one common lock.*

Theorem 2. *When the algorithm LockAssignment (G) terminates, any pair of non-interfering critical sections do not share any lock.*

Theorem 3. *Lock assignment on a concurrency graph G is optimal if and only if the lock assignment on its SNIG G_n^s is optimal.*

The concurrency graph partitioning runs in $\mathcal{O}(V + E)$ time, and the graph coloring runs in $\mathcal{O}(V^2)$ time. At the worst case, the time complexity of `HandleSerializingEdges` and `LockPropagation` are $\mathcal{O}(E * V)$ and $\mathcal{O}(E^2 + V * E)$, respectively. Therefore, at the worst case the total time complexity of `LockAssignment` is $\mathcal{O}(E^2 + V * E)$.

3.3 ILP Formulation

In this section, we formulate the MLA problem as an ILP problem. Given a concurrency graph $G = (V, E)$, we introduce 0-1 variables $f_{u,i}$ to indicate whether lock i is assigned to node u in G, $1 \leq u \leq |V|$, and $1 \leq i \leq |M|$, where M is the set of shared memory locations that are accessed in all critical sections. Recall that the number of locks given by an optimal solution cannot exceed $|M|$. Since each critical section must be assigned at least one lock, we have the following constraint:

$$f_{u,1} + f_{u,2} + \cdots + f_{u,|M|} \geq 1 \quad \text{for all } u \in G \tag{1}$$

We use 0-1 variables l_i to indicate whether lock i is assigned to any critical section, $l_i = f_{1,i} \vee f_{2,i} \vee \cdots \vee f_{|V|,i}$. This condition is represented by the following constraints:

$$f_{1,i} + \cdots + f_{|V|,i} \geq l_i \tag{2}$$

$$f_{1,i} + \cdots + f_{|V|,i} \leq |V| \times l_i \tag{3}$$

Next we derive conditions that ensure the lock assignment is correct and maximizes the parallelism. Recall that a lock assignment solution is correct if interfering critical sections u and v share some lock, and parallelism is maximized if non-interfering critical sections are assigned two disjoint sets of locks. Let 0-1 variable $s_{u,v,i}$ indicate whether u and v share lock i, then $s_{u,v,i} = f_{u,i} \wedge f_{v,i}$. This condition is imposed by the following constraints:

$$f_{u,i} + f_{v,i} \geq 2 \times s_{u,v,i} \tag{4}$$

$$f_{u,i} + f_{v,i} \leq 2 \times s_{u,v,i} + 1 \tag{5}$$

We use 0-1 variable $s_{u,v}$ to indicate whether u and v share any lock. Then $s_{u,v} = s_{u,v,1} \vee \cdots \vee s_{u,v,|M|}$. The following two constraints represent this condition:

$$s_{u,v,1} + \cdots + s_{u,v,|M|} \geq s_{u,v} \tag{6}$$

$$s_{u,v,1} + \cdots + s_{u,v,|M|} \leq |M| \times s_{u,v} \tag{7}$$

Then

$$s_{u,v} = 1 \qquad \text{for interfering edge } (u, v) \tag{8}$$

$$s_{u,v} = 0 \quad \text{for non-interfering edge } (u, v) \tag{9}$$

The total number of locks used is:

$$N = l_1 + \cdots + l_{|M|} \tag{10}$$

Therefore, the MLA problem is to minimize N subject to inequalities (1) to (9).

4 Experimental Results

In this section, we present two sets of experiments to evaluate our lock assignment algorithm. In the first set of experiments, we compare the results produced by our MLA heuristic with the optimal solutions based on the ILP formulation on a set of 300 random concurrency graphs. In the second set of experiment we evaluate the effectiveness of the MLA heuristic using Splash2 [3] benchmarks.

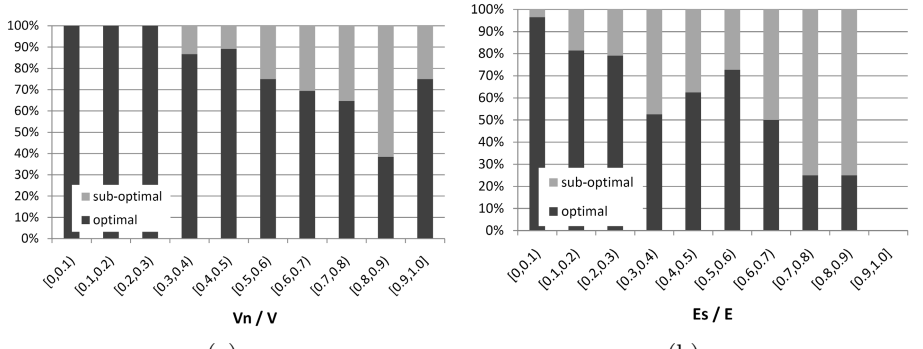
4.1 Precision Evaluation

To study the precision of our MLA heuristic we implemented our ILP formulation in the commercial ILP solver CPLEX, and tested the heuristic and the ILP formulation on a set of 300 randomly generated concurrency graphs with characteristics shown in Table 1. We limited our random concurrency graphs to contain at most 16 nodes due to time constraints in the ILP solver. It shows that our heuristic solution is optimal for 83.3% of tested graphs. For the remaining 16.7% of graphs our heuristic assigns more locks than the optimal solutions, and in the worst case two more locks than optimal solutions are assigned.

We also evaluated the influence of non-interfering subgraph G_n and serializing interfering edges E_s for lock assignment. For this purpose, we showed the precision of the MLA heuristic with the increase of the relative size of non-interfering subgraph, given by V_n/V, and with the increase of the relative number of serializing interfering edges, given by E_s/E, in Figure 5(a) and (b), respectively. As an example, Figure 5(a) shows that our MLA heuristic gives optimal solutions to about 70% of test cases that have $V_n/V = [0.6, 0.7)$ and sub-optimal solutions (i.e., assign extra locks) for the remaining 30%. Figure 5(a) and (b) illustrate that the precision of our heuristic depends on the non-interfering subgraph size and the relative number of serializing interfering edges.

Table 1. Features of random concurrency graphs

	Avg	Min	Max
Vertices (V)	8.63	2	16
Edges(E)	16.73	1	53
Edge Density E/V^2	0.19	0.09	0.28
Non-interfering edges (E_n)	3.37	0	20
E_n/E	0.22	0	1
V_n/V	0.43	0	1
Serializing interfering edges	2.85	0	27
E_s/E	0.10	0	0.53



(a) (b)

Fig. 5. Precision of the MLA heuristic

Table 2. Benchmarks and lock assignment results

Application	Barnes	Cholesky	Ocean-cont	Radiosity	Water-nsq
Description	N-body	Matrix factoring	Hydro-dynamics	3-D rendering	Water molecules
Problem size	262144 bodies	tk29.O B8 C256	514 × 514	largeroom batch	512 molecules
CSs	6	7	4	37	9
CS time (1 proc)	6.29%	32.37%	0.11%	9.93%	11.54%
Lines of code in CS (avg/max)	17.17 / 68	10.86 / 37	1.75 / 3	12.79 / 85	2.89 / 6
Funcs in CS	1	1	0	10	1
Locks assigned	3	4	4	8	7
Locks for each CS (max)	1	1	1	4	1

4.2 Performance Study on Sun-Fire

Next we study the performance of the MLA heuristic using a set of Splash2 [3] benchmarks listed in Table 2. Splash2 benchmarks call the Pthreads library [2], and mutual exclusion is enforced by `pthread_mutex_lock(<lock_var>)` and `pthread_mutex_unlock(<lock_var>)` functions with explicit lock variables. For the purpose of our performance study, we manually transformed each lock/unlock region into a critical section. We constructed the concurrency graph for each benchmark manually, and applied the MLA algorithm to calculate the lock assignment. The number of locks assigned to each benchmark is shown in Table 2.

We then ran the set of benchmarks on Sunfire 10-processor 750MHz machine, and collected two sets of data for each benchmark to evaluate our heuristics: (1) T_s: the execution time of the benchmark when all critical sections are controlled by a single lock, and (2) T_{MLA}: the execution time of the benchmark with lock assignment using our MLA heuristic. Figure 6 shows the performance improvement of our lock assignment with respect to the single global lock, i.e., $(T_s - T_{MLA})/T_s$, running on different number of threads. Cholesky and Radiosity have shown a performance improvement of 30.17% and 14.76%, respectively, due to the decrease of lock contention and serialization. On the other hand, Barnes, Ocean-cont and Water-nsq show a much lower performance improvement for two main reasons. First, the amount of time spent on critical sections is a small portion of the total execution time. For instance, as shown in Table 2, for Ocean-cont, the time spent on critical sections takes only 0.11% of the total execution time. Second, in Barnes and Water-nsq data is often organized as arrays or complicated user-defined data structures, and is accessed in a dynamic pattern that cannot be predicted during the compilation time. When we constructed the

[2] The original Splash2 benchmarks utilize the Argonne National Laboratories (ANL) parmacs macros for parallel constructs. We have re-configured them to call the Pthreads library.

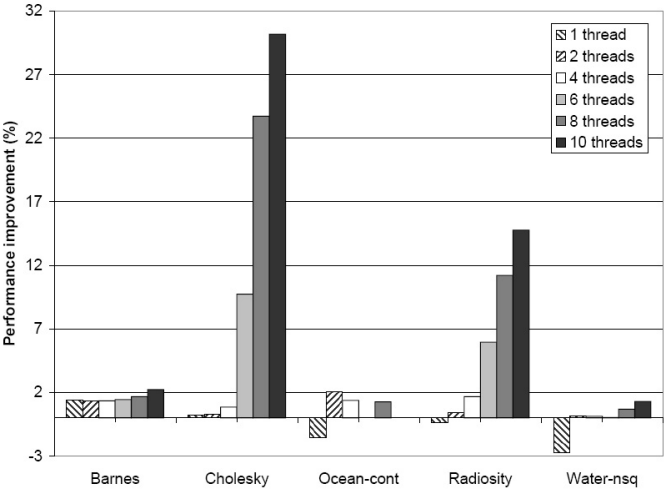


Fig. 6. Performance improvement with respect to single lock

concurrency graphs we conservatively treated such arrays and user-defined data structures as scalar units. This conservative approach may introduce "spurious" interference among critical sections, which results in unnecessary serialization. The unnecessary serialization will then increase lock contention among threads during the execution time. Some more sophisticated analysis techniques such as shape analysis [6], or dynamic conflict resolving techniques such as transactional memory and synchronization state buffer (SSB) [7] are needed to exploit further concurrency among critical sections in these benchmarks.

5 Related Work

Recently there has been some work on compiler based lock inference technique. Emmi et al. [8] propose a lock allocation problem that takes a multithreaded program annotated with `atomic` sections and infers a lock assignment to `atomic` sections to preserve its atomicity and deadlock freedom. They formulate the lock allocation problem as an ILP problem which minimizes the conflict cost between atomic sections and minimizes the number of locks. No heuristic solution is presented in their work. Our lock assignment differs from lock allocation in the following two aspects. First, our lock assignment problem maximizes the parallelism among critical sections using the minimum number of locks, while the lock allocation problem uses the minimum number of locks to minimize the conflict cost, a metric that is not clearly related with the parallelism. Second, we present both the heuristic solution and the ILP formulation for lock assignment problem. We use the ILP formulation to evaluate the optimality of the lock assignment heuristic. We also use scientific applications to evaluate the lock assignment heuristic and present performance improvement.

Hicks et al. [9] has proposed a lock inference techniques for atomic sections, which first determines a set of shared memory locations in the program, then uses a "mutex inference" algorithm to infer a set of locks for each atomic section to preserve its atomicity. The basic idea of their mutex inference algorithm is to find the dependence relation among shared memory locations, and partition the shared memory locations into sets according to this dependence relation. Locks are then assigned to each memory location set. Since the mutex inference algorithm is not optimization based, it may infer more locks than our lock assignment algorithm.

Autolocker [10] takes the programs annotated with pessimistic atomic sections and a programmer controlled lock assignment, and infers a compiler controlled lock assignment that is free of deadlocks and data races.

Vaziri et al. [11] proposed a data-centric synchronization approach for writing concurrent programs using atomic sets, which are a set of shared memory locations that have "similar" data consistency properties. Accesses to fields in an atomic set are assumed to take place atomically in "units of work". Taken a program with annotated atomic sets, the compiler infers units of work automatically and translates them into synchronized blocks. Our work complements Vaziri et al.'s work in that we can analyze and determine the atomic sets and units of work using concurrency analysis and lock assignment algorithm.

Some other optimization techniques on locks have been reported. Diniz and Rinard [12] present data lock coarsening and computation lock coarsening techniques to reduce the overhead of fine-grain locks in Java programs. Choi et al. [13] and Aldrich et al. [14] remove unnecessary synchronization from Java programs.

6 Conclusions

In this paper we proposed a lock assignment technique to simplify the mutual exclusion in multithreaded programs. It takes the programs annotated with `critical` sections and finds the minimum number of locks needed to enforce mutual exclusion among interfering critical sections without any loss of concurrency. Experimental results are very encouraging and show that our method can be used to improve the performance of multithreaded programs with mutual exclusion by exploiting concurrency among multiple critical sections. An extension of this work to support read/write locks is a subject for future work.

References

1. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann, San Francisco (1993)
2. 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)
3. The Stanford FLASH Prjoect. Stanford parallel applications for shared memory (SPLASH) benchmark, `http://www-flash.stanford.edu/apps/SPLASH/`

4. Briggs, P.: Register Allocation via Graph Coloring. Ph.D thesis, Rice University (1992)
5. Zhang, Y., Sreedhar, V., Zhu, W., Sarkar, V., Gao, G.: Optimized lock assignment and allocation: A method for exploiting concurrency among critical sections. Technical Report CAPSL-TM-065-revised, University of Delaware, Newark, DE (2007)
6. 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: POPL 1996: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 1–15 (1996)
7. Zhu, W., Sreedhar, V.C., Hu, Z., Gao, G.R.: Synchronization state buffer: supporting efficient fine-grain synchronization on many-core architectures. In: ISCA 2007: Proceedings of the 34th annual international symposium on Computer architecture, pp. 35–45 (2007)
8. Emmi, M., Fischer, J.S., Jhala, R., Majumdar, R.: Lock allocation. In: POPL 2007: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 291–296 (2007)
9. Hicks, M., Foster, J., Pratikakis, P.: Lock inference for atomic sections. In: TRANSACT 2006: Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (2006)
10. McCloskey, B., Zhou, F., Gay, D., Brewer, E.: Autolocker: synchronization inference for atomic sections. In: POPL 2006: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 346–358 (2006)
11. Vaziri, M., Tip, F., Dolby, J.: Associating synchronization constraints with data in an object-oriented language. In: POPL 2006, pp. 334–345. ACM, New York (2006)
12. Diniz, P., Rinard, M.: Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. In: Sehr, D., Banerjee, U., Gelernter, D., Nicolau, A., Padua, D.A. (eds.) LCPC 1996. LNCS, vol. 1239, pp. 284–299. Springer, Heidelberg (1997)
13. 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)
14. Aldrich, J., Sirer, E., Chambers, C., Eggers, S.: Comprehensive synchronization elimination for java. Science of Computer Programming 47(2-3), 91–120 (2003)

Set-Congruence Dynamic Analysis for Thread-Level Speculation (TLS)

Cosmin E. Oancea and Alan Mycroft

Computer Laboratory, Cambridge University, Cambridge, CB3 0FD, UK
Cosmin.Oancea@cl.cam.ac.uk, Alan.Mycroft@cl.cam.ac.uk

Abstract. The move to multi-core has increased interest in parallelizing sequential programs. Classical dependency-based techniques, although successful for some classes of programs, often fail due to the one-sided (conservative) approximation of program behavior. Thread-level speculation enables increased parallelism by allowing out-of-order execution: correct dependences are ensured by run-time monitoring and possible rollbacks. Two-sided approximations of program behavior are now acceptable if the rollback ratio is kept small. We describe dynamic analyses, based on representing dependencies as modular congruences, which can be employed on-line or off-line. One, *thread partitioning*, efficiently enables loop iterations to be allocated to threads (and calculates the maximum effective concurrency); the other, *fine-grain memory partitioning*, calculates a hash function that reduces space overhead and performance loss due to TLS-metadata-based and cache-based task interference.

1 Introduction

Thread-level speculation (TLS) is a parallelization technique that allows the compiler to partition the program into concurrent threads even in the presence of dependencies. While important work examines hardware TLS [2, 10, 21, 22], this paper examines higher-level analyses than the one naturally done by hardware.

Current software-TLS approaches [5, 6, 18] exhibit heavy-transactional support, and can yield good speed-up when (i) the static compiler disambiguates[1] enough accesses to amortize the speculation overhead, and (ii) the iteration granularity is high enough to amortize the transactional overhead related to starting a new iteration. Their design assumes little about patterns of accesses to memory, merely that these accesses generate relatively few dependencies.

Lightweight models [16, 17] facilitate a compositional perspective to software-TLS: a coarse variable-based memory partitioning is performed first and separate optimized (adaptive) TLS models are employed on each partition to exploit regular access-patterns. In the latter's presence these models can be very effective even when most of the instructions require speculative support. These lightweight models perform best when memory accesses with a partition are regular (think linear strided), but cannot be proved so statically because (i) of perhaps few singular points and (ii) of static analysis hindrances such as: complex control-flow

[1] I.e. it is provable that no transactional support is needed.

J.N. Amaral (Ed.): LCPC 2008, LNCS 5335, pp. 156–171, 2008.

and data-structures (increased abstraction level), potential aliasing. Evidence of these hindrances is illustrated by proposals for C language extensions that provide a special scope that guarantees the absence of cross-iteration dependencies [13]. Lightweight models require a *hash function*[2] which aims to reduce speculative storage without generating inter-thread conflicts.

This paper proposes a framework for dynamic analysis to guide the introduction of lightweight TLS models. The main idea is to use profile runs to build patterns capturing dependent iterations. This leads to two orthogonal techniques: First, the iteration space is partitioned based on dependent statement-instance pairs with the goal of executing dependent iterations on the same thread. Second, the data space is partitioned into (nearly) disjoint access patterns from threads, to produce efficient TLS models' *hash-functions*. While TLS-related optimizations [2, 11, 23] have focused so far on tuning the original code to enhance speed-up, we investigate the equally important direction of fine-tuning the TLS model according to code's access patterns (see Section 4.3 for speed-up results).

Previous profiling solutions for TLS were mainly aimed at (i) identifying suitable code for TLS (few dependencies) and designing flexible thread-formation schemes that delay thread-spawning to minimize violations [1, 12], (ii) inferring linear predictors [2, 19] for scalars which are very likely to violate dependencies, and (iii) developing TLS cost-models to predict speed-up [7]. This paper's main contribution is to introduce (at a high-level) an address-based, set-congruence model and algebra that, to our knowledge, is the first that attempts to:

- compute a iteration-to-thread partitioning that respects frequent dependencies[3] and addresses the iteration granularity need;
- identify *coarse-grained memory partitions*, i.e. an exhaustive set of address ranges; access patterns for these may vary, thus assigning TLS models per partition is most effective in general;
- identify regular, *fine-grained* access-patterns and use them to construct the hash functions that allow lightweight TLS models to be effective (*small memory overhead*). The latter is sketched in Section 4 due to space constraints.

In principle, we are interested in both conservative – all events are modeled, and two-sided approximation – enough events are modeled that the cost of speculation failure is kept within cost bounds. With the latter, our analysis is light enough to be applied both off-line and on-line (just-in-time), as desired: the analysis is run on the profiled information corresponding to a *small* iteration window W, and the algorithms are $O(n \log n)$ in the number of profiled addresses. (Regularity can also be verified on a conveniently far away window.)

Comparing with static approaches [14, 15], besides the obvious more conservative (and hence imprecise) trait these exhibit, we note two interesting differences: First, static approaches need to investigate how various loop-index variables are combined to form an array index, hence requiring complex, relational analysis. Since dynamic analysis looks directly at the accessed address, our model is simpler (non-relational flavor) while covering the exploitable cases. Second,

[2] We abuse notation here: it describes regular accesses instead of randomizing.

[3] Executes dependent iterations on the same thread – enabled by *in-place* TLS models.

our dynamic analysis may be applied to richer containers (linked lists, trees) than (mere) arrays as long as the memory has a regular structure; Chilimbi and Larus's work [3, 4] improves cache behavior by re-organizing memory to a similar regular structure that also facilitates our dynamic analysis.

The rest of the paper is structured as follows: Section 2 provides the background, motivation, states the general problem and compares with several classical approaches. Section 3 presents how the profiling information is gathered and introduces (formally and in detail) the thread partitioning analysis. Due to space constraints, Section 4 only briefly sketches the memory-partitioning analysis and presents speed-up results. Section 5 concludes the paper.

2 Background, General Problem, Related Work

This paper uses modular arithmetic significantly. We write $\mathbb{Z}_n$ to mean the integers (mod n) ($\mathbb{Z}_n \equiv \{\{0, n, 2n, \ldots\}, \{1, n+1, 2n+1, \ldots\}, \ldots, \{n-1, 2n-1, \ldots\}\}$). Elements of $\mathbb{Z}_n$, are referred to as *cosets*. This section briefly introduces software-TLS, provides the motivation and states the general problem for our two dynamic analysis techniques, and compares them with related static approaches.

2.1 Software TLS

We give here the essential TLS information required to understand this paper; [2, 6, 18, 24] provide a more comprehensive perspective. TLS exploits code regions that expose good amount of parallelism but for which static analysis fails to guarantee safety. Under TLS threads execute out of order, and use software/hardware structures, referred as *speculative storage*, to record the necessary information to track the inter-thread dependencies and to revert to a *safe* point and restart the computation upon the occurrence of a *dependency violation* (*rollback recovery*).

The thread executing the lowest numbered iteration of all is referred to as the *master* thread since it encapsulates both the correct sequential state and control-flow; the others are *speculative* threads since they may consume "dirty" values and cause rollbacks. *Serial-commit* TLS models [5, 6, 17] isolate the speculative from the global state: each thread buffers its write-accesses, and commits them when it becomes master, hence WAR and WAW dependencies are implicitly satisfied. *In-place* models [8, 16, 20] modify directly the program state, while still enforcing the sequential semantics. Important differences with respect to *serial commit* models are that (i) all types of dependencies (RAW, WAR and WAW) may generate violations, but (ii) they are scalable – in number of processors that may contribute to speed-up, and (iii) allow a more flexible iteration-to-thread partitioning (threads may execute non-consecutive groups of iterations, see later).

Finally, empirical results suggest that a software-TLS application requires an iteration's granularity to be in the range of thousands of instructions: (i) big enough to amortize the speculative overhead corresponding to starting a new iteration, (ii) but not too big – so that the speculative storage is kept within reasonable bounds. As discussed in the next section, when the original loop does not provide enough granularity, we refactor the loop in a fashion that preserves

iterations' execution locality. In this sense, we denote by W_{min} and W_{max} the minimal, maximal bounds for the number of consecutive, original iterations that are allowed to execute concurrently. A collateral, but important advantage of increasing iteration granularity is that it improves load-balance among threads.

2.2 Thread Partitioning – High Level View

Given a block B forming the body of the loop `for(int i=0; i<N; i++)` B`(i);` we would like to schedule the iterations $B(i)$ for a multi-core processor. We denote by P the number of processors and by C the number of threads used to parallelize the program. (In general, maximal speed-up occurs when $C \geq P$).

We assume we have profiled a window of W_{max} consecutive iterations. The general problem addressed in Section 3 is to find a repetitive structure (π) that defines how iterations are assigned to threads so likely dependencies are satisfied. $\pi : \{0, \ldots, W-1\} \rightarrow \{0, \ldots, C-1\}$ gives the mapping from W consecutive iterations to concurrent threads. Writing as usual $\pi^{-1}(c) = \{i \mid \pi(i) = c\}$, the iterations executed by thread j are simply $\pi^{-1}(j)$. Note that C and W are also analysis outputs; convenient values should maximize application's available degree of parallelism, keep threads well (load) balanced, and provide TLS's desired granularity ($W_{min} \leq W \leq W_{max}$). With π, W and C the loop is re-written as:

```
parfor(t=0; t<C; t++)
   for(k=0; k<N/W; k++) {
      for_each(j ∈ π⁻¹(t)) B(k*W+j);
      cond_wait; /* required by TLS */  }
```

The application of TLS requires loose synchronization between threads to keep concurrency well-localized This is depicted via `cond_wait` which preserves the invariant that always, at most `C` consecutive "expanded" iterations execute concurrently ($|\mathtt{k}_i - \mathtt{k}_j| < C,\ 0 \leq i, j < C$, where $\mathtt{k}_i$ is `k`'s value on thread i).

We give two examples to illustrate forms of B, in which we assume $P = 8$. *The first example* takes B(`i`) to be `a[i+4] = a[i] + 2`, code that features cross-iteration dependencies of distance 4. Without considering the iteration-granularity factor, a possible result is $C = 4$, $W = 4$ and $\pi^{-1}(j) = \{j\}$, meaning that iterations $j + 4\mathbb{Z} \equiv \{j, j+4, j+8, ...\}$ execute on thread j. Note that, with this code, hardware-parallelism is only partially exploited: we use 4 threads on 8 processors. To increase the iteration granularity, we may choose W a convenient multiple of 4, say $W = 16$ and have $\pi^{-1}(j) = \{j, j+4, j+8, j+12\}$. (The first refactored iteration for thread 0 consists of the original iterations $\{0, 4, 8, 12\}$.) Note however that this way of increasing iteration-granularity is only applicable to *in-place* TLS models, since the write-back phase of a serial-commit model cannot be implemented in any effective way ($W = 4$ for the serial-commit model).

The second example takes B(`i`) to be `a[i] = a[i] + 2`, and hence corresponds to (cross-iteration) dependency-free code. A possible result is $C = 8, W = 8$ and $\pi^{-1}(j) = \{j\}$ (iterations $j + 8\mathbb{Z}$ execute on thread j). Increasing iteration granularity by a factor of 4, yields $W = 32$ and $\pi^{-1}(j) = \{4j, 4j+1, 4j+2, 4j+3\}$, meaning that thread 0 executes iterations $\{0, 1, 2, 3, 32, 33, 34, 35, ...\}$ and so on.

(The first refactored iteration for thread 0 consists of the original iterations $\{0, 1, 2, 3\}$.) This method of increasing iteration granularity is applicable to both *serial commit* and *in-place* TLS models. (The serial-commit phase operates as expected since the new iteration is formed from consecutive original iterations.)

2.3 Exploiting Access-Patterns Via Adaptive TLS Models

Oancea and Mycroft [17] argue that rather than applying one over-arching TLS model to parallelize an application, software flexibility is, in some cases, better exploited by combining several lightweight TLS models [16, 17], each protecting disjoint areas of memory. In principle, lightweight TLS models attempt to exploit a program's access patterns and, where these exist, yield a very small memory overhead and hence good performance.

For illustration, we intuitively present a simple TLS technique to track RAW dependencies. Assume `LdVct[]` is a vector with as many entries as the size of an array `arr[]` that requires speculative support. A read from `arr[i]` in iteration `r` sets `LdVct[i]=r` *iff* `r` is currently the maximal iteration that has read `arr[i]`. A write to `arr[i]` by iteration `w` *discovers* a RAW *violation* when `w<LdVct[i]` since iteration `LdVct[i]` should have read the value written by `w`, but it did not.

To decrease speculative storage (`LdVct`) size, a (not one-to-one) hash function, of form $\texttt{hash}_{s,q,Q}(x) = ((x - s)\ \texttt{quo}\ q)\ \ \texttt{rem}\ Q$ can be used to map memory locations into indexes in `LdVct`. Now, the data space is partitioned into equivalence-classes ($x_1 \sim x_2 \Leftrightarrow \texttt{hash}(x_1) = \texttt{hash}(x_2)$), and a speculative read/write operation is interpreted as if any locations belonging to the same equivalence class may have been read/written. Although the execution soundness is guaranteed for any such (not one-to-one) `hash`, good speed-up is achieved only when the number of false-positives ($\texttt{hash}(x_1) = \texttt{hash}(x_2)\ \&\ x_1 \neq x_2$) leading to dependence violations is small, so that the additional rollback-recovery cost is vastly overcome by the small speculative memory-footprint and improved cache behavior. (Naively chosen `hash`es will likely translate to poor performance.)

Section 4 presents at a very high-level the analysis that determines `hash`'s $s, q,$ `and` Q parameters. Assuming a 32-bit word, the *first example* in Section 2.2, with $B(\texttt{i}) \equiv \texttt{a[i+4]} = \texttt{a[i]} + 2$, $C = 4$, $W = 16$ and $\pi^{-1}(j) = \{j, j+4, j+8, j+12\}$, $j \in \{0, .., 3\}$, yields $\texttt{hash}(x) = ((x - s)\ \texttt{quo}\ 4)\ \texttt{rem}\ 4$, where $s = a\ \texttt{quo}\ 4$, and a stands for the start address of array `a`. One can verify that $\texttt{hash}(x) \equiv i$, for all addresses x accessed by thread i. Similarly, the *second example*, with $B(\texttt{i}) \equiv \texttt{a[i]} = \texttt{a[i]} + 2$, $C = 8$, $W = 32$ and $\pi^{-1}(j) = \{4j, 4j+1, 4j+2, 4j+3\}$, $j \in \{0, .., 7\}$, yields $\texttt{hash}(x) = ((x - s)\ \texttt{quo}\ 16)\ \texttt{rem}\ 8$. One can verify that thread i accesses addresses that map to i via `hash`.

We can thus introduce speculation via a very small memory-overhead (the load/store vectors that track dependencies have sizes 4 and 8 for the two cases). Moreover, since a thread repetitively accesses the same index of `LdVct` (and different threads access different indexes) in the dependency tracking-structure we can obtain a cache-ideal layout of speculative storage.

2.4 Comparison with Static Analysis Techniques

The classical (static) treatment depends on the assumption that the loop-body B is simple in terms of (i) control flow – typically no conditionals, (ii) access-patterns – linear indexing, and (iii) used data-structures – basic type arrays, and (iv) provable no-aliasing. Where one of these does not hold, dependence analysis is likely to indicate sequential execution even on a multi-core processor. However, the dynamic behavior (in particular the data-dependencies) may in fact be reasonably regular, with a perhaps small number of exceptions; TLS allows the code parallelism to be extracted while providing the safety net with respect to these few exceptions. The dynamic analyses introduced in this paper optimizes TLS application: where strong regular behavior exists, lightweight, software-TLS models are effective even when most B's instructions require speculative support.

Our thread-partitioning analysis, presented in Sections 2.2 and 3, most closely resembles a form of octagonal analysis [15] but also using congruences [9]. Note also the difference that the traditional use of octagonal congruences is for analysing relationships between values of user variables while we analyse to determine values of the iteration number appearing at the ends of a run-time dependency ($x - y = c\ (modM)$ is a octogon-type congruence).

Our address-partitioning analysis, introduced in Section 2.3 and briefly presented in Section 4, is at a high level related to Masdupuy's analysis of trapezoid congruences [14]. The latter is a complex framework for relational integer analysis, aimed at describing multi-dimensional array indexes, that leads to "interval-like" or "congruence-like" information when interval or congruence analysis is relevant, respectively. We employ a similar strategy aimed at reducing `hash`'s image cardinality (i.e. Q), and thus TLS's memory overhead, but we restrict our intervals to be equal-sized, since we need a fast `hash`. While the introduction has recounted several profiling-related approaches, other TLS-related optimizations include data-flow algorithms for identifying "idempotent references" [11], aggressive instruction scheduling techniques aiming at reducing the stalls associated with scalar values [23], and other optimizations related to loop induction variables, light thread synchronization locks, and reduction operators [2].

3 Thread Partitioning

The analysis presented in this section (i) identifies the cross-loop dependencies that are likely to yield run-time violations, (ii) classifies dependencies into rare-events, which can be ignored, and (frequent) repetitive-events that need to be satisfied, and (iii) attempts to describe the iteration space via a *regular structure*, in which iterations involved in cross-loop dependencies are assigned to execute on the same thread, while maintaining the load-balance among threads. The latter is the most efficient method of satisfying frequent dependencies.

Constructing the regular structure is more useful than merely representing the value set of addresses at a given program point because of the need to model dependencies. It therefore involves representing relationships between addresses occurring at two program points in different iterations – one the source of the

dependency and one the target. Our analysis applies to both regular and irregular, and simple and nested loops. However, for simplicity, in the following we restrict our discussion to single loops and generalize in Section 3.6 for loop nests.

3.1 Notations, Preliminaries and Profiling Instrumentation

For simplicity, throughout the paper, we discuss our profiling and analysis techniques in the context of loop parallelization, where threads concurrently execute iterations out of the program order. However, this can be easily generalized to any thread-partitioning, as long as partitions are numbered in a fashion that respects the total order imposed by the sequential program's control flow.

We assume that a simple static-analysis is performed first, to identify the read/write accesses of memory locations that cannot be disambiguated and hence require speculative support. We refer to the latter as speculative program points (SPP). Note that a SPP is associated with either a write or a read access of memory locations; $\texttt{mode} : \texttt{SPP}_{dom} \rightarrow \{\texttt{r}, \texttt{w}\}$ represents this relation. We denote by $\texttt{A}_{dom}$, $\texttt{IS}_{dom}$ and $\texttt{SPP}_{dom}$ the domains of valid addresses, loop iteration space, and SPP. Hence $\texttt{A}_{dom}$, $\texttt{SPP}_{dom}$ and $\texttt{IS}_{dom} \subset \mathbb{Z}$, where we consider iteration i to be the i^{th} executed iteration in sequential program order.

At run-time, we employ an instrumentation phase that, for each SPP, gathers address-iteration pairs (PIA) recording which addresses were read/written by which iterations. Hence $\text{PIA}_{\texttt{q}} \subset \{\ (a, i) \mid a \in \texttt{A}_{dom}\ \texttt{and}\ i \in \texttt{IS}_{dom}\}$, $\texttt{q} \in \texttt{SPP}_{dom}$.

The cross-iteration dependencies that may appear at run-time can be identified by analyzing the PIAs corresponding to SPP pairs (PPP). For example if $(a, i_1) \in \texttt{PIA}_{\texttt{q}_1}$, $(a, i_2) \in \texttt{PIA}_{\texttt{q}_2}$, $i_1 < i_2$, $\texttt{mode}(\texttt{q}_1) = \texttt{w}$, and $\texttt{mode}(\texttt{q}_2) = \texttt{r}$, then we have a true-dependence (RAW) with the source being executed in iteration $\texttt{i}_1$ and the sink in iteration $\texttt{i}_2$. This leads to the following definitions:

Definition 1 (Dependency-Class Notation). *We denote by (it_{src}, it_{snk}, tp) the class of run-time, cross-iteration dependencies, such that the source/sink of the dependency (on some memory location) is executed by the iteration numbered it_{src} / it_{snk}, respectively, and $tp \in \{t, a, o\}$ denotes the dependency type: true (RAW), anti (WAR), or output (WAW). By construction we have: $it_{src} < it_{snk}$.*

Definition 2 (ADDG). *The dependency classes introduced in Definition 1 induce a directed acyclic dependency graph (ADDG), in which nodes are iteration numbers, and edges are directed from dependency's source to sink and are annotated with the type of the dependence (t, a, o). Singleton nodes are eliminated (they correspond to no dependency or to iteration-independent dependencies).*

After (just-in-time) profiling a number of iterations, we construct for each SPP pair ($(\texttt{q}_1, \texttt{q}_2) \in$ PPP) their associated directed acyclic dependency graph ($\text{ADDG}_{(\texttt{q}_1, \texttt{q}_2)}$), in which the singleton nodes are eliminated. Dependencies of distance greater than W_{max} are trimmed out since they cannot result in run-time violations (see `cond_wait` in Section 2.2). This approach of constructing per-PPP ADDGs as opposed to one whole-loop ADDG is motivated by the intuition that the resulting ADDGs often correspond to simple access patterns that, in many cases,

```
const int D = 4; const int B = 128;
1 for(int i=D; i<N; i++) {
2   a[i]    = ....  ;          // PP1
3   ....    = a[i-D];          // PP2
4   if (i%8 = 1)
5     ....  = a[i-1];          // PP3
6   a[i%B]  = ....  ;          // PP4
7   ....    = a[i%D];          // PP5
8   if(highlyUnlikelyCond())
9     ....  = a[i-1];          // PP6
10  e[i]    =                  // PP7
11            e[N-i];          // PP8
}
```

(a) Motivating Example

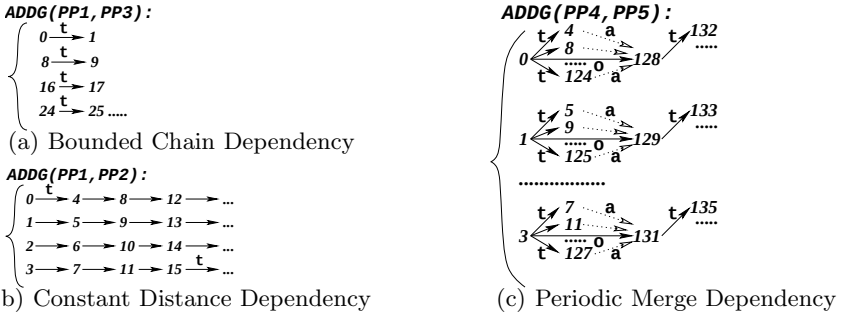
```
if(cond1 || cond2) {
        ... // PP1
} else {
        ... // PP2
}

BECOMES

if(cond1) {
        ... // PP1
} else if(cond2) {
        ... // PP1'
} else ... // PP2
```

(b) Branch Normalization

Fig. 1.



(a) Bounded Chain Dependency

(b) Constant Distance Dependency

(c) Periodic Merge Dependency

Fig. 2. ADDGs for Some Program Point (PP) Pairs in Figure 1

can be easily inferred and expressed through basic congruence formulas. To this end, we assume that *or conditionals* have been normalized via code cloning, as shown in Figure 1(b). The next section exemplifies our approach.

3.2 Example

The cross-iteration dependencies, for the code shown in Figure 1(a), fall in one of two categories: (i) dependencies that, if not synchronized, will result in frequent run-time violations, leading to poor performance, and (ii) dependencies that, at run-time, rarely violate the sequential program semantics.

With respect to *category (i)*, we identify three dependency-patterns that may still allow parallelism to be effectively exploited. Typical examples are the dependencies (of distances `D`, `1` and `1`) between `PP1-PP2`, `PP1-PP3` and `PP4-PP5`, whose corresponding ADDGs are shown in Figure 2. The ADDG corresponding to `PP4-PP4`, representing output-dependencies of distance 128 can be seen as the $\rightarrow^o$ arrows in in Figure 2(c), resulting in a pattern like 2(b).

Category (ii), includes for example the dependencies corresponding to PPP `PP1-PP6` and `PP7-PP8`. The former may cause a true-dependency of distance `1`, but its sink is guarded by a condition that is highly-unlikely to evaluate to

`true`. The latter PPP, if analyzed semantically, yields a group of cross-iteration, true/anti dependencies whose distances range uniformly from `N` to `0`. Since we assume $\texttt{N} \gg W_{max}$ they will cause very few run-time violations, roughly when the execution reaches the middle of the iteration space. (Note that a lightweight profiling approach may in fact not even discover these dependencies, which is consistent with our two-sided approximation strategy which ignores rare events.)

Ideally, we would like to fully exploit the available hardware parallelism, while introducing no explicit synchronization. For example, on a two-processor machine, an optimal thread-partitioning will execute iterations $0, 2, 4...$ on one thread, and iterations $1, 3, 5, ...$ on the other. This satisfies the observed dependencies without introducing any synchronization overhead, as the dependent instructions are executed on the same thread. On a four-processor machine it is probably better to use four threads, in which thread i executes iterations $i, i+4, i+8, ...$, where $0 \leq i < 4$. This satisfies the dependencies in Figures 2(c) and 2(b), while light synchronization is introduced to satisfy the dependencies between threads 0 and 1 (those in Figures 2(a)).

3.3 Set-Congruence Model (in $\mathbb{Z} \times \mathbb{Z}$)

As observed with the previous example, the ADDGs shown in Figure 2 have a repetitive structure that allows parallelism to be efficiently extracted (even under frequent dependencies). We aim at developing congruence relations such that:

- The repetitive structure is concisely and precisely described, and can be easily identified via pattern-matching type algorithms
- They can be effectively combined yielding a parallelization strategy that finds a good trade-off between the available code, the hardware parallelism and the introduced synchronization.

Definition 3 (Modulo/Step Operators). *Given $0 \leq a, b < M$, where $a, b, M \in \mathbb{N}$, we define the modulo operator $<M>$, intended to model Figure 2(b), and the* step *operator $|M>$, intended to model Figure 2(a) as:*

$$(a,b)<M> = \{(x,y) \mid x \equiv a \ (mod\ M) \ \texttt{and} \ y \equiv b \ (mod\ M)\} \quad (Figure\ 2(b))$$

$$(a,b)|M> = \begin{cases} \{(a+kM, b+kM) \quad\quad\quad\quad\quad\quad \mid k \in \mathbb{N}\}, \texttt{if}\ a < b, \\ \{(a+kM, b+(k+1)M) \mid k \in \mathbb{N}\}, \texttt{if}\ a \geq b \end{cases} \quad (Figure\ 2(a))$$

M is called characteristic. For $M = 0$, $(a,b)<0> = (a,b)|0> = \{(a,b)\}$. Finally, we lift the modulo/step operators (from pairs) to sets (of pairs): $S<M> = \cup_{(a,b)\in S}(a,b)<M>$, where $S \in \mathbb{P}(\mathbb{Z}_M \times \mathbb{Z}_M)$. The definition of $S|M>$ is similar.

Note that the step operator is more precise than the modulo operator: $S|M> \subseteq S<M>$. For example $(0,8) \in (0,0)<4>$ but $(0,8) \notin (0,0)|4>$. We represent the ADDG in Figure 2(a) as $(0,1)|8>$, since there is no constraint that requires iterations 0 and 9 to be executed on the same thread, for example.

Although it corresponds to an orthogonal pattern (see Section 3.5), we could represent the ADDG in Figure 2(c) via the modulo operator as $\cup_{0 \leq i < 4}(i,i)<4>$ (the

step operator fails to represent it since, for example, $(0,8) \notin (0,0)|4>$). Furthermore, the ADDG in Figure 2(b) also requires the modulo operator ($\cup_{0 \le i < 4}(i,i)|4>$) due to the implicit transitive closure: iterations 0 and 4, and 4 and 8 execute on the same thread, hence iterations 0 and 8 execute on the same thread (although iterations 0 and 8 are not dependent). (The transitive closure is a result of iterations 0, 4 and 8 belonging to the same ADDG's connected component.)

3.4 Set-Congruence Algebra (in $\mathbb{Z} \times \mathbb{Z}$)

We present now how formulas describing ADDG's basic patterns are combined. The non-relational, static analysis of integer congruence properties employs the lattice of integer cosets to join same-variable formulas: $(a_1 + b_1\mathbb{Z}) \sqcup (a_2 + b_2\mathbb{Z}) = a_1 + \texttt{gcd}(b_1, b_2)\mathbb{Z}$ if $\texttt{gcd}(b_1, b_2)|(a_2 - a_1)$ and $\mathbb{Z}$ otherwise. Applying our analysis on the distinguished subset of $\mathbb{Z} \times \mathbb{Z}$ in Definition 3 results in a (practical) formula to manipulate these descriptions.

Definition 4 (Additive Subgroup). *We denote by $[m]_M$ the additive subgroup of $\mathbb{Z}_M$ generated by m. Note that $[m]_M = [g]_M$, where $g = \texttt{gcd}(m, M)$, since both subgroups have the same cardinality M/g and g generates m.*

Assume $a < b$ and $m_1 < m_2$. The step relation yields the invariant: $(a,b)|m_1> = \{(a+k*m_1, b+k*m_1) \mid k \in \mathbb{Z}\} \subseteq \{(a+e+k*m_2, b+e+k*m_2) \mid k \in \mathbb{Z},\ e \in [m_1]_{m_2}\} = \cup_{e \in [m_1]_{m_2}}(a+e, b+e)|m_2>$.

The modulo relation is similar to the integer congruence unification. Denoting $m = \texttt{gcd}(m_1, m_2)$ ($[m_1]_{m_2} \equiv [m]_{m_2}$) leads to: $(a,b)<m_1> \subseteq (a,b)<m>$. (Also $(a,b)|m_1> \subseteq \cup_{e \in [m_1]_{m_2}}(a+e, b+e)|m_2> \subseteq (a,b)|m> \subseteq (a,b)<m>$.)

We demonstrate now the usefulness of differentiating between step and modulo relations. Combining two congruence relations corresponds to taking the union of the sets they represent. For the modulo relation we have: $\{(0,1)\}<8> \cup \{(0,1)\}<18> \subseteq \{(0,1)\}<2>$, which implies that the program cannot be parallelized (as iterations 0 and 1 modulo 2 are executed on the same thread). However, with the step relation we get: $\{(0,1)\}|8> \cup \{(0,1)\}|18> \subseteq S|18>$, where $S = \{(0,1), (8,9), (16,17), (6,7), (14,15), (4,5), (12,13), (2,3), (10,11)\}$. In this case we can run 9 concurrent threads while satisfying the observed dependencies: thread `i` executes iterations `S[i][0]` and `S[i][1] mod` 18, where $0 \le i < 9$. The next theorem formalizes these results.

Theorem 1 (Step/Modulo Refining). *Let $U \subseteq \mathbb{P}(\mathbb{Z} \times \mathbb{Z})$ be of the form $S|m>$, and $g = \texttt{gcd}(M, m)$. The smallest set $U', U \subseteq U'$, of the form $S'|M>$ exists and is obtained when: $S' = \{(x,y) \mid x \equiv a + e$ `and` $y \equiv b + \alpha * m + e$ `mod` M, `where` $(a,b) \in S, \alpha = 0$ `for` $a < b$ `and` 1 `otherwise`, `and` $e \in [g]_M\}$ For the modulo relation the set S' can also be computed, but $S'<M> \equiv S<g>$.*

Proof. Straightforward application of the finite additive subgroup theory.

The rest of this section gives the unification rules for the step/modulo operators: intuitively, they aim to obtain a formula of best precision (highest degree of parallelism) and conciseness (smallest characteristic), in this order.

Definition 5 (Degree of Parallelism). *Let l_{max} be the maximal number of nodes of a connected component of the* ADDG *induced by a formula of the form $S<m>$, where only iterations $0..(m-1)$ are considered. The* degree of parallelism *of $S<m>$ is $\lceil m/l_{max} \rceil$. The same holds for $S|m>$.*

Definition 6 (Step/Modulo Unification). *MS.1:* $S_1|m> \sqcup S_2<m> = (S_1 \cup S_2)<m>$ *MS.2:* $S_1|m_1> \sqcup S_2<m_2> = ([S_1|m_1>]_m \cup [S_2<m_2>]_m)<m>$*, where $m \in \{m_2, \texttt{gcd}(m_1, m_2)\}$ is the value that maximizes the degree of parallelism. In the case of equality, take the smaller m. Similarly, combining* two step relations *(|>) yields a step relation. Combining* two modulo relations *(<>) yields a modulo relation where the resulting characteristic is computed by taking the* `gcd`*.*

3.5 Pattern Identification

Figure 2 gives three dependency patterns for a given PPP that allow speculative parallelism to be extracted. They hence constitute the basic building-blocks of our analysis; if no such pattern can be found then speculation is likely to be unhelpful. While it might appear natural to merely union the ADDGs for each PPP, in general this gives an ADDG from which is hard to extract such patterns. Instead, we determine potential patterns for each ADDG and unify ($\sqcup$) their set-congruence formulas as described in the previous section.

For space reasons we do not give exact pattern-matching algorithms, instead we prefer to outline the process by (i) identifying the main pattern characteristics such an algorithm should identify, (ii) giving the pattern's congruence formula, and (iii) where not already discussed, present how formula unification is achieved.

Bounded-Chain Dependency Pattern: The first basic pattern corresponds to the one shown in Figure 2(a). An ADDG satisfies this pattern if (i) there are very many connected components, each containing a small number of nodes, l (typically 2), (ii) for any two dependencies in the same position on different connected components $p_i \rightarrow q_i$ and $p_j \rightarrow q_j$, $m_{i,j} = p_i - q_i = p_j - q_j$, and, (iii) denoting by $m = \texttt{gcd}\{m_{i,j}\}$, the degree of parallelism $\texttt{ParDeg} = \lceil m/l \rceil$ is big enough – if the latter is 1 for example, parallelism cannot be extracted, since we intend to execute the connected component's nodes (iterations) on the same thread. The ADDG is finally described by $R|m>$, where $R \subseteq \mathbb{P}(\mathbb{Z}_m \times \mathbb{Z}_m)$ contains the iteration pairs corresponding to the edges of one connected component. (By the pattern's definition, all connected components generate the same S.) For example, the ADDG in Figure 2(a) formula is $(0,1)|8>$.

Constant-Distance Dependency Pattern: The second basic pattern corresponds to the one shown in Figure 2(b). An ADDG of l connected-components satisfies this pattern if letting $m = \texttt{gcd}\{j - i \mid i \rightarrow j \texttt{ is a dependency}\}$ then $l < m$, with m big enough. The ADDG is then described by $R<m>$, where R is the root set of the ADDG (i.e. nodes that are the sink of no dependency).

Periodic Merge Dependency Pattern: This pattern is shown in Figure 2(c). An ADDG satisfies this pattern if there are several, consecutive nodes $([0,3]<128>$

in Figure 2(c)) that are the source/sink of many dependencies, the structure is repetitive, and if eliminating these nodes from the graph results in only singleton nodes. Hence, if iterations $[0, 3]<128>$ are executed sequentially (achieved via synchronization barriers), then iterations ($[4, 127]<128>$) have no dependencies and can be executed in parallel, out of order. Due to space constraints we do not give the formula/unification rules for this pattern here. The intuition is that this pattern is orthogonal with the other two, hence we unify only between instances of this pattern: the semantics is that barriers are enforced while iterations in-between barriers are dispatched to threads as determined by the step/modulo unification formulas. A special case when simplification with a modulo-based pattern is possible and desirable is when the ADDG of this pattern is included in the one of the modulo-based pattern. (The ADDG in Figure 2(c) $\subset \{(i, i)|0 \leq i < 4\}<4>$.) The unification trivially yields the modulo-based formula, thus eliminating the (unnecessary) barrier overhead.

3.6 Further Remarks

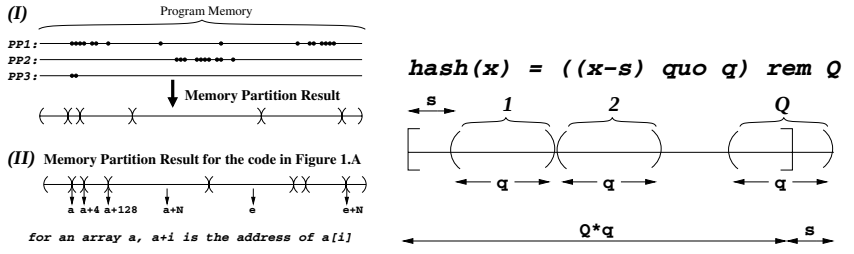
The time complexity of the analysis proposed in this section is dominated by the ADDGs construction phase, which is on average $O(n \log n)$ in the number of profiled addresses[4]. Then, pattern-matching ADDGs to determine formulas is linear in the number of dependencies, while unifying formulas among ADDGs is cheap under the presented algebra. The order in which ADDG formulas are unified is important: a state-of-art framework should aim to assign iteration to threads such that most dependencies are resolved, while preserving the optimal degree of parallelism. Other PPPs whose unification would yield too conservative results are synchronized (see Section 3.2). These heuristics are not discussed here.

Finally, we have discussed our analysis so far in the context of simple-loops. To generalize to loop-nests, we represent iteration numbers in $\mathbb{Z}^p$, where p is the loop's nesting depth. We apply our analysis for the most-outermost loop, L, of suitable TLS granularity, by projecting $\mathbb{Z}^p$ to $\mathbb{Z}$, in the context of L. If the analysis fails to give an acceptable result, we repeat it for inner loops.

4 Memory Partitioning

This section introduces at a high-level the second part of our analyses, whose goals were stated in Section 2.3. Section 4.1 presents how memory is coarse-grained partitioned into disjoint intervals that intuitively correspond to different variables (e.g. arrays) whose accesses were not statically disambiguated. The most suitable type of TLS model is then chosen to protect each of these partitions (see [17] for the software composition techniques). This partitioning facilitates the fine-grained memory partitioning, presented in Section 4.2, by which each individual TLS model's hash function is computed based on coarse-partition's access-patterns. Finally, Section 4.3 presents speed-up results.

[4] Sort the per SPP addresses; constructing the ADDG for a PPP is then linear; we then construct only the ADDG corresponding to PPPs whose address sets overlap.



(a) Memory Partitioning (Coarse) (b) Hash Function – Graphical View

Fig. 3.

4.1 Building Variable-Based Memory Partitions

Figure 3(a).*(I)* intuitively depicts our approach. The profiling phase has yielded the set of addresses accessed at each program point. A clustering technique is then employed to exhaustively partition the memory into mostly-disjoint intervals, where a few exceptions are allowed (e.g. the two "singular" points of `PP1`).

For on-line analysis, when only a window of iterations is profiled, we use a linear formula that predicts how intervals grow. These are used to compute interval boundaries. Note that (i) a SPP may correspond to two intervals (e.g. `PP1`), as it is not required that a variable is laid out contiguously in memory, and (ii) two SPPs may correspond to the same interval/partition (e.g. they may access the same variable). Also, it is preferable to split an interval I into $I_1 \cup I_2$, when we observe that one SPP repeatedly accesses only few addresses in I (e.g. `PP3` forms I_1) – this allows to more aggressively adapt/optimize I_2's TLS model's behavior (small memory overhead within very few false-positives). For safe-languages (Java, C#) an (extra) orthogonal partitioning can be made based on type information or other invariants guaranteeing that two SPP refer to disjoint set of addresses (although their intervals overlap).

Figure 3(a).*(II)* shows the analysis result for the code in Figure 1. We denote by a and e the start address of arrays `a` and `e`. We observe that accesses of `a` form three disjoint partitions: $[a, a+3]$ for `PP5`, $[a+4, a+127]$ for `PP4`, and $[a+128, ...]$ for `PP1`, `PP2` and `PP3`. Accesses of `e` form three intervals: an increasing one starting at e, a decreasing one starting at $e+N$ and a buffer-interval in the middle. While the first two partitions may use aggressive fine-grained partitioning, the middle-one should use a precise partitioning (i.e. hash function is one-to-one).

Having computed the coarse-grained memory partitions, we look at their associated ADDGs to determine the most suitable type of TLS model for them. We keep into account that serial commit models [5, 6, 17] implicitly satisfy WAW and WAR dependencies, while in-place models [16] do not. Also, [17] is inefficient in the presence of many iteration-independent RAW. However, note that if the thread partitioning requires a non-trivial characteristic ($m \neq 0$), then only in-place models may be employed, and only then when a serial phase[5] is not needed.

[5] This might still be required for scalar computation or `IO` operations.

4.2 Fine-Grained Memory Partitioning

For each memory partition we apply a congruence/set analysis that attempts to map addresses accessed by different threads into mostly disjoint (coset-based) equivalence classes. The latter naturally induces the TLS model hash function, and effectively reduces the TLS memory overhead while introducing very few false-positives. An important consequence of this strategy is that it implicitly optimizes the speculative-storage cache-behavior in that the same thread repeatedly accesses the same speculative storage elements (and different ones for different threads) and hence requires mostly $L1$-cache accesses (rather than $L2$).

We are looking for hash functions of the form: $\texttt{hash}_{s,q,Q}(x) = ((x - s)\ \texttt{quo}\ q)\ \texttt{rem}\ Q$, because (i) they are computationally effective (especially when q and Q are powers of 2) and (ii) they enable both an interval and congruence like analysis, which are both useful in reducing the image cardinal Q, and hence the speculative storage size. The hash function can be seen as an equivalence relation among addresses: $a_1 \sim a_2 \Leftrightarrow \texttt{hash}_{s,q,Q}(a1) = \texttt{hash}_{s,q,Q}(a2)$. Figure 3(b) graphically depicts this view: there are Q intervals of equal length q. All the addresses a such that $(a - s)\ \texttt{rem}\ (Q * q)$ belong to the i^{th} interval and are in one equivalence class. This class corresponds to the union of cosets: $(i * q) + Q\mathbb{Z} \cup (i * q + 1) + Q\mathbb{Z} \cup \cdots \cup (i * q + q - 1) + Q\mathbb{Z}$. The use of the offset s is to align equivalence-classes in $\texttt{hash}_{0,1,Q*q}$ so that they can be safely collapsed by interval formation into $\texttt{hash}_{s,q,Q}$.

Intuitively, `hash` should satisfy the invariant that for any two SPPs that may generate a dependency violation (e.g. at least one is a write), the addresses accessed by iterations executed on different threads correspond mainly to different cosets of $\mathbb{Z}_Q$, where a few exceptions can be accommodated. In general this problem is computationally expensive to solve; although not presented here, we have developed a guided-search heuristic that, although does not guarantee an optimal solution, for all practical cases we encountered, it does so and is linear in the number of profiled addresses.

4.3 Speed-Up Results

Table 1 shows speed-up results, computed as the ratio between sequential and parallel timings, for several applications selected from the BYTEmark and SciMark benchmarks (see also [16]). All the tests were performed on a SMP Sun

Table 1. Speed-ups: Sequential / Parallel Time Ratio(4 Processors)

Seq/Parallel	HandPar	OptROHash	OptHash	Naive	OneToOne
IDEA DeKey	3.83	2.78	2.44	0.96	0.65
IDEA Cipher	3.87	3.22	1.44	1.11	0.95
NeuralNetBW	1.64	1.15	1.07	0.90	0.25
NeuralNetFW	2.04	1.65	1.46	0.15	0.11
SparMatMult	2.11	1.93	1.60	0.57	0.13
FFT	2.02	1.90	1.90	0.66	0.66

machine with 8 Gb RAM memory, and four Opteron 850 processors, running Fedora Core 4. We used the gcc3.4.4 compiler at -O2 optimization level.

The *second column* represents the speed-up achieved for optimal, hand-based parallelization (no TLS overhead). The *third column* corresponds to applying TLS via the dynamic analysis presented in this paper. We have used two TLS models: SpLIP – an in-place model [16] and SpRO – the read-only model in which a read just *reads* the value (no time/space overhead), while a write causes a rollback followed by a sequential fix-up. SpLIP models use optimized `hash`es.

The `OptHash` column refers to the case when only SpLIP is employed, however on optimized `hash`es. This serves as base of comparison with the last two columns. The column `Naive` refers to a naively chosen `hash`, in which `q=s=0` and `Q` is roughly the size of the range of addresses accessed by concurrent iterations. The last column uses one-to-one `hash`es: `q=s=0` and `Q` is roughly the data-space size. (For `FFT` and `NeuralNetFW` columns 5 and 6 use roughly the same Q.)

The differences between optimized and (i) naively chosen and (ii) one-to-one `hash`es are significant for all tested applications. The reasons are: (i) the near-ideal cache behavior of optimized `hashes` (column 4 vs 5) and (ii) the small(er) memory overhead. The results for `IDEA DeKey` and `SparseMatMult` look somewhat surprising in that the differences between columns 3 and 4 are more pronounced than in the other cases. The reason is that the read-to-write ratio is very high (and SpRO is very effective). These differences also appear for columns 4 vs 5 because the naive version suffers from read-contention (concurrent cache eviction due to writes to TLS's meta-data), while optimized `hash`es do not. A final observation is that when the application features bad cache-locality (last two benchmarks), but still a regular behavior, we can expect to obtain close to optimal (hand-parallelized) speed-up because the TLS overhead is furthermore amortized by the negative memory-hierarchy effects of the original program.

5 Conclusions

We have shown how dynamic analysis of addresses accessed during a loop can be used to facilitate thread-level speculation. First, we have presented an algebra for partitioning the iteration-space to threads such that repetitive dependencies are resolved and rare dependencies are ignored. Second, we have used dynamic analysis to fine-tune the TLS model to exploit code access patterns, as opposed to previous work, which has concentrated on optimizing the code for one TLS model. We achieved the latter by using coarse-grained followed by fine-grained partitioning of the data space; now *several optimized* TLS model instances are employed to parallelize an application, instead of only *one, over-arching* model. Finally, we have presented results that validate the utility of our analysis.

References

1. Bhowmik, A., Franklin, M.: A General Compiler Framework for Speculative Multithreading. In: SPAA 2002 Proceedings. ACM, New York (2002)
2. Chen, M.K., Olukotun, K.: The Jrpm System for Dynamically Parallelizing Java Programs. In: ISCA-30 (June 2003)

3. Chilimbi, T.M., Davidson, B., Larus, J.R.: Cache-Conscious Structure Definition. In: PLDI 1999 (1999)
4. Chilimbi, T.M., Larus, J.R.: Using Generational Garbage Collection to Implement Cache-Conscious Data Placement. In: ISMM (1998)
5. Cintra, M., Llanos, D.R.: Toward Efficient and Robust Software Speculative Parallelization on Multiprocessors. In: PPoPP 2003, San Diego, California (June 2003)
6. Dang, F., Yu, H., Rauchwerger, L.: The R-LRPD Test: Speculative Parallelization of Partially Parallel Loops. In: IPDPS (2002)
7. Dou, J., Cintra, M.: A Compiler Cost Model for Speculative Parallelization. ACM TACO 4(2) (June 2007)
8. Fraser, K., Harris, T.: Concurrent Programming Without Locks. In: TOCS (2007)
9. Granger, P.: Static Analysis of Linear Congruence Equalities among Variables of a Program. In: Abramsky, S. (ed.) CAAP 1991 and TAPSOFT 1991. LNCS, vol. 493. Springer, Heidelberg (1991)
10. Hammond, L., Willey, M., Olukotun, K.: Data Speculation Support for Chip Multiprocessor. In: ASPLOS (1998)
11. Kim, S., Ooi, C., Eigenmann, R., Falsafi, B., Vijaykumar, T.: Reference Idempotency Analysis: A Framework for Optimizing Speculative Execution. In: PPOPP 2001 (2001)
12. Liu, W., Tuck, J., Ceze, L., Ahn, W., Strauss, K., Renau, J., Torrellas, J.: POSH: A TLS Compiler that Exploits Program Structure. In: PPoPP 2006 (2006)
13. 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, pp. 641–650. Springer, Heidelberg (2007)
14. Masdupuy, F.: Array Operations Abstraction Using Semantic Analysis of Trapezoid Congruences. In: ICS 1992 (1992)
15. Miné, A.: The Octogon Abstract Domain. Higher-Order and Symbolic Computation Journal 19 (2006)
16. Oancea, C.E., Mycroft, A.: A Lightweight, In-Place Model for Software Thread-Level Speculation, email Cosmin.Oancea@cl.cam.ac.uk
17. Oancea, C.E., Mycroft, A.: Software Thread-Level Speculation – An Optimistic Library Implementation. In: IWMSE (2008)
18. Rundberg, P., Stenström, P.: An All-Software Thread-Level Data Dependence Speculation System for Multiprocs. Journal of Instruction-Level Parallelism (1999)
19. Sazeides, Y., Smith, J.E.: The Predictability of Data Values. In: MICRO 30: 30th International Symposium on Microarchitecture, pp. 248–258 (1997)
20. Shpeisman, T., Menon, V., Adl-Tabatabai, A., Balensiefer, S., Grossman, S., Hudson, R., Moore, K., Saha, B.: Enforcing Isolation and Ordering in STM. In: PLDI 2007 (2007)
21. Sohi, G.S., Breach, S.E., Vijaykumar, T.N.: Multiscalar Processors. In: ISCA-22, pp. 414–425 (June 1995)
22. Steffan, J.G., Colohan, C.G., Zhai, A., Mowry, T.: A Scalable Approach for Thread Level Speculation. In: ISCA-27 (2000)
23. Zhai, A., Colohan, C.B., Steffan, J.G., Mowry, T.C.: Compiler Optimization of Scalar Value Communication Between Speculative Threads. In: ASPLOS X (2002)
24. Zilles, C., Sohi, G.: Master/Slave Speculative Parallelization. In: Micro-35 Proceedings. ACM, New York (2002)

Thread Safety through Partitions and Effect Agreements

Nicholas D. Matsakis and Thomas R. Gross

ETH Zurich

Abstract. This paper describes a safety analysis for a multithreaded system based upon transactional memory. The analysis guarantees that shared data is always read and written from within a transaction, while allowing for unsynchronized access to thread-local and (shared) read-only data, as well as the migration of data between threads. The analysis is based on a type and effect system for object-oriented programs called *partitions*. Programmers specify a partitioning of the heap into disjoint regions at a field-level granularity, and then use this partitioning to enforce safety properties in their programs. Our flow-sensitive effect system requires methods to disclose which partitions of the heap they will read or write, and also allows them to specify an *effect agreement* which can be used to limit the conditions in which a method can be called.

1 Introduction

The Java language's traditional, lock-based model of parallel programming has proven to be vulnerable to a number of serious problems, such as deadlock and priority inversion. Programmers are forced into a difficult trade-off between correctness and performance. While a coarse-grained locking scheme is the easiest to implement and verify, fine-grained locking – or even nonblocking algorithms – is required to take full advantage of the parallel hardware.

Recent research in transactional memory [1] offers an attractive alternative. Programmers can use `atomic` statements to group together operations which must – for reasons of correctness – be executed without interruption, and the runtime system will ensure efficient and correct execution.

As convenient as transactional memory is, it is not a panacea. One obvious problem that programmers will face is verifying that they use `atomic` sections consistently. For example, shared objects should only be modified within a transaction, while thread-local and read-only objects may be used freely. Because mainstream programming languages today offer no means to controlling aliasing, verifying that each object is used safely must be done manually and is error-prone.

In this paper, we present a technique that guarantees consistent usage of transactions within a program. Our work is done in the context of a Java-like language augmented with transactional memory. It supports a number of common threading patterns, including thread-local data, read-only data, and the transfer of objects between threads.

J.N. Amaral (Ed.): LCPC 2008, LNCS 5335, pp. 172–186, 2008.

Our technique is based on two new language constructs:

1. *Partitions* extend the type system so that it can describe aliasing at a very fine-grained level. Programmers use partitions to expose the aliasing structure of their program. A flow-sensitive effect system then is used to determine which partitions may be read or written by the program at different points in time.
2. *Effect agreements* allow a method to constrain the effects of its caller, while preserving modular compilation. For example, a method which hands off data in a particular partition to a new thread may prevent its caller from modifying that data while the new thread is executing.

The paper begins with an introduction to partitions and discusses how they are integrated into the type system. We then discuss the effect system, and in particular effect agreements, and give a brief example which uses them to verify the thread safety of a simple web server. Finally, we present the algorithm used to check that effect agreements are respected and close by showing how to extend the `Thread` class so as to enforce safe threading practices throughout the program.

2 Partitions at a Glance

A *partition* is a compile-time abstraction that describes a portion of the heap at a field-level granularity. In other words, if we define the heap $\mathcal{H}(o, f) \mapsto o$ as a mapping from an (object id, field) pairs to another object id, a partition is a set of such (object id, field) pairs. Partitions are similar to memory regions, except that we do not use them for memory management, but rather for alias tracking.

In our system, class and method definitions are parameterized by a set of *partition parameters*, analogous to generic type variables. When the class is instantiated or the method is invoked, each partition parameter will be mapped to a fixed partition.

2.1 Code Example

To demonstrate how partitions are integrated into the language, Figure 1 gives the definition of a class `IntWrapper`. `IntWrapper` has a single partition parameter, named `P`, which is indicated by the `@P` which follows the class name. It contains a single field `field` located in the `P` partition, and two accessors `get()` and `set()`. The partition of `field` is indicated by the `@P` preceding its type.

The `clone()` method of Figure 1 demonstrates many important points:

1. Methods can have partition parameters in addition to classes. In this case, the values for these parameters are inferred when the method is invoked.
2. New partitions can be created via a statement like `new @R`, where the name `R` of the fresh partition should not shadow any other partitions in scope.
3. When new instances of a class are created, concrete values must be given for each partition parameter of the class. In this case, the expression `new IntWrapper@Q()` creates an `IntWrapper` instance whose `P` partition parameter is bound to `Q`, the partition parameter of the `clone()` method.

```
class IntWrapper @P {
    // Leading @P indicates this field is in partition P.
    @P int field;

    // Primitive types require no partitions.
    int get()       { return field; }
    void set(int i) { field = i;    }

    // Methods can be parameterized with partition variables too.
    @Q IntWrapper@Q clone() {
        new @R; // demonstrate syntax to create a new partition
        IntWrapper@Q res = new IntWrapper@Q();
        res.set(field);
        return res;
    }
}
```

Fig. 1. Introductory example showing how partitions are integrated into the syntax

Although this example does not take advantage of it, it is also possible to take the union of several partition parameters. For example, the `new` statement in the `clone()` method could have been written `new IntWrapper@(P∪Q)` instead. Taking the union of several partitions simply refers to a larger partition which always contains every field in each of its components.

Like Java generics, JPart types are non-variant with respect to their partition parameters. This means that two types `C@P` and `C@Q` – both of which refer to the same class, but with different partition parameters – are not subtypes of one another, even if `P` $\subseteq$ `Q`. As a result, partition parameters can be referenced in both co-variant positions, such as return types, or contra-variant positions, such as the types of method parameters.

2.2 Growing Partitions

Data is added to a partition by creating new objects that contain fields located in that partition. All fields exist in exactly one partition. However, when multiple partitions are unioned together, it may not be known statically precisely which partition a field is located in. For example, if the `new` statement of Figure 1 were modified to read `new IntWrapper@(P∪Q)`, then it is not defined whether the field `field` of the resulting `IntWrapper` instance is placed in `P` or `Q`; the type checker must conservatively assume that it may be in either.

2.3 Disjoint Partitions

In general, we do not require that the partition parameters be bound to disjoint partitions when a class is instantiated or a method is invoked. However, in some cases disjoint partitions will be required by the effect checker to prove that a program is safe. Therefore, we allow the programmer to add *disjoint declarations* which name sets of partition parameters that must be mutually disjoint. An example appears in Figure 6. Note that partitions created within the current method are always known to be disjoint from all other partitions.

2.4 Partition Transfer

Creating a new object is not the only way to add data to a partition. It is also possible to change the partitioning of an existing object, so that its fields move from one partition to another. This is called *partition transfer*, and it necessitates changing the type of the object whose fields were moved to reflect the new partitioning.

Partition transfer is denoted via a cast to a type with the new partitions. The old partitions are determined from the type of the expression being cast. As an example, consider the following cast:

```
C@P variable = ...;
C@Q transferred = (C@Q) variable;
```

This has the effect of transferring any field in partition `P` that belongs to an object reachable from `variable` into the partition `Q`.

In the presence of aliasing, partition transfer is not type safe. This is because there may be extant aliases to the object whose fields were transferred, and those aliases will still have the original type. If any code should use one of those aliases, then it would assume that the object's fields were still in the original partition, when in fact they have been moved.

Rather than trying to prevent aliasing, we solve this by using the effect system to prevent the old partition from being read or written after data has been transferred out of it. Therefore, a partition transfer from `P` to `Q` means that `P` can never be used again. Although this restriction may seem draconian, it nonetheless enables a number of usage patterns and in particular allows data to be moved between threads.

3 Language Extensions for Parallelism

Before we proceed to discuss our effect system, it is necessary to say a few words about the threading model which we assume. As in Java, threads are created by creating a new instance of some subclass of the class `Thread`. We have chosen to use transactional memory rather than locks as the basis for synchronization. Furthermore, we have elected to replace Java's `join()` methods with a simpler, lexically scoped mechanism. To that end, we introduce two new kinds of statements:

1. The `atomic` statement, written `atomic {...}`, guarantees that all of its substatements will execute atomically, meaning without interruption by any other thread. Atomic statements are discussed in detail in [2].
2. The `forkjoin` statement, written `forkjoin {...}`, executes its substatements and dynamically tracks the set of threads which they start. Once all substatements have executed, the `forkjoin` statement waits for the threads which were started within to finish before it continues. It must wait not only for the threads which it has directly started, but also for any threads that

they may have transitively begun themselves. `forkjoin` statements are simpler than Java's mechanism `join()` methods, but can still express real-world examples of Fork/Join parallelism [3,4].

4 Effects

Partitions allow the program's data to be divided into distinct logical sections, but the effect system regulates how those partitions may be used.

Figure 2 shows the `IntWrapper` class defined earlier, but with each method annotated to show what effects it may have. Effect declarations precede each method declaration and are part of the method's interface. `get()` is annotated with `!Rd(P)`, as it reads a field in partition `P`, while `set()` has a corresponding write effect. Finally, `clone()` has both a `!Rd(P)` effect, as it reads `field`, and a `!Wr(Q)` effect, as it invokes `set()` on `res`, whose field is located in `Q`.

Effects are generated in two cases: dereferencing pointers and partition transfers. When dereferencing a pointer, the kind of effect depends on whether the field is read or written, and whether the modification takes place in an atomic section. The resulting four kinds of effects are `Rd(P)` and `Wr(P)`, for plain reads and writes, and `ARd(P)` and `AWr(P)`, for reads and writes which take place in an atomic statement. In each case, the parameter `P` indicates the partition that is affected, and may be either a single partition variable, or the union of several variables. Partition transfers out of a partition `P` are indicated by an effect `Xfer(P)`.

In addition to directly modifying fields, effects may be generated indirectly by invoking other methods. In this case, the effect checker uses the effects from the method's declaration to conservatively estimate the set of effects the method invocation may have. Note that if the method invocation takes place within an `atomic` section, plain `Rd` and `Wr` effects are translated into their atomic equivalents.

It is the programmer's responsibility to add annotations which declare how a method may affect the in-scope partition parameters. The effect checker statically ensures that the method body cannot affect any partition parameter in

```
class IntWrapper @P {
    @P int field;

    !Rd(P) int get()       { return field; }
    !Wr(P) void set(int i) { field = i;    }

    @Q !Rd(P) !Wr(Q) IntWrapper@Q clone() {
        new @R;
        IntWrapper@Q res = new IntWrapper@Q();
        res.set(field);
        return res;
    }
}
```

Fig. 2. The class `IntWrapper` which was shown before, annotated with effects

a way that is not declared. It is not necessary to declare effects for partitions which are created within the method.

4.1 Effect Agreements

While effect declarations allow a programmer to limit the effects a method may have, it is sometimes useful for a method to be able to limit the effects of its caller. For example, transferring data from a partition `P` to a partition `Q` is only allowed when it can be guaranteed that `P` is not used after the transfer. If `P` is a partition parameter, however, the method needs some way to convey to its caller that it has invalidated `P` and that `P` should not be used from that point forward.

To enable these sort of guarantees, methods may be annotated with *effect agreements* that constrain what can happen after a method returns. These effects form a kind of agreement between the caller and the callee. We use a flow-sensitive effect checker to enforce them statically.

Agreements are different from Design By Contract [5]. In DBC, methods declare *preconditions*, which the caller must guarantee for the callee to function properly, and *postconditions*, which the callee promises to bring about or maintain. In this way, contract obligations flow in both directions.

In contrast, all effect agreements are obligations the callee imposes on the caller. These obligations always take the form of effects which are not permitted. Each effect agreement has a *time span* that determines precisely when the events are not permitted to occur.

In this paper, we use two different time spans for effect agreements, `post` and `par`. A `post` agreement, written `post -F(`w`)`[1], indicates that the effect `F(`w`)` may not occur at any point in the program execution after the method returns.

`post` agreements are used to ensure the safety of partition transfer. Transferring data from a partition `P` to a partition `Q` imposes a `post` agreement upon the current method, beginning at the point of transfer. The exact agreement is `post -Rd(P) -Wr(P) -ARd(P) -AWr(P) -Xfer(P)`. If the `P` partition was not created by the current method, this may require the method to declare a similar agreement so as to constrain its caller.

While sometimes necessary, `post` agreements are often stronger than is required. `par` (short for parallel) agreements can be used to limit the agreement to a shorter time span. A `par` agreement, written `par -F(`w`)`, indicates that a parallel thread has been started which requires that no effect `F(`w`)` occurs for the duration of its lifetime.

Generally, we do not know when a thread will finish executing, and in these cases a `par` agreement is equivalent to a `post` agreement, as depicted graphically in Figure 3a. In the presence of a `forkjoin` statement, however, the thread's lifetime can be bounded. This is depicted in Figure 3b, where the caller knows that the thread will finish before or by the end of the `forkjoin` region.

Effect agreements are considered binding on the current thread; however, a thread is also responsible for the behavior of any thread which it (transitively)

[1] The reason we use a minus sign `-` and not a `!` before the effect is to indicate that agreements describe effects which are forbidden, not permitted.

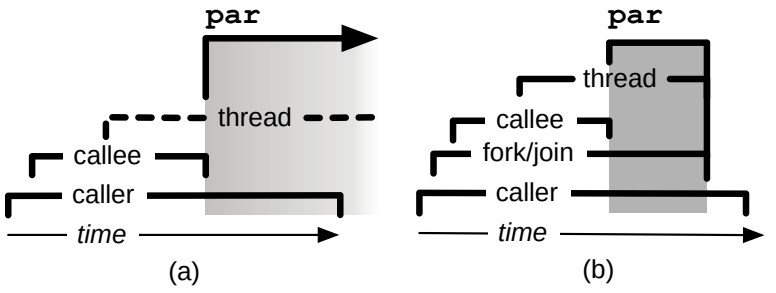


Fig. 3. A timeline demonstrating the time span to which a `par` effect agreement applies, both (a) normally and (b) in the presence of a `forkjoin` region

starts. Therefore, if a thread `T` invokes a method which prohibits it from later writing to a partition, then `T` may not start another thread which writes to that same partition.

4.2 Effects and Inheritance

Because effects and effect agreements form part of the interface of a method, it is important to describe how they interact with inheritance. Because it must be safe for any subtype to be used where a supertype is expected, overriding methods may not (a) have more effects than the methods they override; or (b) prohibit effects via effect agreements which are allowed in the respective supertypes. The type checker verifies these conditions statically.

4.3 Example: Multithreaded Server

One common multithreaded application is a server. Figure 4 shows a simple server which has one thread listening for connections on a given port, defined by the class `ListenerThread`. When a connection arrives, the server initializes an object describing the new connection and creates a `HandlerThread` to handle it. The handler is given control of the connection object and started in parallel. From that point on, the connection object is considered thread-local data for the handler thread, and should not be used by the listening thread anymore. In this example, we show how the effect checker, combined with effect agreements, can be used to verify that the connection object is safely transferred to the new thread. In Section 6, we will expand the technique shown here into a more general solution.

The `ListenerThread` class does not have any partition parameters. Instead, within the `run()` method it creates a fresh partition, `P`, which contains the listening `Socket` instance. The `HandlerThread` class is parameterized by a single partition `L` for it's thread-local data. In its `accept()` method, the `ListenerThread` creates a new partition `H` which it gives to each new `HandlerThread` to use for its thread-local data.

```
class ListenerThread
    extends Thread
{
    public void run() {
        new @P;
        Socket@P socket =
          new Socket@P();
        while(true)
            accept(socket);
    }

    !Rd(P) !Wr(P)
    @P void accept(Socket@P socket) {
        new @H;
        Socket@H connection =
          socket.accept();
        init(connection);
        new HandlerThread@H(
            connection).start();
    }

    !Wr(H)
    @H void init(Socket@H conn)
    { ... }
}
```

```
class HandlerThread@L
    extends Thread
{
    @L Socket@L connection;

    HandlerThread(Socket@L c) {
        connection = c;
    }

    !Rd(L) !Wr(L)
    par -Rd(L) -Wr(L)
    par -ARd(L) -AWr(L)
    public void start()
    { ... }

    !Rd(L) !Wr(L)
    public void run()
    { ... }
}
```

Fig. 4. The skeleton of a simple server which spawns a new thread to handle each incoming request

The error we are trying to prevent is that the `ListenerThread` continues to write to the partition `H` after it has started the `HandlerThread`. To prevent this, the `HandlerThread` has declared effect agreements on its `start()` method which prohibits its local partition `L` (`H`, from the `ListenerThread`'s point of view) from being read or written. These agreements are given `par` scope so that they are in effect as long as the thread may execute.[2]

To verify that effect agreements are respected, the effect checker computes what effects may occur after each statement in the method. The result of this computation for the `accept()` method of `ListenerThread` is shown in Figure 5.

Since we are computing what events are to come, the analysis is done starting at the end of the method and working backwards. Therefore, line 12 depicts the initial set of effects. Because there are no effect agreements declared on this method concerning `P`, we must make the conservative assumption (which happens to be true, in this case) that data in partition `P` may be both read and written in the future. Note that there no conservative assumptions are needed for `H` as it is newly created in this method.

Line 10 contains the call which starts the `HandlerThread`. This is where we must verify the effect agreements for `start()`: to do so, we compare the set of events to come from line 11 with the forbidden events, and determine that the method call is permitted. Since `start()` declares that it reads and writes

[2] A sharp-eyed reader will note that, because the `start()` method is inherited from `Thread`, `HandlerThread` cannot add effect agreements in this fashion. We resolve this in Section 6 by modifying the thread class itself.

```
1  !Rd(L) !Wr(P)
2  @P void accept(Socket@P socket) {
3      // { Rd/Wr(P) }
4      new @H;
5      // { Rd/Wr(P) Rd/Wr(H) }
6      Socket@H connection = socket.accept@H();
7      // { Rd/Wr(P) Rd/Wr(H) }
8      init@H(connection);
9      // { Rd/Wr(P) Rd/Wr(H) }
10     new HandlerThread@H(connection).start();
11     // { Rd/Wr(P) }
12 }
```

Fig. 5. The results of a flow-sensitive effect analysis of the `accept()` method from Figure 4

`HandlerThread`'s partition parameter, we determine that `H` may be both read and written at this point and add those effects to the set, yielding line 9.

From line 9 back to line 5, the effect set is unchanged because it already contains reads and writes. No methods are invoked which declare an effect agreement, so there is no need to check for conflicts. Finally, we reach line 4 which creates the `H` partition and therefore remove the `Rd/Wr(H)` effects, leaving only effects on `P` in line 3.

5 Effect Checker

This section describes the design of the effect checker, the module responsible for verifying that methods properly declare any effects they may have and that they abide by any effect agreements due to method calls or partition transfers.

5.1 Checking the Method Interface

Verifying that the method implementation obeys the bounds of its interface is done with a standard, flow-insensitive analysis. We simply take the union of all effects generated by any statement in the method body, and verify that, for each effect `F(P)`, either (a) `P` is a created by a `new @P` statement in this method, or (b) the method declares an effect `F(Q)`, where $\texttt{P} \subseteq \texttt{Q}$.

5.2 Flow-Sensitive Effect Analysis

Because effect agreements constrain the events which can occur within a specific period of time, a flow-sensitive analysis is required to check them. For this purpose, we use an iterated analysis that propagates sets of effects around the control flow graph until it reaches a fixed point. Similar algorithms are commonly used for data-flow analysis in compilers [6]. Because effect agreements constrain what happens after a method returns, we use a reverse analysis, with set union as the confluence operator for joining multiple control flows.

The effect flow algorithm is shown as Algorithm 1. This algorithm is in fact executed twice, once for `par` effect agreements and once for `post` agreements.

Algorithm 1. effect-flow

```
OUT ← { ∅ for each block }
initialize OUT with starting assumptions
while OUT continues to change do
  for all blocks b do
    f_succ ← ∪(OUT_b' for each b' ∈ successors of b)
    f_stmts ← the set of effects caused by any statement in b
    if checking a par agreement and b terminates a basic block then
      f_stmts ← ∅
    end if
    OUT_b ← f_succ ∪ f_stmts
  end for
end while
```

For each basic block b, it computes a set $\texttt{OUT}_b$ which contains the set of effects that could occur in the current time span. When checking `post` agreements, therefore, $\texttt{OUT}_b$ contains the set of effects that can occur once b begins execution. When checking `par` agreements, on the other hand, it contains the set of effects that might occur in parallel with a thread that starts right before the control flow reaches b.

Computing the `OUT` set for a basic block b begins by taking the union of the `OUT` sets for each successor of b (except for a slight twist regarding `forkjoin` statements, discussed in detail below). The effects which may occur due to statements within b are then added to this set, resulting in the new value for $\texttt{OUT}_b$. This process repeats for all basic blocks until it reaches a fixed point.

5.3 Starting Conditions

To start the algorithm, the initial set of effects at the end of the method are defined conservatively. We assume that any effect may occur which is not specifically forbidden by an effect agreement of the appropriate time span. Therefore, when performing `post` analysis, the starting set contains all possible effects except those barred by some `post` agreement. Likewise, during the `par` analysis, only those effects barred by `par` agreements are removed from the starting set.

5.4 `forkjoin` Statements

When checking `par` agreements, the `forkjoin` statement is given a special significance. This is because it is statically known that all threads beginning within a `forkjoin` statement will have terminated by the time the `forkjoin` statement finishes. This means that, when checking `par` agreements, the basic block which terminates the `forkjoin` statement may safely disregard the effects of its successors, because it is known that they will not occur in parallel with any threads that start within.

6 Enforcing Thread-Safety

We can use effect agreements to ensure that a partition is never accessed by multiple threads in an incompatible fashion. The core idea is to use the type of the `Thread` object, from which all threads must derive, to guarantee that threads only use partitions in one of several pre-approved ways.

We guarantee that all partitions the thread may access fall into one of three categories:

1. *Thread-Local* partitions are read and written by the thread outside of `atomic` sections. Such partitions may not be simultaneously accessed by other threads.
2. *Read-Only* partitions are never written by the thread and are read outside of `atomic` sections. Such partitions may not be written by other threads.
3. *Shared* partitions are modified by multiple threads simultaneously. Accesses to shared partitions, read or write, must take place within an `atomic` section.

We assume that threads are started by invoking the method `start()` defined in the `Thread` class. The actions of a thread are defined by its `run()` method; invoking `run()` directly, however, does not start the thread, but is merely a normal method call that runs in the current thread.

Therefore, we can use the declared interface on the `start()` and `run()` methods to control what threads are permitted to do. The desired interface for class `Thread` is shown in Figure 6. The idea is to parameterize the thread by three partitions, `S`, `R`, and `L`, which contain respectively the shared, read-only, and thread-local data that this thread may access. As we will see, the definition of the `run()` and `start()` methods do not permit data in these partitions to be used in any way other than those outlined above. Furthermore, because a method must list all of its effects in its interface, we can be sure that this thread will not affect any partitions other than `S`, `R`, or `L`.

To get a better understanding of the definition in Figure 6, let us examine it piece by piece. The first line declares the partition parameters, `S`, `R`, and `L`, and states that they must be mutually disjoint.

The `run()` method is defined on lines 3–7. The effect declarations which precede it describe how the thread is allowed to access `S`, `R`, and `L`. Line 3 indicates that any partition may be atomically read, whereas line 4 restricts atomic writes to shared (`S`) and thread-local (`L`) data. Line 5 allows read-only (`R`) and thread-local (`L`) data to be read non-atomically, but only `L` may be modified in a non-atomic fashion, as indicated on line 6.

The `start()` method is defined next. Lines 9–12 indicate that the start() method has the same effects as the run() method. Lines 13–17 define the effect agreement for the start() method. These agreements highlight the important difference between start() and run(): invoking run() does not actually start a second thread. start(), however, performs its actions in parallel with the current thread, and therefore it has to place constraints on the current thread. Note the use of the par time span for these effect agreements, which guarantees that the forbidden events will not occur in parallel with this thread, though they may occur after the thread is known to have terminated.

```
1  abstract class Thread@S@R@L disjoint(S,R,L) {
2
3      !ARd(S ∪ R ∪ L)
4      !AWr(S ∪ L)
5      !Rd(R ∪ L)
6      !Wr(L)
7      abstract void run();
8
9      !ARd(S ∪ R ∪ L)
10     !AWr(S ∪ L)
11     !Rd(R ∪ L)
12     !Wr(L)
13     par −Wr(S ∪ R ∪ L)
14     par −Rd(S ∪ L)
15     par −AWr(R ∪ L)
16     par −ARd(L)
17     par −Xfer(S ∪ R ∪ L)
18     abstract void start();
19
20 }
```

Fig. 6. The definition of legal effects for a `Thread`. Using these effects guarantees that all partitions modified by a thread are used in a safe fashion.

Line 13 guarantees that the parent thread does not write non-atomically to any partition that the child thread has access to. Line 14 guarantees that the parent thread does not try to read non-atomically from any partition which the child thread will be writing to. Line 15 guarantees that the parent thread will not make atomic writes to the `R` or `L` partitions. Line 16 guarantees that the thread-local data is not atomically read by the parent thread. Finally, line 17 guarantees that no data is transferred out of the shared, read-only, or local partitions while the thread is active.

At first, it might seem stringent to require that every `Thread` class describe their data in exactly three partitions. However, due to the possibility of using partition expressions as parameters, this is not a real limitation. For example, to define a thread which has a shared partition `S`, no read-only partition, and two local partitions, `L1` and `L2`, one can extend `Thread@S@∅@(L1∪L2)`.

6.1 Example: Map Reduce

Figure 7 shows a more involved example following the well-known map reduce pattern [7]. The map reduce pattern uses multiple threads to perform some mapping operation on each item in an array in parallel. After all the threads have completed, it then performs a reduce action which combines the output of the map stage.

The example contains two classes `Main` and `MapThread`. The `Main` class is given a list of objects `list`, contained in some partition `A`, and creates a `MapThread` instance for each item contained within. All of the threads share access to the `O` partition, which is where they will output the mapped results. They also share read-only access to the `A` partition containing the input array and its items. Finally, a new partition `L` is created for each thread and used to store its local data.

```
class Main {
  @A !Rd(A)
  void mapreduce(List@A list) {
    new @O;
    List@O results = new List@O();
    forkjoin {
      for (Object@A obj : list) {
        new @L;
        MapThread@O@A@L mt =
          new MapThread@O@A@L();
        mt.inObject = obj;
        mt.results = results;
        mt.start();
      }
    }
    // reduce results,
    // without atomic
  }
}
```

```
class MapThread@S@R@L
  extends Thread@S@R@L
{
  @L Object@R inObject;
  @L List@S results;

  !Rd(R∪L) !Wr(L) !ARd(S) !AWr(S)
  void run() {
    new @T;
    // Map inObject to tmp:
    Object@T tmp = ...;
    // Partition transfer:
    Object@S tmp2 = (Object@S) tmp;
    // Add to output array:
    atomic { results.add(tmp2); }
  }
}
```

Fig. 7. Map-reduce example

The `MapThread`'s `run()` method begins by creating a temporary partition, `T`. It then performs its mapping operation, creating an output value `tmp` located in the partition `T`. In order to give the main thread access to `tmp`, it transfers the object into the shared partition and adds it to the output list `results`. The actual modification to the shared list must take place within an atomic section.

Note that, once the `MapThread`s have completed, the `Main` class is able to access the `results` array and the objects within without synchronization. This is due to the `forkjoin` region, which ensures that all parallel threads have finished, and therefore limits the scope of the effect agreements by which `Main` is bound.

7 Related Work

The work in this paper is in many ways a synthesis of several existing techniques, and therefore touches on many different bodies of work. As we do not have the space to do justice to all of it, we focus here on that which is most closely related.

Guava [8] is a dialect of Java that does not permit data races by construction. Guava's constructs and type system embody several safe, best practices for multi-threaded program. In this sense, it is similar to the restrictions we placed on the `Thread` class, which require that each partition be categorized into one of several pre-approved patterns.

RccJava [9] is a static type checker for Java programs which is able to detect race conditions. Its type system is based on locks, and the tool has been used to verify an impressive body of existing code. Our flow-sensitive effect analysis is able to capture patterns that they cannot, however. For example, we allow all objects to start as thread-local when first created, but transition to a shared state when they become reachable from another thread.

SafeJava [10] enforces the consistent use of a deadlock-free locking discipline through their ownership type system. The locking disciple is based on the ownership structure, and requires that an object's owner be locked before the object

can be accessed. This approach entangles encapsulation and threading, which may require that one or the other be compromised. Fine-grained locking schemes, for example, require a flat ownership structure, which then provides weaker encapsulation guarantees. In contrast, our approach strives to separate the partitioning structure of the program from the thread safety check.

Effect systems have a long history in the literature, beginning with the work of Lucassen and Gifford [11]. Our treatment of effects has much in common with other object-effect systems [12,13] and in particular with the flow-sensitive effect system described in Contextual Effect Systems [14]. One important distinction of our work from prior work is that we include a mechanism for the programmer to describe not only the effects of a method, but also the potential causes of interference between methods.

Another approach to controlling effects is through the use of capabilities, which limit how a particular reference can be used [15,16,17]. For example, such capabilities might prevent writes or enforce uniqueness.

Our approach to partition transfer is similar in spirit to [18], which enforces uniqueness not by forbidding aliases, but by requiring that all aliases are dead at the point where an object must be unique.

8 Conclusion

In this paper, we have presented *partitions*, an abstraction for exposing the alias structure of a program, along with an accompanying flow-sensitive effect system with effect agreements. We also detail how our effect system can be used to check that a multi-threaded program uses safe patterns for its synchronization.

Our long-term goal is to give programmers a simple and expressive way to check semantic properties of their own design. Rather than encoding a specific notion of correctness into the type system, we aim to develop generic mechanisms, such as partitions and effect agreements, that can be reused for a variety of purposes.

References

1. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. SIGARCH Comput. Archit. News 21, 289–300 (1993)
2. Harris, T., Fraser, K.: Language support for lightweight transactions. SIGPLAN Not. 38, 388–402 (2003)
3. Bull, J.M., Kambites, M.E.: JOMP–an OpenMP-like interface for Java. In: JAVA, pp. 44–53. ACM, New York (2000)
4. Lea, D.: A Java fork/join framework. In: JAVA, pp. 36–43. ACM, New York (2000)
5. Meyer, B.: Object-Oriented Software Construction. Prentice Hall PTR, Englewood Cliffs (2000)
6. Aho, A.V., Ullman, J.D.: Principles of Compiler Design. Addison-Wesley Longman Publishing Co., Inc., Amsterdam (1977)
7. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: OSDI, USENIX Association (2004)

8. Bacon, D.F., Strom, R.E., Tarafdar, A.: Guava: a dialect of Java without data races. In: OOPSLA, pp. 382–400. ACM, New York (2000)
9. Abadi, M., Flanagan, C., Freund, S.N.: Types for safe locking: Static race detection for java. ACM Trans. Program. Lang. Syst. 28, 207–255 (2006)
10. Boyapati, C.: Safejava: a unified type system for safe programming. Ph.D thesis, Massachusetts Institute of Technology (2004); Supervisor-Martin C. Rinard
11. Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. In: POPL 1988, pp. 47–57. ACM, New York (1988)
12. Greenhouse, A., Boyland, J.: An Object-Oriented Effects System. In: Guerraoui, R. (ed.) ECOOP 1999. LNCS, vol. 1628, pp. 205–229. Springer, Heidelberg (1999)
13. Rustan, K., Leino, M.: Data groups: specifying the modification of extended state. In: OOPSLA, pp. 144–153. ACM, New York (1998)
14. Neamtiu, I., Hicks, M., Foster, J.S., Ratikakis, P.P.: Contextual effects for version-consistent dynamic software updating and safe concurrent programming. SIGPLAN Not. 43, 37–49 (2008)
15. Aldrich, J., Kostadinov, V., Chambers, C.: Alias annotations for program understanding. In: OOPSLA, pp. 311–330. ACM, New York (2002)
16. Boyland, J., Noble, J., Retert, W.: Capabilities for Sharing: A Generalisation of Uniqueness and Read-Only. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 2–27. Springer, Heidelberg (2001)
17. Noble, J., Vitek, J., Potter, J.: Flexible Alias Protection. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, pp. 158–185. Springer, Heidelberg (1998)
18. Boyland, J.: Alias burying: unique variables without destructive reads. Softw. Pract. Exper. 31, 533–553 (2001)

P-Ray: A Software Suite for Multi-core Architecture Characterization*

Alexandre X. Duchateau, Albert Sidelnik,
María Jesús Garzarán, and David Padua

Department of Computer Science
University of Illinois at Urbana-Champaign
{axdn,asideln2,garzaran,padua}@uiuc.edu

Abstract. The increasing complexity of computer architectures has made the approach of automatically generating code that is optimized for the target machine a growing area of interest. Examples of such systems are library generators, such as ATLAS, SPIRAL, and FFTW. To generate optimized code without manual intervention, these systems need to know the values of certain hardware parameters, such as the cache size or the number of registers. Current software such as X-Ray or LMbench can automatically determine some of these parameters for single processor super-scalar machines but cannot determine multi-core specific characteristics.

In this paper, we present P-Ray, a software suite that characterizes hardware characteristics of multi-core architectures. Such characteristics include the number of cores that share the L2 cache, the different processors' interconnection topologies, and the bandwidth-to-memory. Our experiments show that, for several different architectures tested (desktop and server), P-Ray generates accurate results.

1 Introduction

With multi-core processors as the current dominant trend, and architectures more complex and less documented, finding hardware specifications is becoming increasingly difficult. Knowledge of hardware features can be useful in driving program optimization, such as in library generators. ATLAS [8], SPIRAL [5], and FFTW [2] are examples of known library generators. ATLAS generates linear algebra routines (BLAS) with a focus on matrix-matrix multiplication. SPIRAL and FFTW are similar to ATLAS, but generate signal processing libraries. Analytical models have been [9], and are being [1] developed for library generators that use hardware characteristics to reduce the search time. For example, ATLAS will use knowledge of the L2 cache size in order to determine optimal tile sizes for matrix-matrix multiplication.

* This material is based upon work supported by the National Science Foundation under Awards CCF 0702260 and CNS 0509432 and by DARPA under award W911NF0710416.

J.N. Amaral (Ed.): LCPC 2008, LNCS 5335, pp. 187–201, 2008.

Programs such as X-Ray [10], the suite reported in [6], and LMbench [4] try to address the problem of automatically finding machine characteristics, but focus on features of uniprocessor super-scalars. To our knowledge, there is no software designed to automatically measure parameters of multi-core machines.

In this paper, we extend the existing set of hardware characterizing software to multi-cores to find the number of caches shared by the cores, the processors' interconnection topologies, the effective bandwidth and block size used by the cache coherence mechanism.

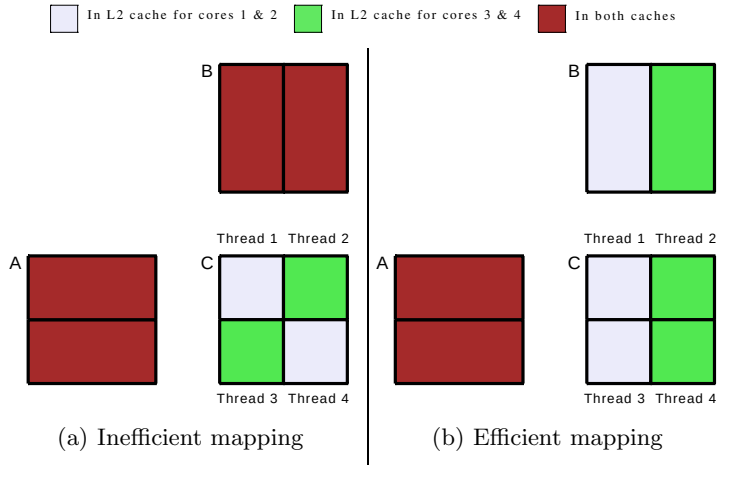
Our experimental results for three different platforms show that P-Ray generates accurate results.

The remainder of this paper is organized as follows. Section 2 provides motivating examples for our work. Section 3 presents the different hardware characteristics studied. Section 4 describes our implementation requirements and details. Section 5 summarizes the experimental environment and discusses results. Section 6 describes related work. Section 7 proposes future work. Finally in Section 8, we summarize our work and offer concluding remarks.

2 Motivation

Multi-Threaded Matrix-Matrix Multiplication

Library generators need a detailed knowledge of the architectural features of the machine to generate high-performance code. To show that this is the case, we ran an implementation of matrix-matrix multiplication ($C = A * B$) using POSIX threads on an Intel Core 2 Quad desktop that has four cores and two L2 caches, each cache shared by two cores. Figure 1 shows two different possible mappings for the matrices depending on thread affinity. For this experiment, matrix C is split into four sub-matrices, and each thread is assigned to one



(a) Inefficient mapping (b) Efficient mapping

Fig. 1. Data Locality depending on thread to core affinity

quadrant. Matrices are of size 800×800 each, so that they fit in memory but not in the L2 cache. With the mapping in Figure 1(a), both matrices A and B need to be loaded in both L2 caches. Using the mapping of Figure 1(b), matrix B can be split so that one half goes to one L2 cache and the other half goes to the second. Our experimental results show that an inefficient mapping can run up to 32% slower than an efficient one. To correctly map the threads to cores as in Figure 1(b), it was necessary to use P-Ray to obtain the ID of the cores that share the L2 cache. Thread affinity has been used in the past to pin a thread to a core in order to avoid its mapping to a different core (and subsequent cache trashing) after a context switch [7]. However, in current architectures where several cores could share a cache, thread affinity can be used to place in L2 the data shared by two threads. In most cases, this use of thread affinity can only be done if the programmer has the information provided by a tool such as P-Ray, by exhaustive search of all the possibilities, or if additional operating support is provided.

3 Targeted Characteristics

In this section, we present each part of the hardware we want to characterize and provide a high level description of the programs we propose to do so. Implementation specifics and detailed interpretation of the produced results will be discussed in Sections 4 and 5.

3.1 Cache Coherence Protocol Block Size

Knowing the block size used by the coherence protocol can aid the programmer in reducing false sharing misses. Other solutions already exist to measure a cache line size, but are slower than the one we propose. By exploiting false sharing our solution infers the block size in a fast and simple way.

Algorithm 1. Calculate block size

```
measure-size(core1,core2) {
char data[MAXLSIZES]
i ← 1
while i ≤ MAXLSIZE do
  Start timing
  Spawn thread-work(core1,0)
  Spawn thread-work(core2,i)
  Wait for threads to complete
  Stop timing
  Print i, timing
  i ← 2 * i
end while
}
```

```
thread-work(core,index) {
Set-thread-affinity(core)
i ← 0
while i ≤ SAMPLES do
  data[index] ← data[index] + 1
  i ← i + 1
end while
}
```

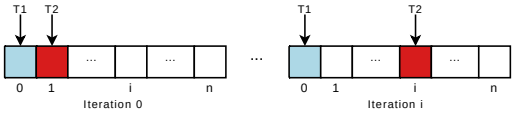


Fig. 2. Coherence Block Size Benchmark

Figure 2 illustrates Algorithm 1, that is used to compute the block size.

Two threads are spawned to work on a shared array of characters[1]. Both threads modify the shared data in order to induce coherence traffic. The data is also read to ensure it resides in L1: some architectures implement write-through write-no-allocate caches[2].

Thread one will always access the first element of the array. Thread two starts accessing the second element of the array; however, with each iteration, it accesses an element which is further apart from the first one.

At first, both threads will access the same cache line and have poor performance due to false sharing; as the spacing between accesses increases, the performance stays poor until both accessed values are on two separate cache lines. At this point, execution time decreases drastically and we automatically infer the coherence block size.

This algorithm can tell us the block size of different levels of cache. When the threads are mapped to cores that share a L2, this algorithm measures the block size of L1. However, when threads are mapped to cores that do not share a L2, this algorithm measures the block size of L2. When we do not have information about the mapping of a core to the caches, the second thread can be mapped to different cores in the system. By comparing the execution times of the different mappings, P-Ray can determine whether the block size corresponds to the L1 or L2 cache. When there is no coherency between the caches, this mechanism cannot determine the block size.

3.2 Cache Mapping

With this program we find the number of caches at a given level on the system and cores that share them.

For this to work, knowledge of the cache size is required for the level we are interested in. For completeness, P-Ray includes a program to approximate it; however cache size can also be measured with other programs [4,10]. Algorithm 2 calculates the number of caches. Each thread accesses an array approximately sized to L2 in order to cause misses between cores that share the same cache. This array is initialized as described in Algorithm 5 below. The first step of the algorithm is to measure the time it took for one thread to read the elements of this array when running in isolation. This time will be used as a reference to interpret the results.

[1] In most architectures, a character is a basic type of size 1 byte.

[2] Sun Niagara T1:
`http://opensparc-t1.sunsource.net/specs/OpenSPARCT1_Micro_Arch.pdf`

Algorithm 2. Calculate cache mapping

```
cache-mapping(core_1, core_2) {
i ← 1
while i ≤ SAMPLES do
  Spawn thread-work(core_1,1)
  Spawn thread-work(core_2,2)
  Wait on thread barrier
  Wait for threads to complete
  i ← i + 1
end while
}

thread-work(core, id) {
Set-thread-affinity (core)
Pointer p ← Initialize local data
Wait on thread barrier
Start timing
for i ← 0 to SIZE do
  p ← *p
end for
Stop timing
if id = 1 then
  Print core pair, timing
end if
}
```

Then, we run a similar test with two threads. Each thread sets its affinity to a different core, initializes its workset, and waits on a barrier for the other thread. Once both threads leave the barrier, we measure the time it takes for the threads to read their arrays while running simultaneously. If the measured execution time is higher than the reference time, we conclude that both threads ran on cores that share a cache, and that performance degraded due to the worksets of the two threads competing for cache space. If it is the same, we instead infer that both threads ran on separate caches.

This test is run for all pairs of cores on the system. After gathering all the results, P-Ray determines the number of caches on the system and the ID of the cores that map to them.

3.3 Processor Mapping

Here is a solution to determine the processors' interconnection topology.

Algorithm 3 uses two threads sharing a workset the size of the L1 cache, but running on separate cores. This algorithm functions by having one thread that

Algorithm 3. Calculate processor mapping

```
thread-work1(core_id_2) {
Set-thread-affinity (core_id_2)
p ← InitData(data,size,stride)
Wait on thread barrier
}

Require: Pointer p is global
thread-work2(core_id_1) {
Set-thread-affinity (core_id_1)
Wait on thread barrier
Start timing
for i ← 0 to L1SIZE do
  p ← *p
end for
Stop timing
Print (core_id_1,core_id_2), timing
}
```

reads and modifies its workset (i.e. brings it into L1) and measuring the time it takes for the second thread to read the data. By comparing the different access times of all possible pairs of cores, this program will determine the different relative distances between all cores.

3.4 Effective Bandwidth

This is the solution used to measure the available bandwidth for one core to memory by saturating it from one or several threads. In the following description the term "memory" will be used both for memory and caches unless otherwise specified.

Algorithm 4. Calculate bandwidth

```
Iteration1() {
Start timing
for i ← 0 to N_ITER do
  p ← *p
end for
Stop timing
Print timing
}
```

```
Iteration2() {
Start timing
for i ← 0 to N_ITER/2 do
  p1 ← *p1
  p2 ← *p2
end for
Stop timing
Print timing
}
```

We use an array that does pointer chaining with multiple entry points (this data structure and its initialization are described in Algorithm 5 and Figure 3 below). The offset between two entry points is here set to the size of a memory page. The stride between accesses is set to the smallest multiple of the page size that avoids overlap between chains.

To target a specific level in the memory hierarchy, we control the number of elements in the pointer chain before the loop back. We ensure that any reuse would happen after the data was displaced from levels closer to the core. Moreover, when measuring memory bandwidth, L2 is flushed after initialization.

Single-threaded bandwidth. The first step is to measure the bandwidth to memory for an isolated thread.

In the first iteration, the program traverses the array through a single entry pointer, as shown by Iteration1 in Algorithm 4. The code in Iteration1 serializes array accesses, as the access to the next element of the array cannot be issued until the pointer load returns. The second iteration of this program traverses the array through two entry pointers, as shown by Iteration2 in Algorithm 4. This loop has two independent accesses that can be sent simultaneously to the memory. However, accesses in an iteration depend on the accesses of the previous iteration for the same pointer chain due to the loop-carried dependences for all pointers. The program proceeds by increasing the number of independent requests. By measuring the execution time of these loops, P-Ray can determine

the number of requests that a core can have in-flight as well as its saturation point.

To calculate effective bandwidth, we use the following equation: Effective Bandwidth for program $= \frac{ClockFreq*ReadSize}{CyclesPerRead}$, where $CyclesPerRead$ is obtained by dividing the Execution Cycles at any of the saturation points by the number of iterations in our access loop. $ClockFreq$ is the clock rate for the given core. $ReadSize$ is the size of the cache line being read.

Multi-threaded bandwidth. We then look at the memory bandwidth when multiple cores are sending requests simultaneously. For that, we use Algorithm 4 in parallel over multiple threads. When considering a cache, we limit ourselves to running the program with threads on the cores that share that cache. When considering memory, we run the program with any number of cores in the system.

To better understand the impact of concurrent access on the bandwidth for the targeted memory, we test different numbers of threads: we test anywhere between two and the number of cores sharing the targeted memory.

4 Implementation

4.1 Requirements

Our software has two major requirements: i) a high resolution wall timer (e.g. on Intel machines we use the RDTSC instruction to get timing in clock cycles [3]) and ii) library and operating system support to set thread to core affinity.

4.2 Implementation Details

Pointer chaining. The data structure used by most of our solutions is an array of pointers where each element of the array contains the address to the next element to access when traversing the structure. A similar data structure has been used by X-ray [10] and LMbench [4], but we initialize it using our own techniques.

A picture of the data structure is shown in Figure 3, and the algorithm for its initialization is shown in Algorithm 5. The initialization algorithm takes four arguments: i) *data* is a pointer to the allocated memory, ii) *size* is the size in memory of the data structure, iii) *stride* is the distance between two consecutive

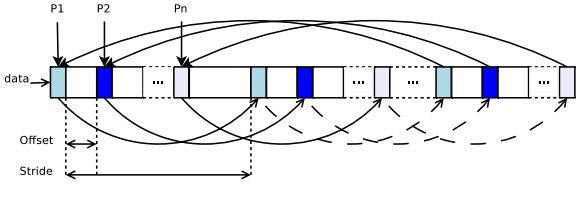


Fig. 3. Pointer chaining: General case

accesses, iv) $offset$ is the distance between two entry points, and v) *entries* is the number of entry pointers.

Our initialization routine uses a stride larger than page size to circumvent the hardware prefetcher and offset larger than the cache line size to prevent consecutive entry pointers from sharing a cache line. Since some experiments need to run for a large number of iterations, we limit the size of the array by having the last element of the chain point back to the first element (bottom lines of Algorithm 5).

Algorithm 5. Pointer Chaining

```
Init-data(data,size,stride,offset,entries)
i ← 0
while i < entries do
  uoffset ← i * offset
  Init-entry(data,size,stride,uoffset)
  i ← i + 1
end while
```

```
Init-entry(data,size,stride,offset)
i ← offset
while i ≤ size − stride do
  data[i] ←&data[i + stride]
  i ← i + stride
end while
data[i] ← &data[offset]
```

This structure has many advantages: i) it minimizes overhead, as no address has to be computed, ii) it allows for easy ways to experiment with different access patterns by tuning the initialization parameters, and iii) it prevents compiler optimizations that could interfere with performance measurements.

Loop Overhead. The loop overhead should not be considered in the timing. Thus, in order to minimize the control overhead, the main data access loops are unrolled by a factor of 512. This value was chosen because it reduces loop overhead without adding substantial instruction cache pressure. Additionally, we time an empty loop to remove the control overhead from our timing. This way, the final time reflects the actual access times.

Code Reordering. In order to prevent the compiler from performing any reordering of instructions within the timed kernel, *volatile* data modifiers are used. We were careful to not excessively mark every variable *volatile* when used outside of timed kernels, as this can hurt performance substantially.

System Noise. To deal with the problem of system noise from the operating system and other user applications, we take numerous timing samples and use the minimum timed value as our result. This compensates for other programs and daemons running on the system.

5 Evaluation

5.1 Experimental Environment

We tested on three different architectures described in Table 1.

Table 1. Architectures tested

	X86-64 Intel Xeon Harpertown	X86-64 Intel Core 2 Quad Kentsfield	Sun UltraSPARC T1 Niagara
Num cores	8[a]	4	8 (32 threads)
Clock Rate (GHz)	2.0	2.4	1.2
L2 Cache size (MB)	6	4	3
OS (Kernel)	Fedora 8 (2.6.24)	Fedora 8 (2.6.24)	Solaris 10
Compiler	GCC 4.1.2	GCC 4.1.2	GCC 3.4.3

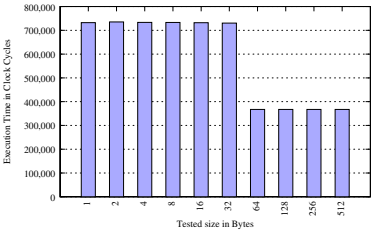
[a] Composed of two four-core chips

5.2 Experimental Results

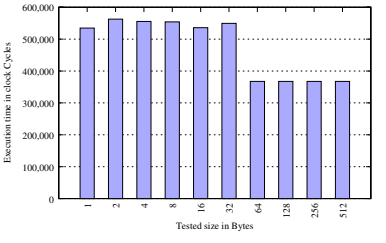
Coherence Block Size. Figure 4 illustrates the results for Algorithm 1. On the Intel machines (Figure 4(a) and 4(b)), the threads were mapped to cores not sharing the L2 cache. The results show that, on the Intel machines, there is a notable time decrease as soon as the two accesses are 64-bytes apart. From this, we conclude that the coherence protocol on these machines uses 64-byte blocks.

On the Sun UltraSPARC T1 (Figure 4(c)), we observe that the largest performance difference occurs at the 16-byte block size, which corresponds to the size of the L1 data cache block.

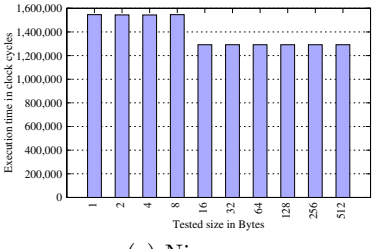
To show block sizes at the different cache levels and communication latencies, we evaluated different mappings of threads to cores. Figure 5 shows the results for the Intel Harpertown architecture, which has eight cores. We first mapped the threads to the two cores on the same dual-core. We then mapped



(a) Harpertown



(b) Kentsfield



(c) Niagara

Fig. 4. Coherence Block Size Results

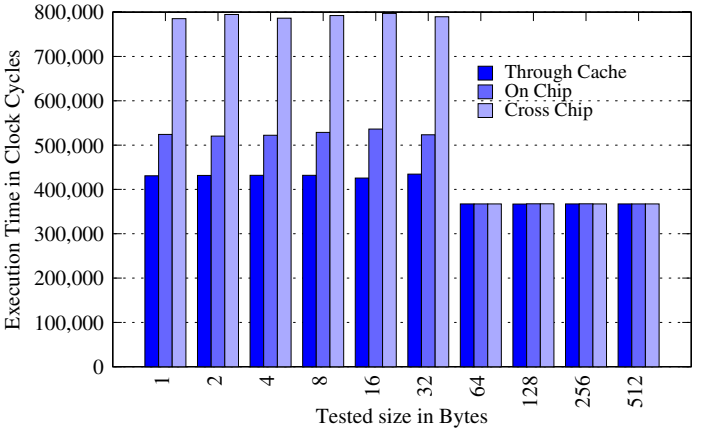
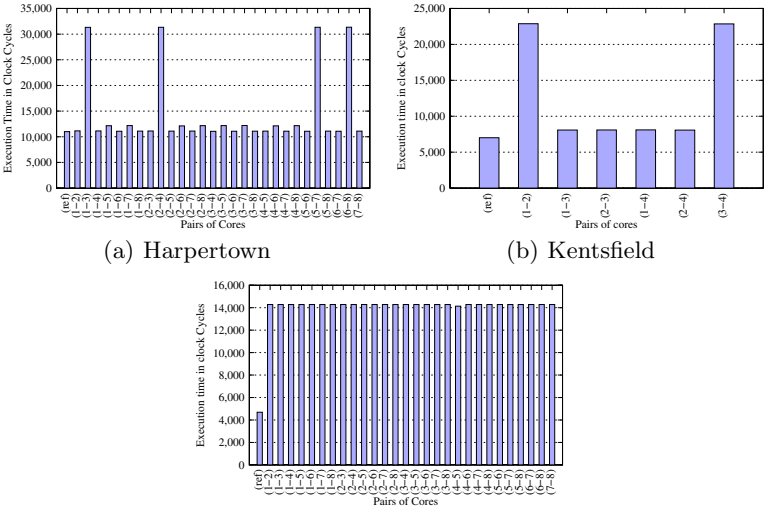


Fig. 5. Coherence block size and communication latency

the threads to cores on the same chip but not the same cache. Finally we mapped the threads to cores on different chips. We see that for this machine, while the block size is always 64-bytes, the different values of execution time show the different communication latencies among the different cores. There is a higher communication latency when the communicating cores are on different chips, and the communication cost is lower when the cores are closer together.

Cache Mapping. Figure 6 illustrates the results for Algorithm 2. The results clearly show which of the cores share a cache.



(a) Harpertown

(b) Kentsfield

(c) Niagara

Fig. 6. Cache Mapping

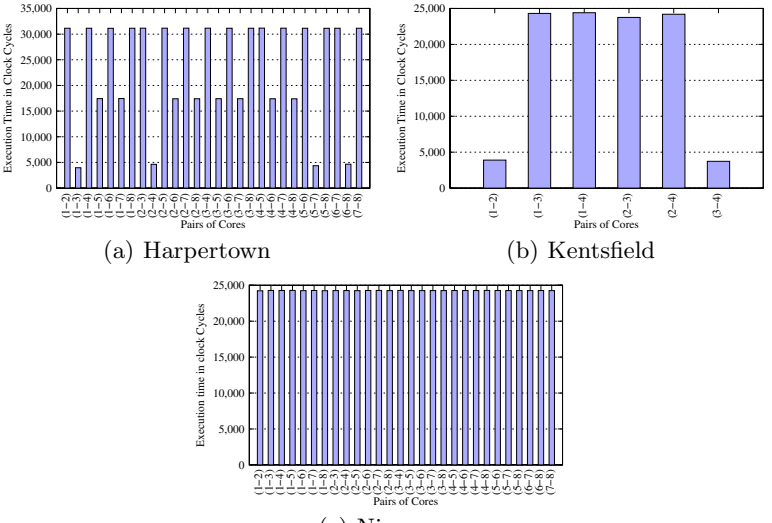
By looking at the pairs of cores with the highest access times, we see that for the Harpertown (Figure 6(a)), core pairs of ID $(1-3)$,$(2-4)$,$(5-7)$, and $(6-8)$ share a cache, and for the Kentsfield (Figure 6(b)), the core pairs ID $(1-2)$ and $(3-4)$ share a cache.

For the Sun UltraSPARC (Figure 6(c)), we see that all core pairs have the same performance. When comparing this performance with the single thread reference time, we realize that all core pairs perform poorly and, thus, we infer that all cores share the same cache.

Processor Mapping. Figure 7 illustrates the results for Algorithm 3. For the Harpertown (Figure 7(a)), we see three different distances. First, we have the pairs of cores that are the closest. These pairs correspond to those that share a cache in Figure 6(a): $(1-3)$,$(2-4)$,$(5-7)$, and $(6-8)$. Then we have two groups of four cores, where communicating between pairs in a group is faster than communicating between cores in different groups: $((1-3)(5-7))$ and $((2-4)(6-8))$. Those results confirm what is found on the design of the two architectures: the machine is composed of two four-core chips, with each four-core chip composed of two combined dual-cores. For the Kentsfield (Figure 7(b)), we confirm that pairs of cores that share a cache communicate faster.

For the Niagara (figure 7(c)), results show that all cores are equidistant, which confirms the results obtained for the cache mapping.

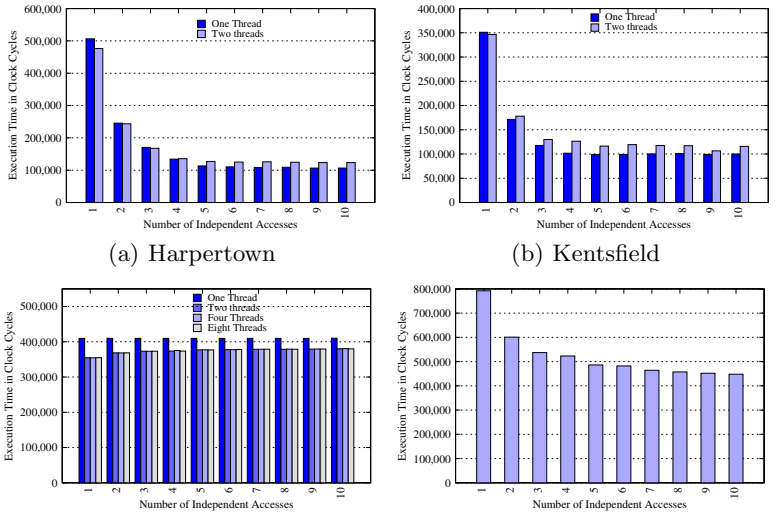
Effective L2 Bandwidth. Figure 8 illustrates the results for Algorithm 4. We show data for each platform when a thread is run in isolation and when two or more threads run concurrently. By looking at results for one thread, we observe that, for all Intel machines, the execution time decreases as the number



(a) Harpertown (b) Kentsfield

(c) Niagara

Fig. 7. Processor Mapping



(a) Harpertown (b) Kentsfield

(c) Niagara (d) Niagara with branch penalty

Fig. 8. Effective Bandwidth to L2 Results

Table 2. Effective Bandwidth to L2

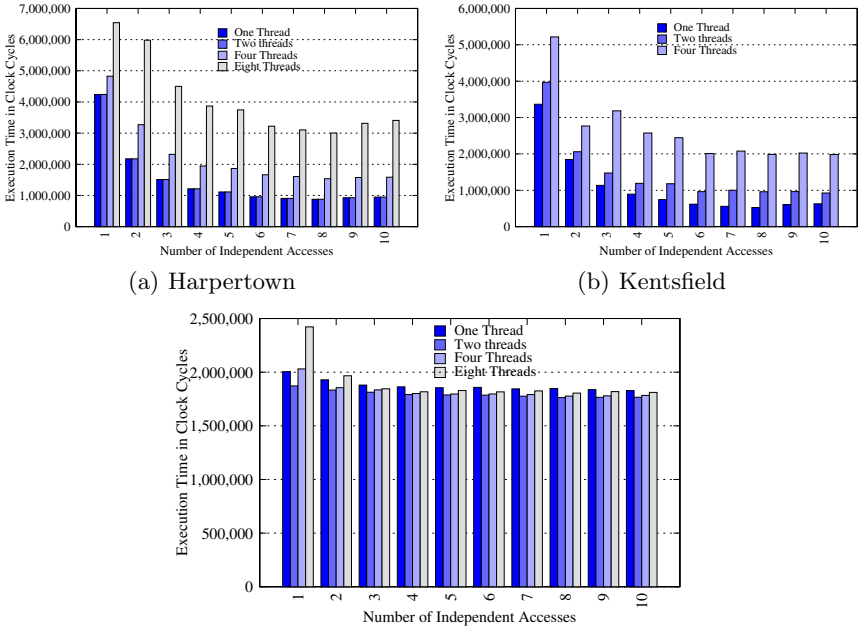
Machine	L1 Block	Cycles per Access (concurrent accesses)				Effective Bandwidth GB/s			
# threads		1	2	4	8	1	2	4	8
Harpertown	64B	6.18 (9)	7.54 (9)	-	-	19.29	15.81	-	-
Kentsfield	64B	6.05 (6)	6.50 (9)	-	-	23.65	22.01	-	-
Niagara	16B	25.00 (6)	21.66 (1)	21.64 (1)	21.66 (1)	0.71	0.83	0.83	0.83

of independent accesses that can be issued simultaneously increases. There is a saturation point where the execution time remains constant. The smallest number of independent accesses where the saturation point is reached tell us the number of requests that can be served in parallel. When two or more threads run concurrently, the bars have a similar trend but have a slightly higher execution time.

Figure 8(d) shows the execution time for the program running without removing the loop overhead. The improvement in execution time looks like it comes from more parallelism between memory requests. In fact, the Niagara does not have branch prediction; the reduction in execution time is only due to the decrease in number of iterations (i.e. number of branches per access).

Finally Table 2 shows the bandwidth computed using the formula shown in Section 3.4 for the different number of threads.

Effective Memory Bandwidth. Figure 9 illustrates the results for Algorithm 4. For the Intel processors (Figure 9(a) and 9(b)), as the number of cores accessing memory increases, we observe a substantial decrease in bandwidth.



(a) Harpertown (b) Kentsfield

(c) Niagara

Fig. 9. Effective Bandwidth to Memory Results

Table 3. Effective Bandwidth to Memory

Machine	L2 Block	Cycles per Access (concurrent accesses)				Effective Bandwidth GB/s			
# threads		1	2	4	8	1	2	4	8
Harpertown	64B	52.09 (8)	53.60 (8)	93.92 (9)	183.30 (9)	2.29	2.22	1.29	0.65
Kentsfield	64B	32.03 (8)	56.49 (10)	121.02 (10)	-	4.47	2.53	1.18	-
Niagara	64B	111.62 (10)	107.61 (8)	108.45 (8)	110.14 (8)	0.64	0.67	0.66	0.65

On the Niagara (Figure 9(c)), we observe similar results as for the L2 cache; the bandwidth available to the cores stays the same regardless of the number of concurrent requests. Finally, Table 3 shows the values for the effective bandwidth.

6 Related Work

As discussed in the introduction, there are other software suites such as LMBench [4], Saavedra [6], and X-Ray [10] that measure architectural characteristics and, while they focus on single core features, P-Ray focuses on multi-core specific features. Benchmark suites like SPEC OMP[3] are designed for shared memory

[3] Standard Performance Evaluation Corporation. `http://www.spec.org/omp`

multiprocessors. Such benchmarks only give a relative performance scale, but do not give any information about the characteristics of the targeted system.

7 Future Work

We are looking into additional characteristics to target whith this suite. While synchronization contention is mostly a library/OS issue, it could be a useful feature. It would be also be beneficial to have a similar characterizing software for heterogeneous systems such as the Cell or GPU-based architectures.

From feedback received, there is a need for these multi-core characteristics to be found online as well as offline. This would be beneficial for projects such as adaptable systems with dynamic hardware features[5]. If we are to do online processing for certain characteristics, execution time will have to be addressed to minimize the overhead of using our framework.

It would be interesting to test our software suite on additional hardware architectures, such as IBM Power6 and Intel Itanium 2.

8 Conclusion

We have developed a suite of conceptually simple solutions that focuses on multi-core characteristics. Our suite returns results that are in accordance with vendor specifications when available and coherent when they are not.

The main difference between P-Ray and existing software is that P-Ray offers a unique view of the system design, showing the position of the different cache levels and relative distances between (virtual) cores in the system. With this information at hand, a programmer has the ability to use more efficient hardware-aware optimizations in their applications. In addition, we provide a faster means to calculate a cache block size by exploiting false sharing. Finally, the execution and analysis framework is extensible, allowing for the addition of further hardware characterization.

Acknowledgments. We would like to thank Dario Suarez Garcia (University of Zaragoza, Spain), Basilio Fraguela (University of Coruña, Spain), James Brodman (University of Illinois at Urbana Champaign), and Megan Osfar for their advice and comments.

References

1. Personal communication with Basilio B. Fraguela, Universidade da Coruña
2. Frigo, M.: A Fast Fourier Transform Compiler. In: PLDI 1999 — Conference on Programming Language Design and Implementation (1999)
3. V.U. Instruction. Intel architecture software developer's manual
4. McVoy, L., Staelin, C.: lmbench: portable tools for performance analysis. In: ATEC 1996: Proceedings of the annual conference on USENIX Annual Technical Conference, San Diego, CA, pp. 23–23. USENIX Association (1996)

5. Püschel, M., Moura, J.M.F., Johnson, J., Padua, D., Veloso, M., Singer, B., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R.W., Rizzolo, N.: SPIRAL: Code Generation for DSP Transforms. Proceedings of the IEEE, special issue on Program Generation, Optimization, and Adaptation 93(2), 232 (2005)
6. Saavedra, R.H., Smith, A.J.: Analysis of benchmark characteristics and benchmark performance prediction. ACM Trans. Comput. Syst. 14(4), 344–384 (1996)
7. Torrellas, J., Tucker, A., Gupta, A.: Evaluating the performance of cache-affinity scheduling in shared-memory multiprocessors. J. Parallel Distrib. Comput. 24(2), 139–151 (1995)
8. Whaley, R.C., Petitet, A., Dongarra, J.: Automated Empirical Optimizations of Sofware and the ATLAS Project. Parallel Computing 27(1-2), 3–35 (2001)
9. Yotov, K., Li, X., Ren, G., Garzaran, M.J.S., Padua, D., Pingali, K., Stodghill, P.: Is search really necessary to generate high-performance blas? Proceedings of the IEEE 93(2), 358–386 (2005)
10. Yotov, K., Pingali, K., Stodghill, P.: X-ray: A tool for automatic measurement of hardware parameters. In: QEST 2005: Proceedings of the Second International Conference on the Quantitative Evaluation of Systems on The Quantitative Evaluation of Systems, p. 168. IEEE Computer Society Press, Los Alamitos (2005)

Scalable Implementation of Efficient Locality Approximation

Xipeng Shen[1] and Jonathan Shaw[2]

[1] Computer Science Department
The College of William and Mary, Williamsburg, VA
[2] Shaw Technologies, Inc.
Tualatin, OR

Abstract. As memory hierarchy becomes deeper and shared by more processors, locality increasingly determines system performance. As a rigorous and precise locality model, reuse distance has been used in program optimizations, performance prediction, memory disambiguation, and locality phase prediction. However, the high cost of measurement has been severely impeding its uses in scenarios requiring high efficiency, such as product compilers, performance debugging, run-time optimizations.

We recently discovered the statistical connection between time and reuse distance, which led to an efficient way to approximate reuse distance using time. However, not exposed are some algorithmic and implementation techniques that are vital for the efficiency and scalability of the approximation model. This paper presents these techniques. It describes an algorithm that approximates reuse distance on arbitrary scales; it explains a portable scheme that employs memory controller to accelerate the measure of time distance; it uncovers the algorithm and proof of a trace generator that can facilitate various locality studies.

1 Introduction

In modern computers, memory hierarchy is becoming deeper and shared by more processors; system performance is increasingly determined by program data locality. Reuse distance, also called LRU stack distance, is a widely used model for locality analysis. Compared to other locality metrics such as cache miss rate, reuse distance is hardware independent, cross-input predictable, and more precise in characterization [7, 12, 20]. It has been used in system performance analysis [6,9,10], performance prediction [12,19], program analysis and optimizations [2, 8, 20].

On the other hand, reuse distance is also one of the most expensive locality models to build. Despite decades of enhancement (e.g., [7,13]), the measurement still slows down a program's execution by hundreds of times [16]. The high cost impedes the practical uses of reuse distance: It would make offline performance debugging and locality analysis painfully slow or even infeasible for long-running applications, and prevent the uses in runtime locality optimizations. The high cost is inherent in the definition of **reuse distance**—the number of *distinct* data accessed between this and the previous access to the same data item [7]. The requirement of being "distinct" implies that the measurement has to recognize and filter out all the repetitive accesses in an interval, which is often costly.

J.N. Amaral (Ed.): LCPC 2008, LNCS 5335, pp. 202–216, 2008.

Recently Shen et al. discovered the strong statistical connection between reuse distance and **time distance**—the number of data accessed between this and the previous access to the same data item (e.g., the last "a" in "a b b c a" has time distance of 4). This discovery gives rise to an algorithm that approximates reuse distance histograms from time distance histograms. Given that time distance is much cheaper to measure, the algorithm speeds up reuse distance measurement significantly [16].

However, it remains un-exposed how to implement the algorithm efficiently. In particular, this paper focuses on the problems on three folds.

The first is a scale problem. The algorithm described previously requires the finest granularity of distance histograms [16]. Every bar in a histogram must have width of 1—that is, all references in a bar have to have the same reuse (or time) distance. Such a scale requires the recording of and computation on millions of distance bars for a typical execution, making the algorithm not applicable to long running programs. However, extending the algorithm to support histograms of arbitrary scales is challenging. A basic extension has a too high time complexity (shown in Section 2.2). In this paper (Section 2.3), we describe some algorithmic changes to enable the removal of redundant computations, lowering the time complexity by orders of magnitude.

The second problem addressed in this paper is the overhead in time distance measurement. Although time distance is less costly to measure than reuse distance, a straightforward implementation still slows down a program's execution significantly, forming the bottleneck in our extended scalable algorithm. This paper (Section 3) presents an optimization that reduces the measurement overhead by more than a factor of 3 with the help of memory management unit (MMU). The optimization differs from previous MMU-based techniques in that it avoids the direct access to system registers and therefore is more portable.

Finally, this paper (Section 4) presents a trace generator, which produces data reference traces that satisfy the given requirement of reuse distance histograms. We are not aware of any prior systems that offer such a function. This generator not only facilitates the comprehensive evaluation of the reuse-distance approximation algorithm, but also can serve as a tool for other locality studies.

2 Algorithm Design for Scalability

This section reviews the statistical connection between time distance and reuse distance, identifies the scalability bottlenecks in a basic algorithm for reuse distance approximation, and then presents our changes to the algorithm to reduce the cost by orders of magnitude, making the algorithm scalable to long-running programs.

2.1 Review of the Connection between Time Distance and Reuse Distance

The key connection that bridges reuse distance and time distance is the following equation [16]:

$$p(\Delta) = \sum_{\tau=1}^{\Delta} \sum_{\delta=\tau+1}^{T} \frac{1}{N-1} p_{\tau}(\delta), \tag{1}$$

where, $p(\Delta)$ is the probability for a randomly chosen data item (e.g., a variable) of a program to be referenced in an interval of length Δ, and $p_{_T}(\delta)$ is the probability for a randomly chosen data reference to have time distance of δ. This connection suggests that if we know all $p_{_T}(\delta)$s—that is, the time distance distribution of an execution—we will be able to compute all $p(\Delta)$s.

With $p(\Delta)$, the reuse distance can be computed easily. For a reuse interval of length Δ, the probability for its reuse distance to be k in an execution is

$$p(k, \Delta) = \binom{N}{k} p(\Delta)^k (1 - p(\Delta))^{N-k}, \tag{2}$$

where, N is the total number of distinct data items in a program. The intuition behind this equation is that $p(k, \Delta)$ is the probability for k distinct data items to appear in a Δ-long interval. This probability is like the probability to see k heads when N coins are tossed, with each coin's probability of showing heads to be $p(\Delta)$[1]. This probability obeys a binomial distribution.

It is easy to see that, with $p(k, \Delta)$, the probability for a data reference to have reuse distance of k is

$$p_{_R}(k) = \sum_{\Delta} p(k, \Delta) \cdot p_{_T}(\Delta). \tag{3}$$

2.2 Basic Algorithm for Approximating Reuse Distance Histograms

A program execution often conducts a large number of data references. A typical way to concisely characterize reference locality is to use reuse distance histograms instead of individual reuse distances [2, 7, 12, 20]. Figure 1 illustrates a reuse distance histogram. A bar in the graph shows the fraction of the memory references whose reuse distances are in a certain range. For the same reason, individual time distances are usually not affordable; time distance histograms are used.

Therefore, extending the equations in Section 2.1 to handle histograms is vital for practical uses. A straightforward way to do the extension is to consider that all the references in a bar have the same $p(\Delta)$, denoted by $P(b_i)$ ($i = 1, 2, \cdots, L_T$ for a time distance histogram that has L_T bars). So, the probability for k variables to be accessed in a reuse interval contained in bar i is

$$P(k, b_i) = \binom{N}{k} P(b_i)^k (1 - P(b_i))^{N-k}.$$

And the probability for a random data reference in an execution to have reuse distance of k is

$$P_R(k) = \sum_{i} P(k, b_i) \cdot P_T(b_i).$$

Therefore, the key step in the extension is to compute $P(b_i)$ from a time distance histogram. If we assume that reuse distances are in a uniform distribution inside a bar,

[1] This analogy assumes that two data items are independent in terms of the probability for them to be accessed in a given interval; this assumption has shown little influence to reuse-distance approximation [16].

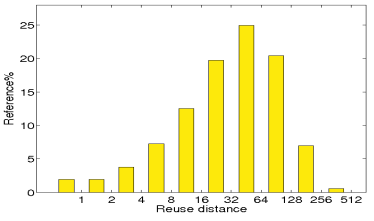


Fig. 1. A log-scale reuse distance histogram

$P(b_i)$ can be approximated by $p(\frac{\overleftarrow{b_i}+\overrightarrow{b_i}}{2})$ (where $\overleftarrow{b_i}$ and $\overrightarrow{b_i}$ are the lower and upper bounds of all the reuse distances in the ith bar)[2]. Under this assumption, $P(b_i)$ can be computed by Equation 1 as follows:

$$P(b_i) = \sum_{\tau=1}^{\frac{\overleftarrow{b_i}+\overrightarrow{b_i}}{2}} \sum_{\delta=\tau+1}^{T} \frac{1}{N-1} p_T(\delta).$$

Clearly, the sum of $p_T(\delta)$ can be converted to a sum of $P_T(b_i)$—the time distance histogram.

This basic algorithm has high cost, mainly due to the calculation of $P(b_i)$ and $P(k, b_i)$. The time complexity to compute all of $P(b_i)$s is $O(L_T^3)$ (recall that L_T is the number of bars in a time distance histogram), and the complexity for all $P(k, b_i)$s is a factorial of N. We have to lower the complexity before the algorithm can be applied to real programs.

2.3 Scalable Algorithm

The basic algorithm contains some repetitive computations, especially in the computation of $P(b_i)$. We make two changes to the algorithm to remove the redundant computations and lower the complexity.

The first change is based on a well known fact that a binomial distribution can be approximated by a Normal distribution. For further speedup, we use an offline generated table to store the standard Normal distribution. The computation of $P(k, b_i)$ is reduced to a table-lookup operation with $O(1)$ complexity.

The second change is to decompose the computation of $P(b_i)$ into 3 sub-equations with repetitive computations removed. The 3 sub-equations are as follows:

$$P(b_i) = P_2(b_i) \Big/ 2 + \sum_{j=0}^{i-1} P_2(b_j) \tag{4}$$

[2] This assumption is also used in the scalable algorithm; its influence to accuracy is evaluated by the experiments in Section 5.

$$P_2(b_i) = \left[\sum_{j=i+1}^{L_T} P_1(b_j) \frac{\overrightarrow{b_i} - \overleftarrow{b_i}}{\overrightarrow{b_j} - \overleftarrow{b_j}} \sum_{\tau=\overleftarrow{b_j}}^{\overrightarrow{b_j}-1} \frac{1}{\tau - 1} \right] + P_1(b_i) \frac{1}{\overrightarrow{b_i} - \overleftarrow{b_i}} \sum_{\tau=\overleftarrow{b_i}+1}^{\overrightarrow{b_i}-1} \frac{\tau - \overleftarrow{b_i}}{\tau - 1} \quad (5)$$

$$P_1(b_i) = \frac{\overleftarrow{b_i} + \overrightarrow{b_i} - 3}{2(N-1)} P_T(b_i). \quad (6)$$

The detailed derivation of these sub-equations is too complex to be included in this paper. We give a brief explanation and refer the reader to our technical report [15]. Recall that $P(b_i)$ can be approximated by $p(\frac{\overleftarrow{b_i}+\overrightarrow{b_i}}{2})$—that is, the probability for a variable to be accessed in an interval whose length is $\frac{\overleftarrow{b_i}+\overrightarrow{b_i}}{2}$. This probability can be viewed as the probability for the variable's last access prior to a random time point t to be after $t - \frac{\overleftarrow{b_i}+\overrightarrow{b_i}}{2}$, i.e., to be in the dark segment illustrated in Figure 2. This dark segment can be regarded as a sequence of sub-segments[3], $[t - \frac{\overleftarrow{b_i}+\overrightarrow{b_i}}{2}, t - \overleftarrow{b_i})$, $[t - \overleftarrow{b_i}, t - \overleftarrow{b_{i-1}})$, $\cdots$, $[t - \overleftarrow{b_1}, t)$. We use $P_2(b_j)$ to denote the probability that the variable's last access prior to t occurs in a sub-segment, $[t - \overleftarrow{b_{j-1}}, t - \overleftarrow{b_j})$, which leads to Equation 4.

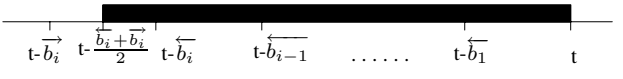


Fig. 2. Illustration of $P(b_i)$

The probability $P_1(b_i)$ in Equations 5 and 6 is the probability that for a randomly chosen variable v, a random time-point t is in one of v's reuse intervals whose time distance is in range $[\overleftarrow{b_i}, \overrightarrow{b_i})$. Equations 5 and 6 come from a complex mathematical deduction, which examines the different sections of a reuse interval and the different relations between reuse distance and time distance histograms [15].

This extended algorithm enables two simplifications. First, through the following commonly used mathematical approximation (m_1, m_2 and i are positive integers):

$$\sum_{i=m_1}^{m_2} \frac{1}{i} \simeq ln \frac{m_2 + 0.5}{m_1 - 0.5},$$

Equation 5 can be simplified to the following form:

$$P_2(b_i) \simeq \left[\sum_{j=i+1}^{L_T} P_1(b_j) \frac{\overrightarrow{b_i} - \overleftarrow{b_i}}{\overrightarrow{b_j} - \overleftarrow{b_j}} ln \frac{\overrightarrow{b_j} - 1.5}{\overleftarrow{b_j} - 1.5} \right] + \frac{P_1(b_i)}{\overrightarrow{b_i} - \overleftarrow{b_i}} \left[\overrightarrow{b_i} - \overleftarrow{b_i} - 2 - (\overleftarrow{b_i} - 1) ln \frac{\overrightarrow{b_i} - 1.5}{\overleftarrow{b_i} - 0.5} \right]$$

[3] This work uses logical time: the number of memory references since program starts.

Second, Equation 4 reveals the relation between $P(b_i)$ and $P(b_{i-1})$ as follows:

$$P(b_i) = P(b_{i-1}) + \frac{P_2(b_{i-1}) + P_2(b_i)}{2}.$$

So, given all $P_2(b_i)$s, the time complexity of obtaining all $P(b_i)$s is $O(L_T)$. Apparently, the time complexity of computing all $P_2(b_i)$s is $O({L_T}^2)$. Therefore, the complexity for computing all of $P(b_i)$ is $O({L_T}^2)$. With the simplification to $P(k, b_i)$ by the first change, the time complexity of this extended algorithm is $O({L_T}^2)$, orders of magnitude lower than that of the basic algorithm.

3 Measurement Acceleration for Efficiency

To efficiently implement the approximation model, we use a portable scheme to take advantage of MMU. This scheme accelerates time distance measurement—the bottleneck in the scalable algorithm—by more than a factor of 3.

The measurement of time distance requires detailed recording of memory references. We use a binary instrumentation tool, PIN [11], to insert a function call for recording the memory address after each memory reference. Algorithm 1 shows the basic implementation of the recording procedure. It first stores the accessed memory address into a buffer and then checks if the buffer is full. When it is full, a procedure, *ProcessBuffer()*, is invoked to calculate the time distances in the buffer and record them into a histogram.

Algorithm 1. Basic measurement of time distance

```
Procedure RecordMemAcc (address)
  buffer [index++] ⇐ address;
  if index == BUFFERSIZE then
    ProcessBuffer ( ); // Calculate and record reuse distances and reset index to 0
  end if
end
```

The basic implementation slows down a program's execution significantly. To reduce the overhead, we optimize the procedure *RecordMemAcc( )* by removing the branch and function call in it with the help of MMU. When the instrumented program starts, through MMU, the last page of the array *buffer* is set to be non-writable. When the program tries to write an address to that page, a page access violation signal is triggered. In the customized signal handler, the procedure *ProcessBuffer()* is invoked and the buffer index is reset. This optimization removes the necessity for buffer boundary check, reducing procedure *RecordMemAcc( )* to just one statement as shown at the top of Algorithm 2.

Zhao et al. used memory management unit to remove similar branches in detailed execution profile [18]. Our scheme is different in how to resume the execution in the signal handler. The previous work resets the register that contains the buffer index to zero so that after returning from the signal handler, the program can write to the first

Algorithm 2. Portable MMU-based optimization for time distance measurement 1

```
Procedure RecordMemAcc (address)
    buffer [index++] ⇐ address;
end

// Procedure to handle memory access fault
// Initially, the last page of buffer is locked
Procedure sig_SEGV_H (siginfo)
    faultAddr ⇐ GetFaultAddress (siginfo);
    if IsInLastPage(faultAddr, buffer) then
        ProcessBuffer ( ); // Calculate and record reuse distances
        OpenLastPage (buffer); // Open the access permission of the last page
        Close2ndToLastPage (buffer); // Close the access permission of the second to last
        page
    else if IsIn2ndToLastPage(faultAddr, buffer) then
        ProcessBuffer ( );
        Open2ndToLastPage (buffer);
        CloseLastPage (buffer);
    else
        . . .
    end if
    index = -1; // RecordMemAcc() will make it zero right after the current instruction
    finishes
end
```

element of the buffer. The shortcoming of the scheme is the portability problem: The registers that contain the buffer index may differ on different platforms.

We address that problem using a portable scheme, shown by procedure *sig_SEGV_H* in Algorithm 2. Initially we close the permission to access the last page of *buffer*. The signal handler opens the permission of the *last* page and closes the *second to last* page. (The variable for the buffer index is reset.) The execution continues until trying to access the second to last page of *buffer*. The signal handler then opens the second to last page and closes the last page again. By using the last two pages alternatively to signal the end of *buffer*, this scheme removes the necessity for modifying specific register values.

The optimization accelerates time distance measurement by 3.3 times as shown in Section 5. The significant benefits come from three sources. First, it directly reduces the number of instructions and branch miss predictions in the recording procedure. Second, it reduces the overhead of the dynamic instrumentor. The instrumentor, PIN, uses a virtual machine equipped with a just-in-time compiler for dynamic instrumentation. Fewer instructions in the analysis code leads to fewer computations in the virtual machine. Third, the optimization enables the inlining of the recording procedure. PIN is strict in inlining since the instrumented procedures usually have a large number of call sites. The control flow in the basic implementation prevents PIN from inlining the procedure, causing high calling overhead. Function calls in PIN are especially expensive: At an function call, PIN calls a bridge routine that saves all caller-saved registers, sets up analysis routine arguments, and finally calls the target function [11].

This scheme provides an architecture-independent way to employ MMU for code optimization. It suggests the potential of serving as a general optimization technique for a compiler to apply.

4 Trace Generator for Evaluation

Although this work is on measuring locality from data accesses, this section discusses the reverse problem—how to generate data traces from locality. The initial motivation is for the evaluation of our locality approximation model: Although the model demonstrates high accuracy for SPEC CPU2000 benchmarks, the reuse distance histograms of those programs fall into several categories, covering only a small portion of the entire histogram space. The capability to generate data access traces that satisfy given locality requirements is desirable: We can freely use various traces to evaluate the locality model comprehensively. Although this capability provides conveniences for many locality studies, we are not aware of any prior explorations on it.

This work constructs a stochastic trace generator. Its inputs include a reuse distance histogram P, and the length T and the number of distinct variables N of the trace to be generated; its output is a data reference sequence satisfying the input requirements.

4.1 Algorithm

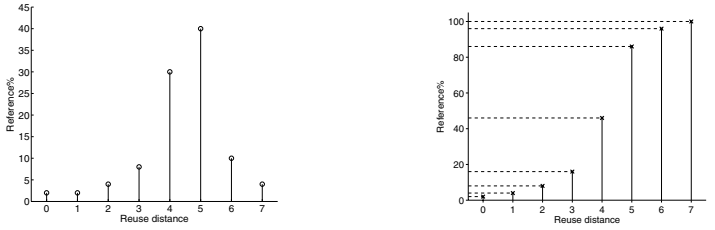
Algorithm 3 contains the pseudo code of the trace generator. For the purpose of clarity, the following explanation assumes that the bars in the histograms are of unit width.

Algorithm 3. Algorithm to generate a data trace from a reuse distance histogram

```
Procedure GenTrace (RDH[], N, T, trace[])
  // RDH[]: the given reuse distance histogram;
  // N: the number of distinct data;
  // T: the length of the trace to be generated;
  // trace[]: the array to contain the generated trace;
  BuildCH (RDH, CH); // build cumulative histogram from RDH

  // fill the first N positions
  for i=0 to N-1 do
    trace[i] = var[i];
  end for

  // fill the rest
  for i=N to T do
    a = random();
    s = findSegment (a, CH); // find the segment in CH that contains a
    r = RDH[s];
    trace[i] = findData (trace, r); // find the data having reuse distance of r at the end
    of current trace
  end for
end
```



(a) Reuse distance histogram (with bar width as 1)

(b) Cumulative distribution of the histogram in (a)

Fig. 3. A histogram and the accumulative distribution

The first step is to construct the cumulative distribution of the given reuse distance histogram. The probability for a data access to have reuse distance no longer than i is calculated as $C_i = \sum_{j=0}^{i} P_j$, where, P_j is the probability for a data access to have reuse distance of j—that is, the Y-axis value at reuse distance j in the reuse distance histogram.

The second step is to fill a variable into each position in a T-element trace. The first N positions are simply filled by all the variables. (The order does not matter.) To explain the process of filling the remaining positions, we use the example shown in Figure 3. In the example, there are 8 possible reuse distances 0 to 7. Their cumulative probabilities C_i (i=0,1,. . .,7) separate the range [0, 100%] into 8 segments, $[0, C_0]$, $(C_0, C_1)]$, . . ., $(C_6, C_7]$ (apparently $C_7 = 100\%$), shown on the Y-axis of the cumulative histogram.

Every time, a random number α is generated whose value is between 0 and 1. If α falls into segment $(C_{i-1}, C_i]$, the trace generator uses i as the desired reuse distance of the current position and identifies the corresponding variable and put it into the current position. For example, if the current data trace is ". . . a b c b c" and α is 0.05, the segment that α falls into is $(C_1, C_2]$. Therefore, the reuse distance of the current position should be 2. Because "a" satisfies the distance requirement, an instance of "a" is put into the current position.

4.2 Proof of Correctness

This section outlines the proof that the trace generated by Algorithm 3 satisfies the input requirements.

Theorem 1. *In a trace generated by Algorithm 3, the statistical expectation of the number of accesses that have reuse distance of i is $(T * P_i)$ (i=0,1,. . .,N-1)—that is, the statistical expectation of the reuse distance histogram of the generated trace is equal to the given histogram. (T: the number of total accesses; P_i: the value of the ith bar in the given reuse distance histogram.)*

To see the correctness of the theorem, notice that the probability for α to fall into the ith segment in the cumulative reuse distance histogram is equal to P_i. This is because

α is uniformly distributed between 0 and 1, and the length of the ith segment is P_i. So, if the trace generator follows Algorithm 3, the probability for a generated data to have reuse distance of i equals P_i; the conclusion thus follows. (The proof assumes $T >> N$, which holds for most program executions.)

Finding the data with reuse distance of i takes $O(logN)$ time when we organize the last access to each data in a binary tree. The time complexity of the whole trace generation is therefore $O(TlogN)$.

5 Evaluation

Our evaluation platform is an Intel Xeon 2GHz processor running Fedora Core 3 Linux. We use PIN 3.4 for instrumentation and GCC 3.4.4 (with "-O3") as the compiler. We employ Performance Application Programming Interface (PAPI) [3] to read hardware performance counters.

This section first presents the benefits from the portable MMU scheme in accelerating the measurement of time distance, then reports the efficiency and accuracy of the scalable algorithm on real programs, and finally shows the accuracy of the trace generator, along with its uses in the evaluation of the reuse distance approximation algorithm.

5.1 Time Distance Measurement

Table 1 shows the effect of the portable MMU-based optimization on time distance measurement. The 9 benchmarks are randomly chosen from SPEC CPU2000 suite. We use their *train* runs for measurement. The second column in the table contains the reduction of the total instructions executed by the instrumented code. The optimization reduces 60–69% instructions with an average of 66.7%. The Third column shows that the optimization reduces branch miss prediction by 2.2–74.3% with an average of 18%. Together, the two kinds of reduction accelerate the time distance measurement by

Table 1. Optimization benefits for time distance measurement

Prog	Instr. reduct (%)	Branch miss pred reduct (%)	Speedup (%)
gcc	59.7	74.3	225.7
gzip	66.4	2.6	308.0
mcf	66.5	20.9	99.2
twolf	66.8	30.8	267.0
ammp	69.2	6.5	384.1
applu	66.6	15.7	389.4
equake	69.4	3.9	354.9
mesa	67.8	4.7	522.3
mgrid	67.3	2.2	409.6
Average	**66.7**	**18.0**	**328.9**

99–522% with an average of 329%. We apply this MMU-based optimization to direct measurement of reuse distances (based on Ding and Zhong's method [7], the fastest tool we know), but only see 34.4% average speedup. This relatively modest speedup is because the major bottleneck in reuse distance measurement is in the computation of reuse distances rather than memory monitoring, whereas, the computation of a time distance is trivial—only a single reduction operation.

5.2 Locality Approximation on SPEC *ref* Runs

We use the *ref* runs of the 9 SPEC programs to evaluate the effectiveness of the scalable algorithm for reuse distance histogram approximation. Compared to the *test* and *train* runs used in our previous work [16], the *ref* runs are over 8 times longer, including a wider range of reuse distances, which pose more challenges to measurement efficiency.

We measure the accuracy of the reuse distance approximation on both element and cache-line levels. On the element level, each address is a data item; on the cache-line level, a block of consecutive memory addresses are treated as a single data item (the block size is the width of a cache line).

The accuracy is measured on a linear scale: The width—that is, the range of reuse distance—of each bar in the histograms is 1000. The formula to calculate the accuracy is $(1-\sum_i |R_i - \widehat{R_i}|/2)$, where, R_i is the Y-axis value of the ith bar in a real histogram, and $\widehat{R_i}$ is for the approximated histogram. The division by 2 normalizes the accuracy to [0,1].

As shown in Table 2, the accuracy for element reuse is 82.8% on average. Benchmark *mcf* has the lowest accuracy, 42.6%, due to the unusual thin peaks on its reuse distance histogram. The accuracy on the cache-line level is much higher, 94% for *mcf*, 98.6% on average. This higher accuracy is because a data item on this level becomes larger, and results in a smaller range of reuse distances. Most local peaks in the histograms are therefore smoothed out. Because most uses of reuse distance requires cache-line level information (e.g., for cache performance analysis), the occasional poor accuracy on element reuses has little effect. Compared to the latest reuse distance measurement [7]

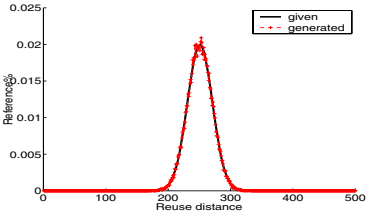
Table 2. Accuracy and speedup of reuse distance approximation

Prog	Element		Cache line	
	acc. (%)	speedup	acc. (%)	speedup
gcc	89.0	21.2X	99.4	16.7X
gzip	99.0	19.0X	99.5	17.0X
mcf	42.6	8.3X	94.0	18.2X
twolf	88.2	5.9X	98.1	20.2X
ammp	95.8	14.3X	99.2	21.5X
applu	86.1	19.0X	99.2	21.4X
equake	57.6	23.7X	98.5	15.1X
mesa	97.3	26.3X	100	14.0X
mgrid	89.7	20.6X	99.6	21.5X
Average	**82.8**	**17.6X**	**98.6**	**18.4X**

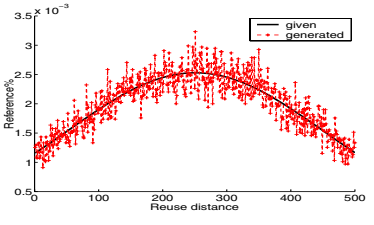
(with the MMU-based optimization), the approximation technique improves the speed by a factor of 17.

5.3 Trace Generator

We evaluate the trace generator on reuse distance histograms of some Normal and exponential distributions, two kinds of distributions that are close to typical data reuses. In the exponential distribution, the height of a bar at reuse distance of k is proportional to $e^{-0.02*k}$. In the Normal distributions, we change the variance from 20 to 200; the histograms change from a shape with thin high peaks to a flatter shape as illustrated by Figure 4. The figure also shows the histograms of the generated traces, whose curves fluctuate around the given histograms. In our experiments, each trace contains 50,000 references to 500 variables. To prevent histogram bins from hiding inaccuracy, we let each bin in the histograms have width of 1.



(a) mean=250, var=20



(b) mean=250, var=200

Fig. 4. Reuse distance histograms of two Normal distributions, along with those of the generated traces

Table 3. Accuracy of trace generation and reuse distance approximation

Distr.	Gen. acc. (%)	Approx. acc. (%)
Normal (var=20)	98.2	92.8
Normal (var=100)	96.4	96.3
Normal (var=200)	96.1	95.8
Exponential	96.0	96.9
Average	**96.7**	**95.5**

The second column in Table 3 reports the accuracy of the trace generator, reflecting the difference between the generated and the given histograms. All accuracies are greater than 96%, demonstrating the effectiveness of the trace generator in meeting the input requirements. The last column of the table contains the accuracy of the histograms that are approximated by the scalable algorithm presented in Section 2.3. On average, the accuracy is 95.5%, demonstrating the effectiveness of the algorithm in approximating reuse distance histograms of different distributions.

6 Related Work

Compiler analysis has been successful in understanding and improving locality in basic blocks and loop nests. McKinley and Temam carefully studied various types of locality within and between loop nests [14]. Cascaval presented a compiler algorithm that measures reuse distance directly [4]. Allen and Kennedy discussed the subject comprehensively in their book [1]. Thabit identified data often used together based on their access frequency [17]. Chilimbi used grammar compression to find hot data streams and reorganized data accordingly [5].

As a locality model, reuse distance has been studied for several decades since Mattson et al.'s first measurement algorithm [13]. A more recent work is from Ding and Zhong, who proposed an approximation algorithm that used dynamic tree compression [7] to reduce time complexity to $O(Tloglog N)$. Shen et al. discovered the statistical connection between time distance and reuse distance, and proposed an algorithm to approximate reuse distance from time distance [16]. This current paper exposes the extensions to the algorithm to make it more scalable, meanwhile presenting a portable scheme for resolving the bottleneck in the implementation of the algorithm, and describing a trace generator to facilitate the evaluation.

7 Conclusions

This paper presents a set of techniques to efficiently approximate reuse distance histograms on arbitrary scales. It describes an extended algorithm that significantly reduces the time complexity of the basic algorithm. It exposes a portable MMU-based optimization that accelerates time distance measurement by more than a factor of 3. Finally, it presents a trace generator that facilitates the comprehensive evaluation of the approximation algorithm. The output from this work will enhance the applicability of reuse distance in practical uses, opening new opportunities for program analysis and optimizations.

Acknowledgment. We thank the anonymous reviewers for all the helpful comments. Chen Ding suggested the lock switching scheme in the memory controller component; Brian Meeker collected some preliminary data in the early stage of this research. This material is based upon work supported by the National Science Foundation under Grant No. 0720499. Any opinions, findings, and conclusions or recommendations expressed

in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures: A Dependence-based Approach. Morgan Kaufmann Publishers, San Francisco (2001)
2. Beyls, K., D'Hollander, E.H.: Reuse distance-based cache hint selection. In: Proceedings of the 8th International Euro-Par Conference, Paderborn, Germany (August 2002)
3. Browne, S., Dongarra, J., Garner, N., London, K., Mucci, P.: A scalable cross-platform infrastructure for application performance tuning using hardware counters. In: Proceedings of Supercomputing (2000)
4. Cascaval, G.C.: Compile-time Performance Prediction of Scientific Programs. Ph.D thesis, University of Illinois, Urbana-Champaign (2000)
5. Chilimbi, T.M.: Efficient representations and abstractions for quantifying and exploiting data reference locality. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, Snowbird, Utah (June 2001)
6. Ding, C.: Improving Effective Bandwidth through Compiler Enhancement of Global and Dynamic Cache Reuse. Ph.D thesis, Dept. of Computer Science, Rice University (January 2000)
7. Ding, C., Zhong, Y.: Predicting whole-program locality with reuse distance analysis. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, San Diego, CA (June 2003)
8. Fang, C., Carr, S., Onder, S., Wang, Z.: Instruction based memory distance analysis and its application to optimization. In: Proceedings of International Conference on Parallel Architectures and Compilation Techniques, St. Louis, MO (2005)
9. Huang, S.A., Shen, J.P.: The intrinsic bandwidth requirements of ordinary programs. In: Proceedings of the 7th International Conferences on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA (October 1996)
10. Li, Z., Gu, J., Lee, G.: An evaluation of the potential benefits of register allocation for array references. In: Workshop on Interaction between Compilers and Computer Architectures in conjunction with the HPCA-2, San Jose, California (February 1996)
11. Luk, C.-K., et al.: Pin: Building customized program analysis tools with dynamic instrumentation. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, Chicago, Illinois (June 2005)
12. Marin, G., Mellor-Crummey, J.: Cross architecture performance predictions for scientific applications using parameterized models. In: Proceedings of Joint International Conference on Measurement and Modeling of Computer Systems, NewYork City, NY (June 2004)
13. Mattson, R.L., Gecsei, J., Slutz, D., Traiger, I.L.: Evaluation techniques for storage hierarchies. IBM System Journal 9(2), 78–117 (1970)
14. McKinley, K.S., Temam, O.: Quantifying loop nest locality using SPEC 1995 and the perfect benchmarks. ACM Transactions on Computer Systems 17(4), 288–336 (1999)
15. Shen, X., Shaw, J., Meeker, B.: Accurate approximation of locality from time distance histograms. Technical Report TR902, Computer Science Department, University of Rochester (2006)
16. Shen, X., Shaw, J., Meeker, B., Ding, C.: Locality approximation using time. In: Proceedings of the ACM SIGPLAN Conference on Priciples of Programming Languages, 7 pages (2007) (short paper)

17. Thabit, K.O.: Cache Management by the Compiler. Ph.D thesis, Dept. of Computer Science, Rice University (1981)
18. Zhao, Q., Sim, J.E., Wong, W.-F., Rudolph, L.: DEP: detailed execution profile. In: Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (2006)
19. Zhong, Y., Dropsho, S.G., Shen, X., Studer, A., Ding, C.: Miss rate prediction across program inputs and cache configurations. IEEE Transactions on Computers 56(3) (2007)
20. Zhong, Y., Orlovich, M., Shen, X., Ding, C.: Array regrouping and structure splitting using whole-program reference affinity. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (June 2004)

P-OPT: Program-Directed Optimal Cache Management*

Xiaoming Gu[1], Tongxin Bai[2], Yaoqing Gao[3], Chengliang Zhang[4], Roch Archambault[3], and Chen Ding[2]

[1] Intel China Research Center, Beijing, China
[2] Department of Computer Science, University of Rochester, New York, USA
[3] IBM Toronto Software Lab, Ontario, Canada
[4] Microsoft Redmond Campus, Washington, USA
{xiaoming,bai,zhangchl,cding}@cs.rochester.edu
{ygao,archie}@ca.ibm.com

Abstract. As the amount of on-chip cache increases as a result of Moore's law, cache utilization is increasingly important as the number of processor cores multiply and the contention for memory bandwidth becomes more severe. Optimal cache management requires knowing the future access sequence and being able to communicate this information to hardware. The paper addresses the communication problem with two new optimal algorithms for *Program-directed OPTimal cache management (P-OPT)*, in which a program designates certain accesses as bypasses and trespasses through an extended hardware interface to effect optimal cache utilization. The paper proves the optimality of the new methods, examines their theoretical properties, and shows the potential benefit using a simulation study and a simple test on a multi-core, multi-processor PC.

1 Introduction

Memory bandwidth has become a principal performance bottleneck on modern chip multi-processors because of the increasing contention for off-chip data channels. Unlike the problem of memory latency, the bandwidth limitation cannot be alleviated by data prefetching or multi-threading. The primary solution is to minimize the cache miss rate. Optimal caching is NP-hard if we consider computation and data reorganization [8, 12]. If we fix the computation order and the data layout, the best caching is given by the optimal replacement strategy, MIN [2]. Since MIN cannot be implemented purely in hardware, today's machines use variations of LRU (least recently used) and random replacement. This leaves room for significant improvement—LRU can be worse than optimal by a factor proportional to the cache size in theory [14] and by a hundred folds in practice [6].

* The research was conducted while Xiaoming Gu and Chengliang Zhang were graduate students at the University of Rochester. It was supported by two IBM CAS fellowships and NSF grants CNS-0720796, CNS-0509270, and CCR-0238176.

J.N. Amaral (Ed.): LCPC 2008, LNCS 5335, pp. 217–231, 2008.

Recent architectural designs have added an interface for code generation at compile time to influence the hardware cache management at run time. Beyls and D'Hollander used the cache-hint instructions available on Intel Itanium to specify which level of cache to load a data block into [5]. Wang et al. studied the use of the evict-me bit, which, if set, informs the hardware to replace the block in the cache first when space is needed [15]. The techniques improve cache utilization by preserving the useful data in cache either explicitly through cache hints or implicitly through the eviction of other data. The advent of collaborative cache management raises the question of whether or not optimal cache management is now within reach.

In the paper, we will prove that in the ideal case where the operation of each access can be individually specified, two simple extensions to the LRU management can produce optimal results. We will first discuss the optimal algorithm MIN and its stack implementation OPT. We then describe a new, more efficient implementation of OPT called OPT* and the two LRU extensions, *bypass LRU* and *trespass LRU*, that use OPT* in off-line training and to generate annotated traces for the two LRU extensions. We will show an interesting theoretical difference that bypass LRU is not a stack algorithm but trespass LRU is. Finally, we will demonstrate the feasibility of program annotation for bypass LRU and the potential improvement on a multi-core, multi-processor PC.

2 Two New Optimal Cache Management Algorithms

In this paper an *access* means a memory operation (load/store) at run time and a *reference* means a memory instruction (load/store) in the executable binary. An access is a *hit* or *miss*, depending whether the visited data element is in cache immediately before the access.

The operation of an access has three parts: the *placement* of the visited element, the *replacement* of an existing element if the cache is full, and the *shift* of the positions or priorities of the other elements. The shift may or may not be an actual action in the hardware, depending on implementation.

Following the classic model of Mattson et al. [11], we view cache as a stack or an ordered array. The data element at the top of the array has the highest priority and should be the last to evict, and the data element at the bottom is the next to evict when space is needed.

The original MIN solution by Belady is costly to implement because it requires forward scanning to find the cache element that has the furthest reuse [2]. Noting this problem, Mattson et al. described a two-pass stack algorithm, which computes the forward reuse distance in the first pass and then in the second pass maintains a priority list based on the pre-computed forward reuse distance [11]. Mattson et al. gave a formal proof the optimality in the 7-page appendix through four theorems and three lemmata. They called it the OPT algorithm. The main cost of OPT is the placement operation, which requires inserting an element into a sorted list. In comparison, the cost of LRU is constant. It places the visited element at the top of the LRU stack, which we call the *Most Recently Used (MRU)*

Table 1. The time complexity of cache-management algorithms in terms of the cost per access for placement, shift, and replacement operations. N is the length of the trace, and M is the size of cache.

Policies	Placement Cost	Shift Cost	Replacement Cost	Optimal?	Stack Alg?
MIN	constant	none	$O(N+M)$ for forward scanning plus selection	Yes	Yes
OPT	$O(M)$ list insertion	$O(M)$ update	constant	Yes	Yes
LRU	constant	none	constant	No	Yes
OPT*	$O(\log M)$ list insertion	none	constant	Yes	Yes
bypass LRU	constant	none	constant	Yes	No
trespass LRU	constant	none	constant	Yes	Yes

position, and it evicts the bottom element, which we call the *Least Recently Used (LRU)* position.

Table 1 compares six cache-management algorithms mentioned in this paper, where M is the cache size and N is the length of the access sequence. The first three rows show the cost of MIN, OPT, and LRU, and the next three rows show the algorithms we are to present: OPT*, bypass LRU, and trespass LRU. It shows, for example, that the original OPT algorithm requires an update cost of $O(M)$ per data access, but OPT* needs no such update and that bypass and trespass LRU have the same cost as LRU. Note that the cost is for the on-line management. The two LRU extensions require running OPT* in the training analysis. All but LRU can achieve optimal cache utilization. All but bypass LRU are stack algorithms.

Bypass and Trespass LRU algorithms use training analysis to specify the type of each cache access. Next we describe the three types of cache access and the OPT* algorithm used in training.

2.1 Three Types of Cache Access

We describe the normal LRU access and define the hardware extensions for bypass LRU access and trespass LRU access.

- *Normal LRU access* uses the most-recently used position for placement and the least-recently used position for replacement
  - Miss: Evict the data element S_m at LRU position (bottom of the stack) if the cache is full, shift other data elements down by one position, and place w, the visited element, in the MRU position (top of the stack). See Figure 1.
  - Hit: Find w in cache, shift the elements over w down by one position, and re-insert w at the MRU position. See Figure 2. Note that search cost is constant in associative cache where hardware checks all entries in parallel.
- *Bypass LRU access* uses the LRU position for placement and the same position for replacement. It is similar to the bypass instruction in IA64 [1] except that its bypass demotes the visited element to LRU position when hit.

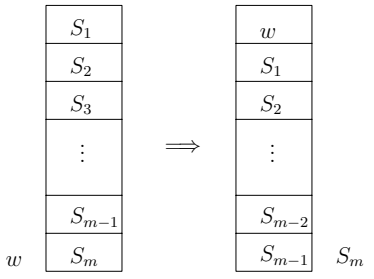


Fig. 1. Normal LRU at a miss: w is placed at the top of the stack, evicting S_m

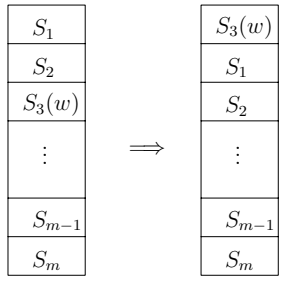


Fig. 2. Normal LRU at hit: w, assuming at entry S_3, is moved to the top of the stack

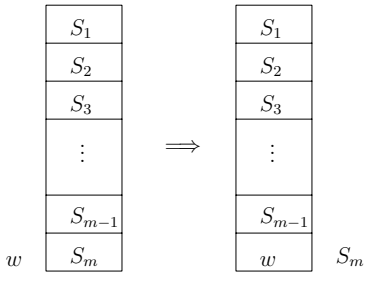


Fig. 3. Bypass LRU at a miss: the bypass posits w at the bottom of the stack, evicting S_m

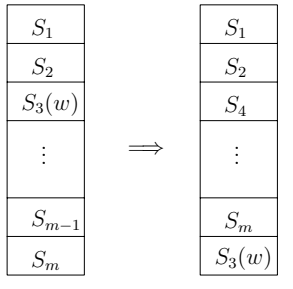


Fig. 4. Bypass LRU at a hit: the bypass moves $S_3(w)$ to the bottom of the stack

- Miss: Evict S_m at the LRU position if the cache is full and insert w into the LRU position. See Figure 3.
- Hit: Find w, lift the elements under w by one position, and place w in the LRU position. See Figure 4.

– *Trespass LRU access* uses the most-recently used position for placement and the same position for replacement. It differs from all cache replacement policies that we are aware of in that both the cache insertion and eviction happen at one end of the LRU stack.

- Miss: Evict the data element S_1 at the MRU position if the cache is not empty and insert w in the MRU position. See Figure 5.
- Hit: If w is in the MRU position, then do nothing. Otherwise, evict the data element S_1 at the MRU position, insert w there, and shift the elements under the old w spot up by one position. See Figure 6.

2.2 OPT* Algorithm

Given a memory access sequence, the original OPT algorithm has two passes [11]:

– First pass: Compute the forward reuse distance for each access through a backward scan of the trace.

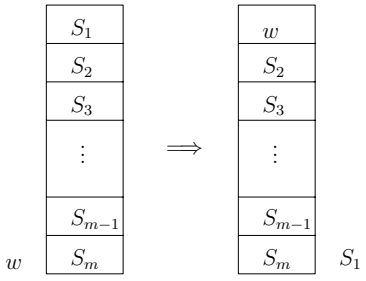


Fig. 5. Trespass LRU at a miss: the trespass posits w at the top of the stack, evicting S_1

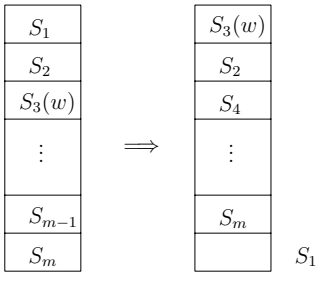


Fig. 6. Trespass LRU at a hit: the trespass raises $S_3(w)$ to the top of the stack, evicting S_1

– Second pass: Incrementally maintain a priority list based on the forward reuse distance of the cache elements. The pass has two steps. First, if the visited element is not in cache, find its place in the sorted list based on its forward reuse distance. Second, after each access, update the forward reuse distance of each cache element.

The update operation is costly and unnecessary. To maintain the priority list, it is sufficient to use the next access time instead of the forward reuse distance. At each point p in the trace, the next access time of data x is the logical time of the next access of x after p. Since the next access time of data x changes only at each access of x, OPT* stores a single next access time at each access in the trace, which is the next access time of the element being accessed. OPT* collects next access times through a single pass traversal of the trace. The revised algorithm OPT* is as follows.

– First pass: Store the next reuse time for each access through a backward scan of the trace.
– Second pass: Maintaining the priority list based on the next reuse time. It has a single step. If the visited element is not in cache, find its place in the sorted list based on its next access time.

The cost per operation is $O(\log M)$ for cache of size M, if the priority list is maintained as a heap. It is asymptotically more efficient than the $O(M)$ per access cost of OPT. The difference is computationally significant when the cache is large. While OPT* is still costly, it is used only for pre-processing and adds no burden to on-line cache management.

2.3 The Bypass LRU Algorithm

In bypass LRU, an access can be a normal access or a bypass access, which are described in Section 2.1. The type of each access is determined using OPT* in the training step. To ensure optimality, the trace of the actual execution is the same as the trace used in training analysis. For each miss in OPT*, let d be the

element evicted and x be the last access of d before the eviction, the training step would tag x as a bypass access. After training, the untagged accesses are normal accesses.

The training result is specific to the cache size being used. We conjecture that the dependence on cache size is unavoidable for any LRU style cache to effect optimal caching. The result is portable, in the sense that the performance does not degrade if an implementation optimized for one cache size is used on a machine with a larger cache. A compiler may generate code for a conservative size at each cache level or generate different versions of the code for different cache sizes for some critical parts if not whole application. Finally, the training for different cache sizes can be made and the access type specified for each cache level in a single pass using OPT*.

Two examples of bypass LRU are shown in Table 2 to demonstrate the case where the cache is managed with the same constant cost per access as LRU, yet the result is optimal, as in OPT*.

Bypass LRU is not a stack algorithm. This is shown using a counter example. By comparing the two sub-tables in Table 2, we see that at the first access to e, the stack content, given in bold letters, is (**e,d**) in the smaller cache and (**e, c, b**) in the larger cache. Hence the inclusion property does not hold and bypass LRU is not a stack algorithm [11].

Table 2. Two examples showing bypass LRU is optimal but is not a stack algorithm

(a) cache size = 2

Trace	a	b	c	d	d	c	e	b	e	c	d
Bypasses	X	X			X	X			X	X	
Bypass LRU Stack	a	b		c	d	c	**d**	e	e	c	b
Misses	1	2	3	4			5	6		7	8
OPT* Stack	a	b	c	d	d	c	e	b	e	c	d
		a	b	c	c	d	c	e	b	b	b
Misses	1	2	3	4			5	6		7	8

(b) cache size = 3

Trace	a	b	c	d	d	c	e	b	e	c	d
Bypasses	X				X					X	
Bypass LRU Stack		b	c	d	c	c	**e**	b	e	e	d
			b	c	b	b	**c**	e	b	b	e
	a			b	d	d	**b**	c	c	c	b
Misses	1	2	3	4			5				6
OPT* Stack	a	b	c	d	d	c	e	b	e	c	d
		a	b	c	c	d	c	e	b	e	e
			a	b	b	b	b	c	c	b	b
Misses	1	2	3	4			5				6

Bypass LRU is optimal. In Figure 2, bypass LRU has the same number of cache misses as OPT*, which is optimal. We next prove the optimality for all traces.

Lemma 1. *If the bottom element in the bypass LRU stack is last visited by a normal access, then all cache elements are last visited by some normal accesses.*

Proof. If some data elements are last visited by bypass accesses, then they appear only at the bottom of the stack. They can occupy multiple positions but cannot be lifted up over an element last visited by a normal access. Therefore, if the

bottom element is last visited by a normal access, all elements in the cache must also be. ■

Theorem 1. *Bypass LRU generates no more misses than OPT*. In particular, bypass LRU has a miss only if OPT* has a miss.*

Proof. We show that there is no access that is a cache hit in OPT* but a miss in Bypass LRU. Suppose the contrary is true. Let $z\prime$ be the first access in the trace that hits in OPT* but misses in Bypass LRU. Let d be the element accessed by $z\prime$, z be the immediate previous access to d, and the reference trace between them be $(z, ..., z\prime)$.

The access z can be one of the two cases.

- z is a normal access. For z' to miss in bypass LRU, there should be a miss y in $(z, ..., z\prime)$ that evicts d. From the assumption that $z\prime$ is the earliest access that is a miss in bypass LRU but a hit in OPT*, y must be a miss in OPT*. Consider the two possible cases of y.
  - y occurs when the OPT* cache is partially full. Since the OPT* cache is always full after the loading of the first M elements, where M is the cache size, this case can happen only at the beginning. However, when the cache is not full, OPT* will not evict any element. Hence this case is impossible.
  - y occurs when the OPT* cache is full. The element d is at the LRU position before the access of y. According to Lemma 1, the bypass LRU cache is full and the last accesses of all data elements in cache are normal accesses. Let the set of elements in cache be T for bypass LRU and T^* for OPT*. At this time (before y), the two sets must be identical. The reason is a bit tricky. If there is an element d' in the bypass LRU cache but not in the OPT* cache, d' must be replaced by OPT* before y. However, by the construction of the algorithm, the previous access of d' before y should be labeled a bypass access. This contradicts to the lemma, which says the last access of d' (and all other elements in T) is normal. Since both caches are full, they must be identical, so we have $T = T^*$. Finally, y in the case of OPT* must evict some element. However, evicting any element other than d would violate our lemma. Hence, such y cannot exist and this case is impossible.
- z is a bypass access in Bypass LRU. There must be an access $y \in (z, ..., z\prime)$ in the case of OPT* that evicts d; otherwise z cannot be designated as a bypass. However, in this case, the next access of d, $z\prime$ cannot be a cache hit in OPT*, contradicting the assumption that $z\prime$ is a cache hit in OPT*.

Considering both cases, it is impossible for the same access to be a hit in OPT* but a miss in bypass LRU. ■

Since OPT* is optimal, we have the immediate corollary that bypass LRU has the same number of misses as OPT* and is therefore optimal. In fact, the misses

happen for the same accesses in bypass LRU and in OPT*. Last, we show that Bypass LRU as a cache management algorithm has a peculiar feature.

Corollary 1. *Although Bypass LRU is not a stack algorithm, it does not suffer from Belady anomaly [3], in which the number of misses sometimes increases when the cache size becomes larger.*

Proof. OPT is a stack algorithm since the stack content for a smaller cache is a subset of the stack content for a larger cache [11]. The number of misses of an access trace does not increase with the cache size. Since bypass LRU has the same number of misses as OPT*, it has the same number of misses as OPT and does not suffer from Belady anomaly. ■

2.4 The Trespass LRU Algorithm

In trespass LRU, an access can be a normal access or a trespass access. The two obvious choices for efficient LRU stack replacement are evicting from the bottom, as in bypass LRU just described, or evicting from the top, as in trespass LRU. Both are equally efficient at least asymptotically. We will follow a similar approach to show the optimality of trespass LRU. The main proof is actually simpler. We then show an unexpected theoretical result—trespass LRU is a stack algorithm, even though bypass LRU is not.

Similar to bypass LRU, trespass LRU uses a training step based on simulating OPT* for the given cache on the given trace. For each miss y in OPT*, let d be the evicted cache element and x be the last access of d before y. The training step tags the access immediately after x as a trespass access. It is trivial to show that such an access exists and is unique for every eviction in OPT*.

Two example executions of trespass LRU execution are shown in Table 3 for the same trace used to demonstrate bypass LRU in Table 2.

Table 3. Two examples showing trespass LRU is optimal and is a stack algorithm (unlike bypass LRU)

(a) cache size = 2

Trace	a	b	c	d	d	c	e	b	e	c	d
Trespasses		X	X			X	X			X	X
Trespass LRU Stack	a	b	c	d	d	c	e	b	e	c	d
				c	c			e	b	b	b
Misses	1	2	3	4			5	6		7	8
OPT* Stack	a	b	c	d	d	c	e	b	e	c	d
		a	b	c	c	d	c	e	b	b	b
Misses	1	2	3	4			5	6		7	8

(b) cache size = 3

Trace	a	b	c	d	d	c	e	b	e	c	d
Trespasses		X				X					X
Trespass LRU Stack	a	b	c	d	d	c	e	b	e	c	d
			b	c	c	b	c	e	b	e	e
				b	b		b	c	c	b	b
Misses	1	2	3	4			5				6
OPT* Stack	a	b	c	d	d	c	e	b	e	c	d
		a	b	c	c	d	c	e	b	e	e
			a	b	b	b	b	c	c	b	b
Misses	1	2	3	4			5				6

Trespass LRU is Optimal. The effect of a trespass access is less direct than that of a bypass access. We need four additional lemmata. First, from the way trespass accesses are identified, we have

Lemma 2. *If a data element w is evicted by a trespass access x, then x happens immediately after the last access of w.*

Lemma 3. *If a data element is in trespass LRU cache at point p in the trace, then the element is also in OPT* cache at p.*

Proof. Assume that a data element w is in the trespass LRU cache but is evicted from the OPT* cache. Let x be the last access of w. Consider the time of the eviction in both cases. The eviction by trespass LRU happens right after x. Since the eviction by OPT* cannot be earlier, there must be no period of time when an element w is in the trespass LRU cache but not in the OPT* cache. ■

Lemma 4. *If a data element is evicted by a normal access in trespass LRU, then the cache is full before the access.*

This is obviously true since the normal access cannot evict any element unless the cache is full. Not as obvious, we have the following

Lemma 5. *A normal access cannot evict a data element from cache in trespass LRU.*

Proof. Assume y is a normal access that evicts data w. Let T and T^* be the set of data elements in the Trespass LRU cache and the OPT* cache before access y. By Lemma 3, $T \subseteq T^*$. By Lemma 4, the Trespass LRU cache is full before y. Then we have $T = T^*$. In OPT*, y has to evict some element $d \in T^*$. Let x be the last access of d before y. Since Trespass LRU evicts d right after x, the content of the cache, T and T^* cannot be the same unless y is the next access after x, in which case, d is w, and y must be a trespass access. ■

Theorem 2. *Trespass LRU generates no more misses than OPT*. In particular, trespass LRU has a miss only if OPT* has a miss.*

Proof. We show that there is no access that is a cache hit in OPT* but a miss in trespass LRU. Suppose the contrary is true. Let $z\prime$ be the first access in the trace that hits in OPT* but misses in Trespass LRU. Let d be the element accessed by $z\prime$, z be the immediate previous access to d, and the reference trace between them be $(z, ..., y, ..., z\prime)$, where y is the access that causes the eviction of d in trespass LRU.

By Lemma 5, y is a trespass access. By Lemma 2, y happens immediately after z. Since y is a trespass after z, then the next access of d, $z\prime$ must be a miss in OPT*. This contradicts the assumption that $z\prime$ is a hit in OPT*. Therefore, any access that is a miss in trespass LRU must also be a miss in OPT*. ■

Corollary 2. *Trespass LRU has the same number of misses as OPT* and is therefore optimal.*

Trespass LRU is a stack algorithm. Given that bypass LRU is not a stack algorithm, the next result is a surprise and shows an important theoretical difference between trespass LRU and bypass LRU.

Theorem 3. *Trespass LRU is a stack algorithm.*

Proof. Assume there are two caches C_1 and C_2. C_2 is larger than C_1, and the access sequence is $Q = (x_1, x_2, ..., x_n)$. Let $T_1(t)$ be the set of elements in cache C_1 after access x_t and $T_2(t)$ be the set of elements in cache C_2 after the same access x_t. The initial sets for C_1 and C_2 are $T_1(0)$ and $T_2(0)$, which are empty and satisfy the inclusion property. We now prove the theorem by induction on t.

Assume $T_1(t) \subseteq T_2(t)$ $(1 \leq t \leq n-1)$. There are four possible cases based on the type of the access x_{t+1} when visiting either of the two caches. We denote the data element accessed at time x_i as $D(x_i)$.

- If x_{t+1} is a trespass access both in C_1 and C_2, we have

$$\begin{aligned} T_1(t+1) &= T_1(t) - D(x_t) + D(x_{t+1}) \\ &\subseteq T_2(t) - D(x_t) + D(x_{t+1}) \\ &= T_2(t+1) \end{aligned}$$

- If x_{t+1} is a trespass access in C_1 but a normal access in C_2, then by Lemma 5, x_{t+1} does not cause any eviction in cache C_2 and therefore

$$\begin{aligned} T_1(t+1) &= T_1(t) - D(x_t) + D(x_{t+1}) \\ &\subseteq T_2(t) + D(x_{t+1}) \\ &= T_2(t+1) \end{aligned}$$

- The case that x_{t+1} is a normal access in C_1 but a trespass access in C_2 is impossible. Since x_{t+1} is a trespass in C_2, $D(x_t)$ would be evicted by some access y in C_2 using OPT*. However, x_{t+1} is a normal access in C_1, which means that $D(x_t)$ is in C_1 after access y when using OPT*. This in turn means that at the point of y, the inclusion property of OPT* no longer holds and contradicts the fact that OPT* is a stack algorithm.
- If x_{t+1} is a normal access both in C_1 and C_2, then by Lemma 5, x_{t+1} does not cause an eviction either in C_1 or C_2, and therefore

$$\begin{aligned} T_1(t+1) &= T_1(t) + D(x_{t+1}) \\ &\subseteq T_2(t) + D(x_{t+1}) \\ &= T_2(t+1) \end{aligned}$$

From the induction hypothesis, the inclusion property holds for Trespass LRU for all t. ■

The next corollary follows from the stack property.

Corollary 3. *Trespass LRU as a cache management algorithm does not suffer from Belady anomaly [3].*

In Table 3, we have revisited the same data trace used to show that bypass LRU was not a stack algorithm. It shows that the inclusion property holds when trespass LRU is used. The example also shows that trespass LRU cache can become partially empty after it becomes full. Trespass LRU keeps the visited data element and the data elements that will be visited. When the amount of data that have a future reuse is less than the cache size, OPT* and bypass LRU may contain extra data elements that have no future reuse. In OPT* the extra data do not destroy the inclusion property, but in bypass LRU they do.

2.5 Limitations

Bypass LRU and trespass LRU solve the problem of efficient on-line cache management, but their optimality depends on specifying the type of individual accesses, which is not feasible. In practice, however, a compiler can use transformations such as loop splitting to create different memory references for different types of accesses. We will show this through an example in the next section.

Another problem is the size of program data may change in different executions. We can use the techniques for predicting the change of locality as a function of the input [9, 16] and use transformations such as loop splitting to specify caching of only a constant part of the (variable-size) program data.

Throughout the paper, we use the fully associative cache as the target. The set associative cache can be similarly handled by considering it as not just one but a collection of fully associative sets. All the theorems about the Bypass and Trespass LRU hold for each fully associative set, and the optimality results stay the same, so are the stack properties.

The training analysis can also be extended naturally to tag bypass or trespass accesses for each set. There is an additional issue of the data layout, especially if it changes with the program input. Though we have not developed any concrete solutions, we believe that these problems can be approached by more sophisticated training, for example, pattern analysis across inputs, and additional program transformations such as loop splitting.

Bypass LRU is better than trespass LRU because the latter is sensitive to the order of accesses. It is possible that a trespass access is executed at an unintended time as a result of instruction scheduling by the compiler and the out-of-order execution by the hardware. In comparison, the effect of bypass is not sensitive to such reordering.

3 The Potential Improvements of P-OPT

3.1 A Simulation Study

While controlling the type of each access is impractical, we may use simple transformations to approximate bypass LRU at the program level based on the result of training analysis. Assume the fully associative cache has 512 blocks, and each block holds one array element. The code in Figure 7, when using LRU, causes 10000 capacity misses among the total 29910 accesses. The minimal

```
int a[1000]
for(j=1;j<=10;j++)
 for(i=1;i<=997;i++)
  a[i+2]=a[i-1]+a[i+1];
```

Fig. 7. Original Code

```
int a[1000]
for(j=1;j<10;j++) {
 for(i=1;i<=509;i++)
  a[i+2]=a[i-1]+a[i+1];
 for(;i<=996;i++)
  a[i+2]=a[i-1]+a[i+1];
 for(;i<=997;i++)
  a[i+2]=a[i-1]+a[i+1];
}
```

Fig. 8. Transformed Code

number of misses is half as much or, to be exact, 5392, given by OPT*. There are three array references in the original loop. OPT* shows that the accesses by reference a[i+1] are all normal accesses except for three accesses. The accesses by reference a[i+2] are all normal accesses. Finally reference a[i-1] has a cyclic pattern in every 997 accesses with about 509 normal accesses and 488 bypass accesses in each period.

Based on the training result, we split the loop into three parts as shown in Figure 8. In the first loop, all three references are *normal references*, which means the accesses by them are all normal accesses. In the second loop, the references a[i+1] and a[i+2] are normal references but reference a[i-1] is tagged with the *bypass bit*, which means that it is a *bypass reference* and its accesses are all bypass accesses. In the third loop, the references a[i+2] are normal references but references a[i-1] and a[i+1] are bypass references. The transformed program yields 5419 cache misses, an almost half reduction from LRU and almost to same as the optimal result of 5392 misses. After the transformation, it looks like we retain the first part of array a in cache but bypass the second part. Effectively it allocates the cache to some selected data to utilize cache more efficiently.

Not all programs may be transformed this way. Random access is an obvious one for which optimal solution is impossible. However, for regular computations, this example, although simple, demonstrates that P-OPT with training based bypass LRU may obtain near optimal cache performance.

3.2 A Simple But Real Test

We use a program which repeatedly writes to a contiguous area the size of which is controlled by the input as the data size. The access has the large stride of 256 bytes so the average memory access latency is high. The second input parameter to the program is the retainment size specifying how large piece of data should be retained in the cache. Since the access to the rest of the data takes space in cache, the retainment size is smaller than the cache size. Considering the set-associativity of the cache structure, we set the retainment size to $\frac{3}{4}$ of the total cache size. When multiple processes are used, the total retainment size is divided evenly among them. Our testing machine has two Core Duo chips, so cache contention only happens when four processes run together. In that case,

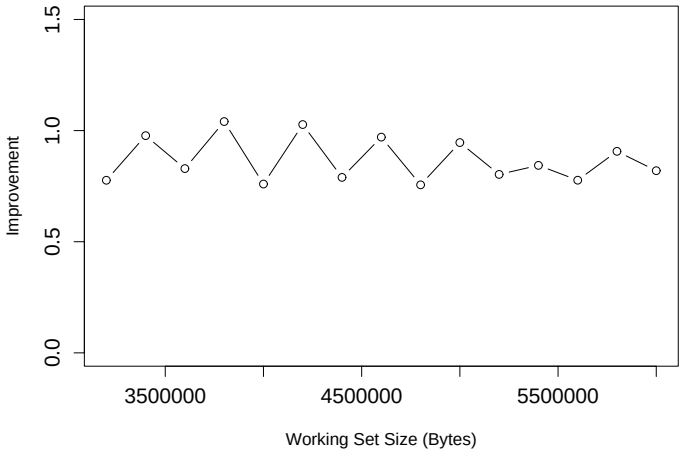


Fig. 9. Ratio of running time between using bypassing stores and using normal stores in 4 processes. The lower the ratio is, the faster the bypassing version. The machine has two Intel Core Duo processors, and each has two 3GHz cores and 4MB cache.

we give half of the total retainment size to each process. The running time is measured for 50 million memory accesses.

Figure 9 shows the effect of cache bypassing running four processes. The instruction `movnti` on Intel x86 processors is used to implement the cache bypassing store. With four processes and contention for memory bandwidth, the improvement is observed at 3.2MB and higher data sizes. The worst is 4% slowdown at 3.8MB, the best is 24% at 4.8MB, and the average is 13%. The result is tantalizing: the bypassing version runs with 13% less time or 15% higher speed, through purely software changes. Our store bypassing is a heuristic that may not be optimal. Therefore the potential improvement from bypass LRU is at least as great, if not greater.

4 Related Work

The classic studies in memory management [2,11] and self-organizing data structures [14] considered mostly uniform data placement strategies such as LRU, OPT, and MIN. This paper establishes a theoretical basis for selective replacement strategies that are optimal yet can be implemented on-line with the same cost as LRU. The cost of annotation is shifted to an off-line step.

The use of cache hints and cache bypassing at the program level is pioneered by two studies in 2002. Beyls and D'Holander used a training-based analysis for inserting cache hints [4]. By instructing the cache to replace cache blocks that have long-distance reuses, their method obtained 9% average performance improvement on an Intel Itanium workstation [4]. Wang et al. published a set of compiler techniques that identified different data reuse levels in loop nests and inserted evict-me bits for references for which the reuse distance was larger

than the cache size [15]. Beyls and D'Holander later developed a powerful static analysis (based on the polyhedral model and integer equations and called reuse-distance equations) and compared it with training-based analysis for scientific code [5].

Our scheme differs from the two earlier methods because bypass and trespass LRU are designed to preserve in cache data blocks that have long-distance reuses. Intuitively speaking, the goal of the previous methods is to keep the working set in the cache if it fits, while our goal is to cache a part of the working set even if it is larger than cache. At the implementation level, our method needs to split loop iterations. Beyls and D'Holander considered dynamic hints for caching working sets that fit in cache [5]. Qureshi et al. recently developed a hardware scheme that selectively evicted data based on the predicted reuse distance [13]. The study showed significant benefits without program-level inputs. As a pure run-time solution, it naturally incorporates the organization of the cache and the dynamics of an execution, but it is also inherently limited in its predictive power.

One important issue in training based analysis is the effect of data inputs. Fang et al. gave a predictive model that predicted the change of the locality of memory references as a function of the input [9]. They showed on average over 90% accuracy across program inputs for 11 floating-point and 11 integer programs from the SPEC2K CPU benchmark suite. Their result suggested that training based analysis can accurately capture and exploit the reuse patterns at the memory reference level. Marin and Mellor-Crummey demonstrated the cross-input locality patterns for larger code units such as loops and functions [10]. In addition for scientific code, compiler analysis can often uncover the reuse-distance pattern, as demonstrated by Cascaval and Padua [7] and Beyls and D'Holander [5], and eliminate the need of training analysis.

5 Summary

In this paper we have presented two new cache management methods, bypass LRU and trespass LRU, which are programmable by software, require similar hardware support as LRU, and may produce the same result as optimal cache management. Bypass LRU is not a stack algorithm while trespass LRU is. Both require training analysis, for which we have presented OPT*, asymptotically the fastest implementation of optimal cache management. We have demonstrated preliminary evidence that bypass LRU can be effectively used by a combination of off-line training and program transformation. The better utilization of cache has led to significant performance improvement for parallel memory traversals on multi-core processors.

References

1. IA-64 Application Developer's Architecture Guide (May 1999)
2. Belady, L.A.: A study of replacement algorithms for a virtual-storage computer. IBM Systems Journal 5(2), 78–101 (1966)

3. Belady, L.A., Nelson, R.A., Shedler, G.S.: An anomaly in space-time characteristics of certain programs running in a paging machine. Commun. ACM 12(6), 349–353 (1969)
4. Beyls, K., D'Hollander, E.: Reuse distance-based cache hint selection. In: Proceedings of the 8th International Euro-Par Conference, Paderborn, Germany (August 2002)
5. Beyls, K., D'Hollander, E.: Generating cache hints for improved program efficiency. Journal of Systems Architecture 51(4), 223–250 (2005)
6. Burger, D.C., Goodman, J.R., Kagi, A.: Memory bandwidth limitations of future microprocessors. In: Proceedings of the 23th International Symposium on Computer Architecture, Philadelphia, PA (May 1996)
7. Cascaval, C., Padua, D.A.: Estimating cache misses and locality using stack distances. In: Proceedings of International Conference on Supercomputing, San Francisco, CA (June 2003)
8. Ding, C., Kennedy, K.: Improving effective bandwidth through compiler enhancement of global cache reuse. Journal of Parallel and Distributed Computing 64(1), 108–134 (2004)
9. Fang, C., Carr, S., Onder, S., Wang, Z.: Instruction based memory distance analysis and its application to optimization. In: Proceedings of the International Conference on Parallel Architecture and Compilation Techniques, St. Louis, MO (2005)
10. Marin, G., Mellor-Crummey, J.: Scalable cross-architecture predictions of memory hierarchy response for scientific applications. In: Proceedings of the Symposium of the Las Alamos Computer Science Institute, Sante Fe, New Mexico (2005)
11. Mattson, R.L., Gecsei, J., Slutz, D., Traiger, I.L.: Evaluation techniques for storage hierarchies. IBM System Journal 9(2), 78–117 (1970)
12. Petrank, E., Rawitz, D.: The hardness of cache conscious data placement. In: Proceedings of ACM Symposium on Principles of Programming Languages, Portland, Oregon (January 2002)
13. Qureshi, M.K., Jaleel, A., Patt, Y.N., Steely Jr., S.C., Emer, J.S.: Adaptive insertion policies for high performance caching. In: Proceedings of the International Symposium on Computer Architecture, San Diego, California, USA, June 2007, pp. 381–391 (2007)
14. Sleator, D.D., Tarjan, R.E.: Amortized efficiency of list update and paging rules. Communications of the ACM 28(2) (1985)
15. Wang, Z., McKinley, K.S., Rosenberg, A.L., Weems, C.C.: Using the compiler to improve cache replacement decisions. In: Proceedings of the International Conference on Parallel Architecture and Compilation Techniques, Charlottesville, Virginia (September 2002)
16. Zhong, Y., Dropsho, S.G., Shen, X., Studer, A., Ding, C.: Miss rate prediction across program inputs and cache configurations. IEEE Transactions on Computers 56(3) (2007)

Compiler-Driven Dependence Profiling to Guide Program Parallelization

Peng Wu[1], Arun Kejariwal[2], and Călin Caşcaval[1]

[1] Programming Models and Tools for Scalable Systems
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598, USA
[2] Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697, USA

Abstract. As hardware systems move toward multicore and multi-threaded architectures, programmers increasingly rely on automated tools to help with both the parallelization of legacy codes and effective exploitation of all available hardware resources. Thread-level speculation (TLS) has been proposed as a technique to parallelize the execution of serial codes or serial sections of parallel codes. One of the key aspects of TLS is task selection for speculative execution.

In this paper we propose a cost model for compiler-driven task selection for TLS. The model employs profile-based analysis of *may*-dependences to estimate the probability of successful speculation. We discuss two techniques to eliminate potential inter-task dependences, thereby improving the rate of successful speculation. We also present a profiling tool, **DProf**, that is used to provide run-time information about *may*-dependences to the compiler and map dynamic dependences to the source code. This information is also made available to the programmer to assist in code rewriting and/or algorithm redesign.

We used **DProf** to quantify the potential of this approach and we present results on selected applications from the SPEC CPU2006 and SEQUOIA benchmarks.

1 Introduction

Thread-level speculation (TLS) [11, 15, 26, 28] is one technique that has been proposed for parallelizing sequential codes to exploit parallel and multi-core architectures. Parallelization using TLS consists of selecting regions of code to execute in parallel, relying on the system to detect dependence violations and re-execute the conflicting sections such that sequential execution semantics is preserved. Typically the code regions are loop iterations and function continuations. A number of researchers made the case that automatically speculating on inner loops at the granularity of single iteration is not very effective for the applications in the SPEC CPU2006 benchmark suite, and gives little advantge over a state-of-the-art parallelizing compiler [14]. This highlights the importance of

J.N. Amaral (Ed.): LCPC 2008, LNCS 5335, pp. 232–248, 2008.

task selection towards the efficacy of TLS. Task selection can be done either automatically using a compiler, or manually through programmer annotations. Liu et al. [17] and Johnson et al. [12, 13] propose mechanisms to automatically identify most profitable tasks for speculation through profiling information [13, 17] or empirical search [12]. On the other side of the spectrum, von Praun et al. [30] argue for user annotation of the speculative tasks and provide a tool that can classify program sections and recommend task placement directives.

It is obvious that programmers need tools to help them make parallelization decisions. This includes choosing a suitable parallelization strategy for a given application, e.g., speculative vs. non-speculative; task selection for speculation; or algorithm restructuring to expose parallelism. We propose a new model that uses compiler analysis and profiling to guide parallelization and task selection, in an attempt to reach a middle ground: the compiler and tools provide as much information as possible and prune the space, so that the user could focus on those parts of the application that may need rewriting and algorithm redesign.

In this paper, we present a compiler-driven approach for program dependence profiling and a cost model to identify loops suitable for TLS parallelism. One metric used in the cost model is the distance between consecutive dependent iterations [20], referred to as the *independence window*. If the independence window is larger than the speculation window (the number of tasks that are potentially executing in parallel at any point in time), the dependence does not affect TLS effectiveness. The Independence window is a dynamic property of a loop since it depends on the iteration schedule.

We developed a dependence profiler, referred to as **DProf**, to measure dependence probability and independence window. Profiling can be implemented using a dynamic binary instrumenter [29] or using compiler instrumentation. When using a binary instrumenter, program properties known to the compiler are lost or hard to obtain at binary level; "transferring" the dependence information from a trace to the compiler is quite involved: the binary instrumenter collects physical addresses which need to be mapped to the variables in the program. In Section 3, we present a compiler-based approach for dependence profiling that overcomes these limitations. We present the dependence and independence window profile obtained by **DProf** for selected programs from the SPEC CPU2006 and SEQUOIA benchmarks. Both the independence window and dependence data measured by the profiler provide useful feedback to the compiler to perform dependence tolerating transformations or to the programmer to restructure algorithms for parallelism.

The main contributions of this paper are:

- A static model for TLS profitability that is used by the compiler to select tasks for speculation;
- A compiler-driven approach for program dependence profiling;
- Two techniques – independence windows and dependence clustering – for increasing the profitability of TLS.

The rest of the paper is organized as follows: Section 2 describes our static model for TLS. The design of **DProf** is described in detail in Section 3 and its

applications are discussed in Section 4. Previous work is discussed in Section 5. Finally, in Section 6 we summarize with directions for future work.

2 Static Modeling of TLS Profitability

In this section we present a cost model used by the compiler to determine profitable TLS code sections. There are three main factors that determine the profitability of speculative execution:

1. Conflict probability (C): defines what is the probability that two speculative tasks access the same data, and at least one is a write, such that they conflict and the speculative execution must be squashed. The conflict probability is a function of the data dependences in the task and of the task size. We discuss the conflict probability in detail in Section 2.1;
2. Speculative spawn and commit overheads (O) which are system dependent costs of speculation, for both successful and aborted execution;
3. Task sizes (S_i), which determine the the fraction of useful work, as well as influence conflict probability – the larger the size the larger the set of data accessed, and efficiency due to load balancing issues.

A TLS section of code is profitable if the time required for parallel execution (T_p), including the overheads, is less than the time required for serial execution (T_s), i.e., $T_p < T_s$. T_s and T_p for a set of N tasks running on P processors can be expressed as follows:

$$T_s = \sum_{i}^{N} S_i \tag{1}$$

$$T_p = \sum_{k} \frac{N \times (1-C)^k}{P} (\max_i(S_i) + O) \tag{2}$$

$$T_p = \frac{N \times (1-C)}{P} (\max_i(S_i) + O) + \sum_{i}^{N \times C} (S_i + O) \tag{3}$$

The serial execution time T_s (Eqn. 1) is the sum of all tasks. The parallel execution time T_p (Eqn. 2) takes into consideration the number of processors and the conflict probability. Eqn. 3 is a simplified version of Eqn. 2 that uses the following assumption: $N \times (1-C)$ tasks can all execute in parallel ($\max_i(S_i)$ determining the parallel execution time), with no other overhead in addition to the spawn and commit overhead O; the other $N \times C$ tasks are all serialized. Of course, this is a conservative assumption, because within the remaining tasks there may be independent sets of tasks, but between the overhead of spawning a task multiple times, and the diminishing returns of executing conflicting task versus just serializing execution, we select the latter.

The compiler may obtain the needed parameters for the model either statically using analysis, or through profiling, as follows:

- New data dependence analysis that takes into consideration dependence probabilities; or profiling information that estimates the conflict probability for a set of tasks;
- A cost model for task sizes; or profiling information that quantifies the sizes of tasks;
- System latencies and overheads for TLS support.

For this paper, we explore the effectiveness of the cost model, and thus we collect profiling information using **DProf** (see Section 3) that provides feedback information on data dependences and task sizes.

2.1 Conflict Probability

We use the metric of conflict probability to determine the likelihood of the speculation execution of a code region success. A conflict probability of 0 implies that the region is independent of all other regions with which it has the potential to run in parallel, while a conflict probability of 1 guarantees that a conflict will happen and the speculation will fail. Intuitively, the larger a code region, the higher the probability of conflict. However, there are many cases in which large code regions are independent, e.g., iterations of DOALL loops.

The conflict probability is computed using the data dependence density metric. In [29], von Praun et al. used the data dependence density metric to determine the available parallelism in an application. They argue that the amount of exploitable parallelism in the application is dependent on the scheduling of threads, and classified application phases into three categories: high-, medium-, and low-dependence density. The low-dependence density regions are the most profitable for speculation. Because we are focusing on loop iterations, and considering two dependent iterations t and s, we can simplify the data dependence density computation from [29] and use the following formula for conflict probability:

$$C(t) := \frac{\sum_{\forall s, has\_dep(t,s)} S_s}{\sum_{i}^{N} S_i} \tag{4}$$

In this paper, the conflict probability is used directly with the speculation overheads and task sizes (Eqn. 3) to select the profitable tasks.

In addition, the compiler can also increase the probability of successful speculation by taking advantage of patterns of dependences. We present two such techniques – the independence window and dependence clustering.

2.2 Independence Windows

An *independence window* is a set of consecutive iterations that are independent of each other. Iterations in the set – the independence window – can be executed in parallel. We call the cardinality of the set the width of an independence window.

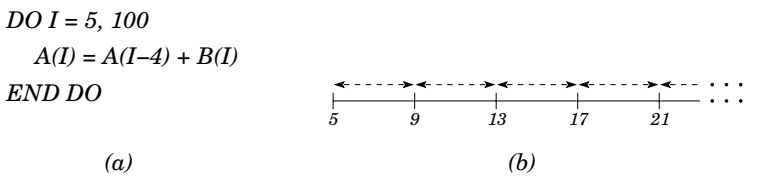


Fig. 1. An example illustrating the independence window

The entire iteration space can be viewed as a partitioned set of independence windows. Consider the loop shown in Figure 1 (a) and the corresponding iteration space shown in Figure 1 (b). The loop has a loop-carried dependence with dependence distance of 4. Therefore, every 4 iterations, marked with dashed arrows in Figure 1 (b), can be executed in parallel. For loops with a constant dependence distance of n, the iteration space is equally partitioned into independence windows of width n. For irregular loops, the independence window is a dynamic property determined by the iteration schedule.

Assuming that tasks are scheduled and retired at a uniform rate at the granularity of one iteration, a loop with an independence window of width n can be parallelized with zero conflicts using no more than n speculative threads. In other words, the width of independence window gives the theoretical upper bound on the size of non-conflicting speculation.

We use the width of independence windows as a metric of dynamic parallelism, especially for loops that do not have a uniform dependence distance. For such loops, the width of the independence window varies across the iteration space, e.g., due to multiple dependences occurring at different intervals. We use the profiler to empirically determine the independence window width of such loops. In our profiler, we compute the maximum, minimum, and average widths of independence windows to capture dynamic widths of independence windows.

The compiler can exploit independence windows by throttling the speculation, such that dependences are naturally satisfied. This can be accomplished by either of the following methods:

- By the compiler inserting explicit synchronization, similar to the technique described in [32]; or
- Providing hints to the hardware for dynamic task merging [24]; or
- Inserting conditional spawn instructions [8] if the hardware supports it.

2.3 Dependence Clustering

Recently it was shown that the profitability of TLS is highly sensitive to the threading overhead [14]. The analysis assumed dynamically scheduling, wherein iterations of a loop are allocated one at a time. Arguably, one can unroll a loop or employ chunk scheduling, where multiple iterations are allocated either statically or dynamically to a processor, to tolerate the high threading overhead. On the other hand, this increases the probability that two speculative threads conflict with each other. However, this implicitly assumes a uniform distribution

of dependences across the iteration space: if there exists 2 dependences between iterations i and $i+1$, then iterations j and $j+1$, where $i \neq j$ will be dependent. In other words, the width of the independence window is "fixed". This assumption may not hold for loops containing conditionals, subscripted subscripts and/or function calls. For example, let us revisit the loop shown in Figure 1. The value of the variable `i` may be the same for the first 10 iterations and different in the remaining iterations; consequently, the first ten iterations have to be executed serially, whereas the remaining iterations can be executed in parallel. In such a case, we say that loop-carried dependences are *clustered* amongst the first ten iterations.

In general, the objective of *dependence clustering* is to determine, regions of the iteration space (of a given loop) with large independence windows. If such regions exist, then the iteration space is partitioned to isolate such regions and these regions are subsequently parallelized via TLS.

2.4 Summary

In Figure 2 we present a pictorial view of profitability analysis of TLS. We consider a hierarchy for non-`DOALL` loops. The first level filtering is done based on the threading overhead. Specifically, non-`DOALL` loops with small amount of computation in the loop body are filtered as these loops are *non-tolerant* (box **A** in the figure) to the threading overhead. The second level filtering is done based on conflict probability, using MinIWW (Minimum independence window width). Loops with MinIWW = 1 are classified as *non-profitable*, box $\boldsymbol{B_1}$ in the figure [14]. The rest of the loops are classified as *profitable*, box $\boldsymbol{B_2}$ in the figure. Amongst the loops belonging to box **A**, we detect loops with high iteration counts. Such loops can be migrated to box **B** via chunk scheduling; it is important to note that these loops cannot be directly migrated to box $\boldsymbol{B_2}$ as chunk scheduling may result in an increase in conflict probability. Next, amongst the loops belonging to box $\boldsymbol{B_1}$, we detect loops with large MaxIWW (Maximum independence window width), exemplified by the loop in `429.mcf`, at `implicit.c:381`

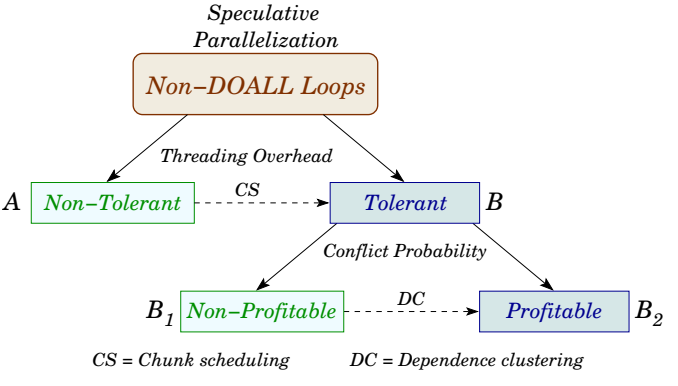


Fig. 2. Role of dependence clustering in facilitating speculative parallelization

whose profile of independence windows is shown in Figure 8. Then, parts of the iteration space (of such loops) with large value of MaxIWW are peeled and classified as profitable for TLS. The percentage of execution time spent in the peeled portions of the iteration space correspond to the performance potential of TLS. Recall that, for given a loop, the existence of independence windows with large widths and their position in the iteration space may be dependent on the input data set. Therefore, independence windows (with large widths) detected using a training data set cannot be parallelized statically.

3 Design of DProf

DProf consists of two components: a compiler-driven instrumenter that selects loops and instruments memory references for dependence profiling, and a runtime library that logs references and profiles dynamic dependences and independence window.

3.1 DProf Instrumenter

The instrumenter is based on the optimizing and parallelizing IBM XL (production) compiler. The compiler first analyzes candidate loops to decide whether they are parallel. Parallel loops are not profiled. The rest of candidate loops may be selected for profiling based on the likelihood of the dependences detected by the compiler and other characteristics of the loop. For instance, a loop with small dependence distances may not be selected for profiling if static information is sufficient to determine the loop as not a good candidate for parallelization.

For loops that are selected for profiling, the compiler transforms every memory reference that carries may-dependences into a call `__profile_access`, and marks the start, the end, and the backedge of a (possibly nested) loop by calls to `__profile_boundary`.

The compiler does not instrument references to induction, reduction, and private variables as dependences carried by these references can be eliminated via compiler transformations. To reduce profiling time, the compiler may not instrument references whose loop-carried dependences can be represented by dependence distances. In this case, the dependence information is recorded by the instrumenter and is later combined with profiled dependences to produce a complete dependence report.

3.2 DProf Runtime Library

The runtime library profiles a set of properties for loops such as whether a given loop is parallel, statistics of independence window size, and dependence properties such as the source, sink, and the frequency of dependences or dependence distances.

The profiler maintains a set of read- and write-logs for each loop: R_{curr}/W_{curr} for the current iteration being profiled, R_{indep}/W_{indep} for iterations in the current independence window, and R_{other}/W_{other} for iterations prior to those in

the current independence window. Loop data structures are kept in a stack so that nested loops can be profiled in one pass.

For each `__profile_access` call, the profiler logs the reference to R_{curr} or W_{curr} accordingly, unless the access is a read and the memory address is already in W_{curr} (i.e., the read is private to current iteration).

For each `__profile_boundary` call that marks a loop backedge, the profiler detects loop-carried dependences by comparing R_{curr} against W_{indep}. If the intersection of the two sets is not a void set, then a true dependence is detected, and the current independence window is reset to start from the current iteration after the sets R_{indep}/W_{indep} are merged to R_{other}/W_{other} respectively. Otherwise, the current independence window is grown by one iteration. Note that, in this algorithm, only true dependences and dependences to references in the latest independence window are detected.

3.3 Source Mapping

The profiler reports source-level information corresponding to the source and sink of the dynamic dependences being profiled. Such information provides a valuable guidance to the programmer to make parallelization decisions and even eliminate these dependences by making source code changes.

The source-level mapping is maintained by the instrumenter which associates each call site of `__profile_access` and `__profile_boundary` with a unique id. The instrumenter also generates a file, which is later used by the profiler, to map ids to its associated source-level information and other properties obtained by the compiler. Since the instrumenter is a part of the compiler, the same mechanism can be used to map profiled properties back to the compiler.

4 Benchmark Evaluation

We implemented **DProf** on top of the development code base of the IBM XL compiler. Essentially, an instrumentation phase is added to an optimizing component of the XL compiler called **TPO** that performs high-level optimization including parallelization. The profiler is implemented as a runtime library that is linked with the instrumented binary.

4.1 Dependence Profiling

We used **DProf** to study the parallelization potential of selected applications from the SEQUOIA [1] and SPEC CPU2006 benchmarks. We profile only hot loops that are not parallelized by the compiler. For example, three applications from SPEC206, `bwaves`, `libquantum`, and `cactusADM`, are parallelized by the compiler, and thus excluded from the study.

We focus on four applications and provide a detailed discussion of their dependence profile characteristics and how they affect speculation profitability. A summary of the dependence characteristics obtained using **DProf** is given in

Table 1. Summary of dependence characteristics obtained using **DProf**

Benchmark	Hot Function	% exec	Profiling Summary
`lammps`	`pair_eam::compute()`	90%	parallel with array reduction
	`Neighbor::half_bin_newton()`	8%	serial due to scalar dependence
`gromacs`	`innerf.f:3932`	57%	inner loop parallel, outer loop serial
`hmmer`	`fast_algorithm.c:119`	90%	both inner/outer loops are serial
`mcf`	`implicit.c:265`	10%	both inner/outer loops are serial

Table 1. Other applications are not profiled due to a lack of hot for-loops (while-loops are not profiled due to limitation of the current implementation), or due to hot loops that are obviously non-parallel (e.g., containing I/O operations).

SEQUOIA/lammps. The hottest function, `pair_eam::compute()`, covers 90% of the execution time. There are two hot loops in this function that exhibit similar dependence patterns. Neither of them is parallelized by the compiler due to inadequate pointer aliasing information.

Figure 3 shows the fragment of the first hot loop, where the outer i-loop traverses a list of atoms and the inner k-loop traverses the neighbor list of each item. Figure 4 gives the profiling report of the outer i-loop shown in Figure 3. The loop has an average independence window of 1 iteration. The narrow independence window is due to the tight dependence caused by read-modify-write to `rho[j]` at line 191, and by the conflict between `rho[i]+=` at line 188 and `rho[j] +=` at line 191. According to these stats, if consecutive iterations of the i-loop are scheduled as TLS tasks, then the conflict rate would be almost 100%.

```
164   for (i = 0; i < nlocal; i++) {
      ...
168     itype = type[i];
169     neighs = neighbor->firstneigh[i];
170     numneigh = neighbor->numneigh[i];
171
172     for (k = 0; k < numneigh; k++) {
173       j = neighs[k];
          ...
180       if (rsq < cutforcesq) {
            ...
188           rho[i] += ((coeff[3]*p + coeff[4])*p + coeff[5])*p+coeff[6];
189           if (newton_pair || j < nlocal) {
190             coeff = rhor_spline[type2rhor[itype][jtype]][m];
191             rho[j] += ((coeff[3]*p + coeff[4])*p +coeff[5])*p+coeff[6];
192           }
193       }
194     }
195   }
```

Fig. 3. Source code of 1st hot loop in `pair_eam::compute()`

```
Loop <1> at line 164 has 101 invocations, average 32000 iter/invoc (min=32000, max=32000)
  The loop has minIndep = 1 maxIndep = 5 avgIndep = 1
  – Detected a true dependence with frequency of 43.7679% between "this–>rho[i]" (line# 188) and "this–>rho[j]" (line# 191) in "pair_eam.cpp"
  – Detected a true dependence with frequency of 51.4117% between "this–>rho[j]" (line# 191) and "this–>rho[i]" (line# 188) in "pair_eam.cpp"
  – Detected a true dependence with frequency of 95.7514% between "this–>rho[j]" (line# 191) and "this–>rho[j]" (line# 191) in "pair_eam.cpp"
```

Fig. 4. Profiling report for hot loop in `pair_eam::compute()`

Note that all the loop-carried dependences are between the statements of the form `rho[x] +=` . In other words, elements of array `rho` are reductions. This loop can be parallelized by parallelizing the reduction [19], with the caveat that the compiler analysis needs to be extended to handle different subscripts.

CPU2006/gromacs. The hottest loop in `gromacs` is in `innerf.f` at line 3932. The loop is a doubly nested and covers 57% of the total execution time. The loop nest contains many array references through subscript arrays (e.g., `faction(jjnr(k)-1)`), thus dependences on this loop nest cannot be detected statically. With the training input set, the inner loop has an average trip count of 28 iterations, ranging from 1 to 173. The loop is profiled to be parallel. Consequently, the loop is marked as profitable, with respect to conflict probability, for TLS.

The outer loop has an average trip count of 1250 iterations, ranging from 4 to 13891. **DProf** reports 30 pairs of true dependences for this loop and an independence window of 1 iteration. Half of the dependences have very low frequency (less than 1%). The rest have frequencies ranging from 18% to 100%. Figure 5 shows fragments of the loop, where all high frequency dependences occur between line 4140 and 4154 at the bottom of the outer loop.

```
3932   do n=1,nri
         ...
         ii3 = 3*iinr(n)-1
         is3 = 3*shift(n)+1
3961     do k=nj0,nj1
         ...
4139     end do
4140     faction(ii3)     = faction(ii3) + fix1 /* dep freq 51% */
4141     faction(ii3+1)   = faction(ii3+1) + fiy1 /* dep freq 51% */
         ...
4148     faction(ii3+8)   = faction(ii3+8) + fiz3 /* dep freq 51% */
4149     fshift(is3)      = fshift(is3) + fix1+fix2+fix3 /* dep freq 18% */
4150     fshift(is3+1)    = fshift(is3+1) + fiy1+fiy2+fiy3 /* dep freq 18% */
4151     fshift(is3+2)    = fshift(is3+2) + fiz1+fiz2+fiz3 /* dep freq 18% */
4152     ggid             = gid(n)+1
4153     Vc(ggid)         = Vc(ggid) + vctot /* dep freq 100% */
4154     Vnb(ggid)        = Vnb(ggid) + vnbto /* dep freq 100% */
4155   end do
```

Fig. 5. Source code of the hot loop at line 3932 in innerf.c

All high frequency dependences occur among statements with array element reduction pattern (i.e., of the form of `a[x] +=`). Of them, updates to elements of arrays `Vc`, `Vnb`, and `fshift` are true reductions. Updates to `faction` exhibit a more complex pattern. Besides the reduction updates to elements of `faction` between line 4140 and 4148, there are additional reads and writes to `faction` in the inner k-loop that are not in the reduction form; however, these references to `faction` lead to very low frequency conflict ($\tilde{1}$%) with those references to `faction` between line 4140 and 4148. This means the outer loop can not be easily parallelized by reduction handling and requires TLS support.

CPU2006/hmmer. We now illustrate the dependence profiling of the hot loop in `hmmer` taken from `fast_algorithms.c`:119. The loop covers 90% of the execution time and is shown in Figure 6. In this loop, variables `mmx, dmx, xmx` and

```
120   for (i = 1; i <= L; i++) {
121     mc    = mmx[i];
122     dc    = dmx[i];
123     ic    = imx[i];
124     mpp   = mmx[i-1];
125     dpp   = dmx[i-1];
126     ip    = imx[i-1];
127     xmb   = xmx[i-1][XMB];
        ...
134     for (k = 1; k <= M; k++) {
135       mc[k] = mpp[k-1]   + tpmm[k-1];  /*flow to mc[k-1] line 143 */
136       if ((sc = ip[k-1]  + tpim[k-1]) > mc[k])  mc[k] = sc;
137       if ((sc = dpp[k-1] + tpdm[k-1]) > mc[k])  mc[k] = sc;
138       if ((sc = xmb  + bp[k])         > mc[k])  mc[k] = sc;
139       mc[k] += ms[k];
140       if (mc[k] < -INFTY) mc[k] = -INFTY;
141
142       dc[k] = dc[k-1] + tpdd[k-1]; /* flow to dc[k-1] line 143 */
143       if ((sc = mc[k-1] + tpmd[k-1]) > dc[k]) dc[k] = sc;
144       if (dc[k] < -INFTY) dc[k] = -INFTY;
145
            ...
152     }

158     xmx[i][XMN] = -INFTY;  /* flows to xmx[i-1][XMN] at line 159 */
159     if ((sc = xmx[i-1][XMN] + hmm->xsc[XTN][LOOP]) > -INFTY)
160       xmx[i][XMN] = sc;
        ...
189   }
```

Fig. 6. Source code of the loop at `456.hmmer:fast_algorithm:119`

```
265 for( ; i < trips; i++, arcout += 3 )
266 {
267     if( arcout[1].ident != FIXED )
268     {
269         arcout->head->firstout->head->arc_tmp = first_of_sparse_list;
270         first_of_sparse_list = arcout + 1;
271     }
272
273     if( arcout->ident == FIXED )
274         continue;
275
276     head = arcout->head;
277     latest = head->time - arcout->org_cost
278         + (long)bigM_minus_min_impl_duration;
279
280     head_potential = head->potential;
281
282     arcin = first_of_sparse_list->tail->arc_tmp;
283     while( arcin )
284     {
285         tail = arcin->tail;
286
287         if( tail->time + arcin->org_cost > latest )
288         {
289             arcin = tail->arc_tmp;
290             continue;
291         }
    ...
310 }
```

Fig. 7. Source code of the loop at `429.mcf:implicit.c:265`

`imx` are declared as `int**`. Due to the lack of aliasing information, the compiler can not determine the precise dependence information for references in the loop.

With the training input set, the inner loop is profiled to have a trip count of 100 iterations, and has an independence window of 1 iteration. Two pairs of dependences are detected for this loop on the `mc` and `dc` variables. The outer loop has an average trip count of 491 iterations, ranging from 7 and 1328 iterations. The loop also has an independence window of 1 iteration. Ten pairs of dependences are detected with 100% conflict rate. This is an example of a loop that is not a good candidate for speculative parallelization.

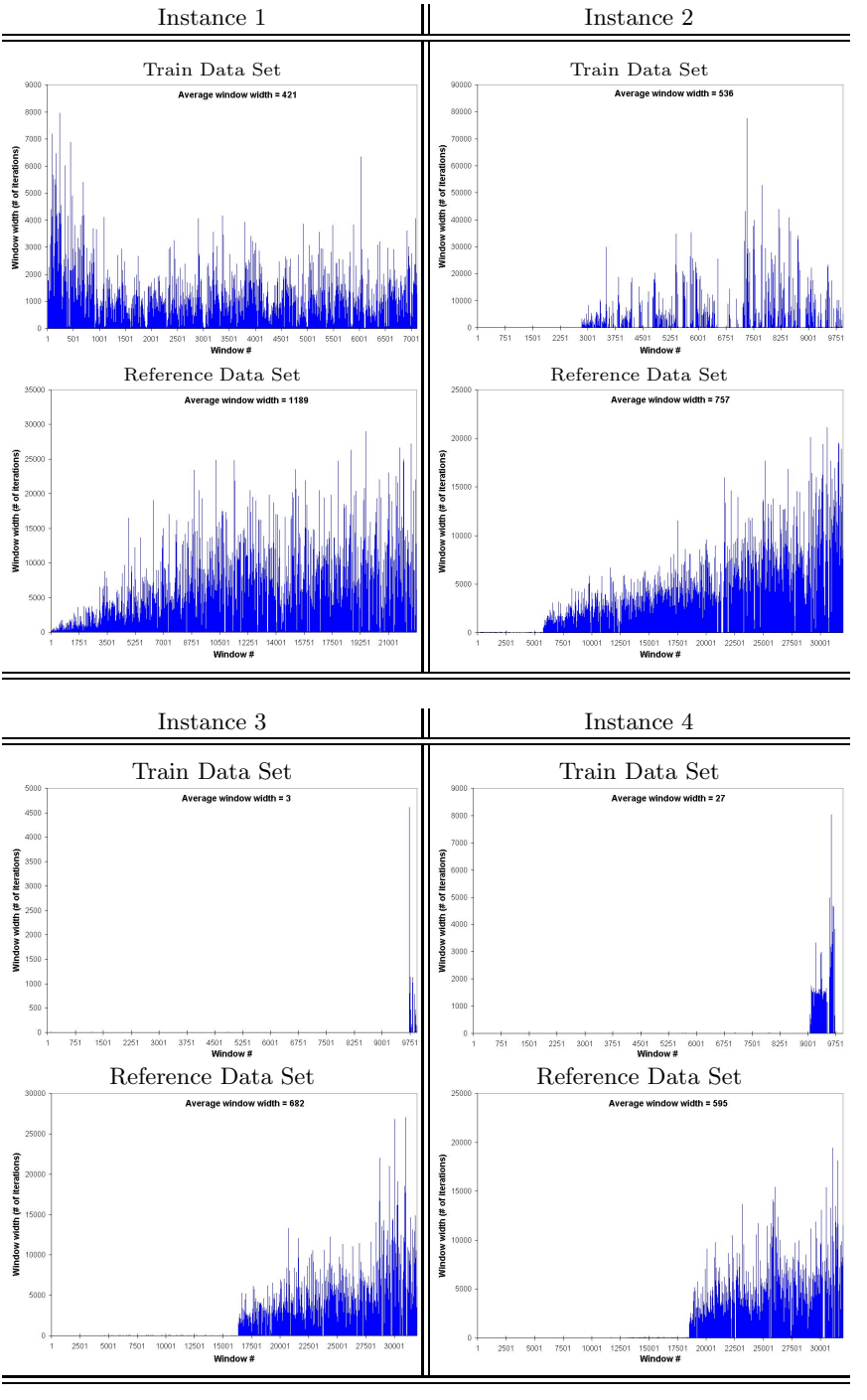


Fig. 8. Illustration of dependence clustering

CPU2006/mcf. The hottest loop in `429.mcf` (`psimplex.c:59`) is a while-loop thus is not profiled. The loop at `implicit.c:265` (see Figure 7) covers 10% of the execution time. The loop is profiled to have an independence window of 1 iteration. The profiler reports 18 pairs of dependences for this loop. Of them, two dependences have a conflict rate of 100%, one on the variable `first_of_sparse_list` at lines 269 and 270 and the other between the pointer accesses `arcout->head->firstout->head->arc_tmp` at line 270 and `tail->arc_tmp` at line 289.

4.2 Determining Independent Clusters

In this subsection, we present a case study to illustrate the use of **DProf** to determine independent clusters. Figure 8 shows the profile of independence windows for four different instances of the loop in `429.mcf`, at `implicit.c:381`, using the training and reference input data sets. All the references in the loop were instrumented, except for the variable`susp` – a reduction recognized by the compiler.

The x-axis in each subfigure of Figure 8 represents the number of independence windows and the y-axis represents the width of an independent window. From Figure 8 we see that parts of the iteration space have large independence window widths. For instance, let us consider the profile shown in the second row and right column. We note that the widths are very small towards the left of the x-axis and is more than 7500, on an average, on the right side of the x-axis. In such cases, we say that dependences occur in clusters. This behavior can be exploited for speculatively parallelization, as discussed earlier in Section 2. Interestingly, we note that the profile of independence windows varies from one instance to another and is significantly different for the training and reference data sets.

For the hot loops we studied, however, we find that the dependence window profile is quite regular: the loops are either fully parallel or have a constant independence window width of 1.

5 Previous Work

Task-level speculative computation has been long proposed as a means for extracting higher levels of parallelism [3]. With the emergence of multithreaded processors [22], many researchers have proposed the use of threads for exploiting speculative parallelism in both hardware and software [2, 10, 23, 25].

To ensure profitable speculation, most prior work use profile-based cost models for task decomposition and selection, while others rely on hardware mechanism [4, 27, 33], probabilistic static analysis (e.g., dependence probability [5]), compiler heuristics [28] for task generation. The rest of the section will focus on related work in software profile-based cost models, which our work belongs to.

In [7], the compiler uses dependence profile for task selection and for partitionning speculative loops into serial and parallel portions. The profiler tracks both intra- and carried- true dependences for speculative loops. Carried-dependences

are used to guide the parition of loop bodies into a serial and a parallel portion. Since dependences originated from the serial portion do not trigger roll-back in the parallel portion, the key part of their framework is to move source computation of frequent dependences (called *violating candidate*) to the serial portion through instruction reording. A cost model is used to select the optimal loop partition, which is based on the size of serial portion and the misspeculation cost of the parallel portion. The latter is computed by combining re-execution cost of individual nodes weighted by probabilities of carried-dependences (for violating candidates) and intra-iteration dependences (for others).

In [18], the POSH compiler uses profiling for task selection. The profiler builds a rudimental timing model for TLS execution from the sequential execution. It assigns timestamps to each instruction as if the tasks were executed in parallel, and detect task squashes by comparing timestamps of conflicting memory accesses. The profiler does not collect individual dependence probabilities thus runs much faster than typical dependence profiler. The compiler also partition the loop into serial and parallel (called *hoisting distance*) portions. But unlike [7], the partitioning uses static information only. Tasks are pruned based on three independent thresholds for task size, hoisting distance, and squash benefit. The latter also factors in prefetching benefit of squashed tasks.

In [21], the Mitosis compiler uses both dependence and edge-profiles for 1) generating speculative precomputation slices (*p-slice*) and 2) selecting spawning pairs. P-slice predicts live-in values for speculative tasks and contributes to the serial portion of the speculative execution. To minimize p-slice overhead while maximizing the accuracy, the compiler uses dependence- and edge-profiles to prune instructions in p-slices. To select spawning pairs, another profile analyzes the sequential execution trace to model the speculative execution time of each candidate spawning pair without considering inter-task memory conflicts. Instead, in this execution model, task squashes is mostly determined by mispredication probability of p-slices.

In [13], Johnson et al. proposed an approach wherein speculative task decomposition is modelled as a balanced min-cut problem. In this framework, edge- and dependence profiles are used to assign weights to graph edges.

In [30], Praun et al. proposed a tool for speculative task head recommendation based on binary instrumentation. The cost model for task recommendation is based on *self length* that models task sizes, *dependence length* that models conflicts, and a parallelization speedup estimate.

Alias profiling has been proposed as an assist for memory disambiguation [9, 16, 31]. Chen et al. [6] proposed a dependence profiler for speculative optimizations.

6 Conclusions

This paper presents a cost model for speculative task selection and a compiler-based approach for program dependence profiling. The dependence profiler, **DProf**, measures width of independence windows to quantify dynamic paral-

lelism in the program. We also propose dependence clustering as a technique to exploit TLS parallelism on segments of the iteration space.

In addition to its use in task selection, **DProf** also reports individual dependences being profiled including dependence probability and source mapping information. This information can be fed to the compiler or the programmer to assist code transformation or algorithmic optimizations.

We present the dependence and independence window profile obtained through **DProf** for selected programs in SEQUOIA and SPEC CPU2006 benchmark suites. We observe that:

- In addition to all the loops parallelized by the compiler, only one hot loop is profiled to be parallel in the programs being studied.
- There is little variability in independence window width in the hot loops we studied. Loops are either parallel or serial with an independence window width of 1.
- For loops with tight independence window, there are often a mixture of high- and low-frequency dependences. Eliminating high-frequency dependences is key to widening the independence window.
- Dependences due to complex reduction updates is one form of high-frequency dependence that can be potentially eliminated.

As future work, we plan to provide support for dependence profiling of `while` loops and enable the profiling of complex reduction patterns.

References

1. ASC Sequoia Benchmark Codes, `http://www.llnl.gov/asc/sequoia/benchmarks/`
2. Bruening, D., Devabhaktuni, S., Amarasinghe, S.: Softspec: Software-based speculative parallelism. In: Proceedings of 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3) (2000)
3. Burton, F.W.: Speculative computation, parallelism, and functional programming. IEEE Trans. Computers 34(12), 1190–1193 (1985)
4. Chen, M.K., Olukotun, K.: The Jrpm system for dynamically parallelizing Java programs. In: Proceedings of the 30th International Symposium on Computer Architecture, pp. 434–446 (2003)
5. Chen, P.-S., Hung, M.-Y., Hwang, Y.-S., Ju, R.D.-C., Lee, J.K.: Compiler support for speculative multithreading architecture with probabilistic points-to analysis. In: Proceedings of the 9th Symposium on Principles and Practice of Parallel Programming, pp. 25–36 (2003)
6. Chen, T., Dai, J.L.X., Hsu, W.-C., Yew, P.-C.: Data dependence profiling for speculative optimizations. In: Proceedings of the 13th International Conference on Compiler Construction, Barcelona, Spain, pp. 57–72 (2004)
7. Du, Z.-H., Lim, C.-C., Li, X.-F., Yang, C., Zhao, Q., Ngai, T.-F.: A cost-driven compilation framework for speculative parallelization of sequential programs. In: Proceedings of the SIGPLAN 2004 Conference on Programming Language Design and Implementation, Washington DC, USA, pp. 71–81 (2004)

8. Dubey, P., O'Brien, K., O'Brien, K., Barton, C.: Single-program speculative multithreading (SPSM) architecture. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (1995)
9. Fernández, M., Espasa, R.: Speculative alias analysis for executable code. In: Proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques, Charlottesville, VA, pp. 222–231 (2002)
10. Franklin, M., Sohi, G.S.: The expandable split window paradigm for exploiting fine-grain parallelsim. SIGARCH Comput. Archit. News 20(2), 58–67 (1992)
11. Hammond, L., Willey, M., Olukotun, K.: Data speculation support for a chip multiprocessor. In: Proceedings of 8th International Conference on Architecutral Support for Programming Languages and Operating Systems (1998)
12. Johnson, T., Eigenmann, R., Vijaykumar, T.: Speculative thread decomposition through empirical optimization. In: Proceedings of the 12th Symposium on Principles and Practice of Parallel Programming (2007)
13. Johnson, T.A., Eigenmann, R., Vijaykumar, T.N.: Min-cut program decomposition for thread-level speculation. In: Proceedings of the SIGPLAN 2004 Conference on Programming Language Design and Implementation, Washington DC, USA, pp. 59–70 (2004)
14. Kejariwal, A., Tian, X., Girkar, M., Li, W., Kozhukhov, S., Saito, H., Banerjee, U., Nicolau, A., Veidenbaum, A.V., Polychronopoulos, C.D.: Tight analysis of the performance potential of thread speculation using SPEC CPU 2006. In: Proceedings of the 12th Symposium on Principles and Practice of Parallel Programming (2007)
15. Krishnan, V., Torrellas, J.: Hardware and software support for speculative execution of sequential binaries on a chip-multiprocessor. In: Proceedings of 12th International Conference on Supercomputing (1998)
16. Lin, J., Chen, T., Hsu, W.-C., Yew, P.-C., Ju, R.D.-C., Ngai, T.-F., Chan, S.: A compiler framework for speculative analysis and optimizations. In: Proceedings of the SIGPLAN 2003 Conference on Programming Language Design and Implementation, pp. 289–299 (2003)
17. Liu, W., Tuck, J., Ceze, L., Ahn, W., Strauss, K., Renau, J., Torrellas, J.: POSH: A TLS compiler that exploits program structure. In: Proceedings of the 11th Symposium on Principles and Practice of Parallel Programming (2006)
18. Liu, W., Tuck, J., Ceze, L., Ahn, W., Strauss, K., Renau, J., Torrellas, J.: POSH: A TLS compiler that exploits program structure. In: Proceedings of the 11th Symposium on Principles and Practice of Parallel Programming, pp. 158–167 (2006)
19. Pottenger, B., Eigenmann, R.: Parallelization in the presence of generalized induction and reduction variables. In: Proceedings of 9th International Conference on Supercomputing (1995)
20. Pugh, W.: The definition of dependence distance. Technical Report CS-TR-2292 (November 1992)
21. Quiñones, C.G., Madriles, C., Sánchez, J., Marcuello, P., González, A., Tullsen, D.M.: Mitosis compiler: An infrastructure for speculative threading based on precomputation slices. In: Proceedings of the SIGPLAN 2005 Conference on Programming Language Design and Implementation, pp. 269–279 (2005)
22. Halstead, J.R.H., Fujita, T.: Masa: a multithreaded processor architecture for parallel symbolic computing. SIGARCH Comput. Archit. News 16(2), 443–451 (1988)
23. Rauchwerger, L., Padua, D.: The lrpd test: speculative run-time parallelization of loops with privatization and reduction parallelization. In: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation, pp. 218–232. ACM, New York (1995)

24. Renau, J., Tuck, J., Liu, W., Ceze, L., Strauss, K., Torrellas, J.: Tasking with out-of-order spawn in tls chip multiprocessors: Microarchitecture and compilation. In: Proceedings of 19th International Conference on Supercomputing (2005)
25. Sohi, G.S., Breach, S.E., Vijaykumar, T.N.: Multiscalar processors. In: Proceedings of International Symposium on Computer Architecture, pp. 414–425. S. Margherita Ligure, Italy (1995)
26. Steffan, J.G., Colohan, C.B., Zhai, A., Mowry, T.C.: A scalable approach to thread-level speculation. In: Proceedings of International Symposium on Computer Architecture (2000)
27. Tubella, J., González, A.: Control speculation in multithreaded processors through dynamic loop detection. In: HPCA 1998: Proceedings of the 4th International Symposium on High-Performance Computer Architecture, p. 14 (1998)
28. Vijaykumar, T., Sohi, G.S.: Task selection for a multiscalar processor. In: Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture (1998)
29. von Praun, C., Bordawekar, R., Cascaval, C.: Modeling optimistic concurrency using quantitative dependence analysis. In: Proceedings of the 13th Symposium on Principles and Practice of Parallel Programming (2008)
30. von Praun, C., Ceze, L., Cascaval, C.: Implicit parallelism with ordered transactions. In: Proceedings of the 12th Symposium on Principles and Practice of Parallel Programming (2007)
31. Wu, Y., Lee, Y.: Accurate invalidation profiling for effective data speculation on EPIC processors. In: Proceddings of the 13th International Conference on Parallel and Distributed Computing Systems, Las Vegas, NV (August 2000)
32. Zhai, A., Colohan, C.B., Steffan, J.G., Mowry, T.C.: Compiler optimization of scalar value communication between speculative threads. In: Proceedings of 10th International Conference on Architectural Support for Programming Languages and Operating Systems (2002)
33. Zilles, C.B., Sohi, G.S.: A programmable co-processor for profiling. In: Proceedings of the Seventh International Symposium on High-Performance Computer Architecture, pp. 241–254. Nuevo Leon, NM (January 2001)

gluepy: A Simple Distributed Python Programming Framework for Complex Grid Environments

Ken Hironaka, Hideo Saito, Kei Takahashi, and Kenjiro Taura

The University of Tokyo
{kenny,h_saito,kay,tau}@logos.ic.i.u-tokyo.ac.jp

Abstract. Problem-solving frameworks in large-scale and wide-area environments must handle connectivity issues (NATs and firewalls), maintain scalability with respect to connection management, accommodate dynamic processes joining/leaving at runtime, and provide simple means to tolerate communication/node failures. All of the above must be presented in a simple and flexible programming model. This paper designs and implements such a framework by minimally extending distributed object-oriented models for maximum *generality* and *flexibility*. To make parallelism manageable, we introduce an implicit serialization semantics on objects to relieve programmers from explicit synchronization, while avoiding the recursion deadlock problems from which some models based on active objects suffer. We show how this design nicely incorporate dynamically joining processes. In our implementation, participating nodes automatically construct a TCP overlay so as to address connectivity and scalability issues. We have implemented our framework, gluepy as a library for Python. For evaluation, we show on over 900 cores across 9 clusters with complex networks (involving NATs and firewalls) and process managements (involving SSH, torque, and SGE) configurations, how a simple branch-and-bound search application can be expressed simply and executed easily.

1 Introduction

Grid environment is a complex environment in which to program. Resources are large scale and distributed across multiple clusters. Connectivity among them is restricted by NAT and firewalls. Sites run different resource management software with different policies, and available resources change constantly during computation. Problem-solving environments on the Grid must be designed so the above complexities are made manageable in a simple and uniform framework that alleviates the burden of the users. Thus, the following characteristics (among others) must be addressed:

- Allow processes to simply and seamlessly communicate across sites despite the complexity of underlying networks
- Incorporate dynamically joining processes into the computation while providing means handling of node and network failures
- A programming model to handle parallelism simply and concisely

J.N. Amaral (Ed.): LCPC 2008, LNCS 5335, pp. 249–263, 2008.

A common approach has been to design a framework that *hides* these issues. Good examples are systems and frameworks for embarrassingly parallel applications [1, 2, 3] and their extensions to express dependencies among tasks [4, 5]. They normally require no programming effort for coordination. A slightly more general approach has been to design programming frameworks specific to certain application domains [6] and coordination models [7]. These frameworks are perfect when the problem at hand naturally fit their model. On the other hand however, they usually provide no or limited means for individual subtasks to communicate with each other. Thus, implementing the coordination among subtasks must frequently resort to using "out-of-band" means (e.g., file transfer, ad-hoc CGI, etc.), making the code awkward and error-prone.

Another approach, which we pursue in this paper, is to leverage a general programming language and conventional parallel/distributed programming concepts, and minimally extend them for issues in large-scale wide-area environments. Object-oriented languages are particularly suitable for this purpose, as they have a general and an accepted model of communication (i.e., remote method invocation or RMI in short). Yet existing parallel RMI-enabled frameworks [8, 9] do not sufficiently address the aforementioned issues (connectivity, scalability, and dynamic processes). Furthermore, managing parallelism and asynchroneity with distributed objects is still not trivial, and race-conditions and deadlocks are commonplace.

We have implemented gluepy, a Python framework with addressing these issues as its primary objective and with the following contributions:

- Objects have implicit serialization semantics. Parallelism is expressed via asynchronous procedure/method calls, using primitives commonly known as *futures* [10]. Yet the semantics eliminate the need for explicit synchronization code. The underlying execution model is still based on passive objects + threads familiar to many programmers. Our design thus does not suffer from the self-recursion deadlocks that some active object-based models do [9, 11].
- An object signalling mechanism that provides simple means by which asynchronous events, such as new process joins, may be handled. It is designed so as to retain the comfortable programming style of using asynchronous method invocations to manage and schedule tasks without resorting to low level event-handling loops.
- A TCP overlay network is built among participating nodes to realize scalable and seamless communication among all participating nodes. To incorporate resources reachable via SSH, it can use SSH port forwarding where specified.

Our experimental platform includes nine clusters with over 900 CPU cores. Many of them perform IP filtering to various degrees, and some only have private addresses. It also includes an almost completely confined cluster that can be reached only by first logging in to its gateway via SSH, then logging in to a cluster frontend via SSH, and finally submitting jobs via a batch scheduler. The main result of this work is a simple scripting environment that can coordinate processes spread across such complex environments. We implemented and deployed, with little effort, a combination optimization problem solver on our

platform of over 900 CPU cores. The core of the solver is an optimized sequential program written in C++ that won the 3rd Grid Plugtest Contest [12]. Python code in our framework merely schedules the underlying C++ processes and exchanges solutions found among participating processes. The glue code is very concise and has only 250 lines, the core of which is shown in Section 3.

This paper is organized as follows. We discuss existing problem solving frameworks on the Grid in Section 2. In Section 3 and 4, we present the model and the implementation of our framework. We evaluate our work in Section 5 and conclude in Section 6.

2 Related Work

2.1 Flexible Inter-process Interactions

In order to address a large range of applications on the Grid, programming frameworks need to provide a simple yet flexible means by which processes may interact. In the realm of Grid enabled bash schedulers [2, 3], users can express inter-task dependencies in a simple script file [4]. However, inter-task interaction and communication means are limited to passing intermediate files among tasks.

The master-worker model allows close interaction between the master and its workers in the form of tasks, and is an accepted paradigm for its simplicity [13]. As such, many frameworks specialize in this type of application [14, 15, 16, 17]. However, if the master and worker require frequent interaction, the assigned tasks must be artificially broken into smaller sub-tasks. Some frameworks [14,16] enable messages to be sent between the two parties, yet this often results in cluttered code.

Satin [18] and distributed-Cilk [19] are frameworks for distributed divide-and-conquer computation. However, the model's applicable problem set is relatively small, and inter-task interaction is limited to times when tasks divide and merge.

In our proposal, we argue in favor of distributed object oriented programming. Communication is made transparent in the form of RMIs, and objects may invoke upon each other. By utilizing asynchronous invocations with futures, processes may freely interact with each other while taking advantage of the simplicity of method invocations.

2.2 Managing Parallelism for Distributed Objects

Extending an exisiting object oriented programming language for distributed computing is a popular approach. With Java, Ibis RMI [8] and ProActive [9], with Python, DisPyte [20] are notable examples. In these frameworks, parallel is expressed via asynchronous RMIs. However, this often results in race-conditions, making it necessary to use mutual exclusion. The active object [9] model takes an approach such that each object has a dedicated thread for execution, yet this can easily induce deadlocks (e.g.: recursive calls).

We propose a novel synchronization semantics for objects where *at most* 1 thread can operate on an object at any given time. When a thread performs

an invocation on the object, it must first acquire its ownership. This implicitly achieves mutual exclusion. However, when the owner thread blocks in the object's context, the ownership is temporarily released, thus preventing deadlocks.

2.3 Handling Process Joins/Failures with Ease

On Grid scale computing, where resources fluctuate constantly, it is paramount that handling of process joins and leaves is made simple for the user. Satin [18], which utilizes the divide-and-conquer model, transparently re-executed lost subtasks while handling load balancing via Random Work-Stealing [21]. However, it is difficult to extend this approach beyond the divide-and-conquer model.

Master-worker frameworks like Jojo2 [14] handle worker joins/leaves via event handlers. This necessitates low-level event-driven loops with explicit mutual exclusions; thus cluttering the code and rendering it hard to understand.

We propose, in conjunction with our synchronization semantics, a signaling mechanism for each object, which unblocks an arbitrary thread blocking in the object's context. Thus it is possible to handle asynchronous events such as node joins while adhering to our implicit synchronization semantics. Additionally, process failures are abstracted as exceptions to method invocations. This widely accepted semantics nicely integrates failures into conventional programming.

2.4 Resolving Connectivity on the Grid

Realizing communication on the Grid, where connectivity is limited by NATs and firewalls, in a scalable manner is a difficult task. PadicoTM [22] enables distributed computing using CORBA on various network hardware, yet does not take connectivity issues into account. ProActive [9] requires users to hand write descriptor files that specify connectable points, yet this is a high burden on the user. SmartSockets [23] attempts to establish connectivity on the Grid by transparently attempting various methods to establish TCP connection(s). However, none of the above take connection scalability into account, which becomes increasingly important with hundreds of coordinating processes.

We propose to construct an overlay network over which communication is routed transparently. Each processes establishes a small number of connections, and address the connection scalability issue. Connectivity is achieved by a high probability via our overlay construction scheme.

3 The Programming Model

In this section, we present a distributed object-oriented framework that operates in a NAT/firewall-prone environment with dynamically joining/leaving nodes. To the user, we provide a seamless view of the underlying environment, and present a set of simple interfaces by which nodes may communicate via RMIs, new nodes may join, and failing nodes are detected.

Like other distributed object-oriented frameworks, our model provides remote objects and communication among them are abstracted in the form of RMIs [8,9].

However, we address a number of topics important to Grid-enabled programming that were not, or only partially discussed before.

3.1 Synchronization and Asynchronous Event Handling

In the context of parallel computing, parallel and asynchronous RMIs are crucial. Yet, manipulating asynchronous RMIs is still a non-trivial task. Some implementations allow the users to define callbacks for when the results are available. However, this requires using locks to handle critical data. To resolve this issue, there are future primitives that allow the invoking thread to block for results when they become necessary; the invoking thread may perform other computation in the mean time. Its advantage is that a single thread is in control of the entire flow, and the transition from sequential programming is natural.

This does not resolve the issue on the RMI handler's standpoint. Since a remote object may receive an incoming invocation handled by an independent thread at any time, the programmer must use locks for objects that *might* receive incoming RMIs. To resolve this problem, some models [9] have implemented *active objects* where there is a dedicated thread for each object. The dedicated thread handles all incoming RMIs sequentially. However, this model easily creates deadlocks when an RMI handler also is an RMI invoker.

Yet another issue arises when the program has to handle asynchronous events, such as new node joins. The RMI-based model alone cannot handle these events. One possible approach is to handle RMI callbacks, node failures, and node joins all in one single event driven loop, like in many other master-worker frameworks. The obvious advantage is that a single control thread does all operations, eliminating locks and conditional variables. However the programmer must take care so that event handlers do not block, and the natural flow derived from sequential programming is completely lost.

We summarize the qualities favorable in a distributed object-oriented model.

- provides future primitives for expressing parallelism
- allows objects to be accessed mutually exclusively without explicit locks
- avoids unpleasant deadlocks induced by implicit serialization semantics above
- may handle asynchronous events without low-level event handling models

In the proposed model, at most one thread may run on an object at a time; the thread implicitly acquires the object's ownership for the duration. However, if this thread blocks while in the scope of a method, it temporarily releases the ownership, and another pending thread is permitted to run. When there are more than one pending thread, an arbitrary thread is scheduled, and acquires the ownership. We supply future primitives by which threads may block for results. Finally, we provide a signaling mechanism for each object, by which a thread blocking on future primitives in the object's context is made to unblock.

When an asynchronous RMI is performed, the invocation returns immediately with a future object. To do an RMI `fib` on an object `foo`, do the following.

```
future = foo.fib.future(args)
```

In order to obtain the results for the asynchronous RMI, we do

```
result = future.get()
```

If the results are not available, the call will block until they are. For scheduling, it is also very common that one waits for an array of future objects simultaneously, until any result becomes ready. To this end, we provide a global `wait` primitive that takes a list of future objects, blocks until at least one of the futures' results are available, and returns a list of futures whose results are available.

```
ready = wait(futures)
```

Finally, each object is provided with a signaling primitive invoked by

```
obj.signal()
```

This forcefully unblocks a thread that is blocking on `wait` in the object's context. If no threads are blocking at that time, the *next* thread that calls on `wait` in its context will be woken. The woken thread will contest for and reaquire the object ownership, after which the unblocked `wait` primitive will return None.

This serialization semantics eliminates the need for explicit locking. This is similar to that of active objects, adopted in some object-based languages [9]. Active objects, however, easily lead to an unpleasant behavior called self-recursion deadlocks. In contrast, our model allows another thread to run on an object when the current thread blocks in the midst of a method execution (a synchronous RMI, call on `wait`, or `future.get`). This property prevents deadlocks due to such recursive calls. With respect to atomicity, a thread is guaranteed to have exclusive control in between potentially blocking operations. Thus, the object's state may be modified without worrying about races.

Furthermore, a special method `signal` allows to unblock a thread currently waiting on the object. This is similar to the semantics of UNIX signals, which unblocks threads blocking on some I/O system calls (e.g., read). This can be used to wake a blocking thread for handling of some event(s).

3.2 Failure Semantics and Bootstrapping Nodes

For RMI failures and object lookups, we utilize the semantics widely accepted in existing RMI frameworks. Node failures are commonplace on the Grid, and simple means for handling them must be presented to the programmer. To this end, node failures are abstracted as exceptions for RMIs to objects on failed nodes. The user may catch such exceptions for failure handling. Aside from uncaught application-level exceptions in an RMI, when any of the below failures occur during an RMI, a `RemoteException` is raised on the caller side. In all cases, they may be handled by the invoker to perform recovery or evasive measures in the regular Python semantics.

1. The communication route between the caller and the callee nodes is broken
2. The callee node fails during execution

Another crucial issue for Grid-computing with dynamically joining nodes is bootstrapping a node. In distributed object-oriented models, this translates to obtaining the *first reference to a remote object*. In our framework, any remote object may be *published* with a human readable string name as follows:

```
object.register("any_name")
```

A process may obtain a reference to a published object using the name.

```
ref = RemoteRef("any_name")
```

This way, the newly joining node may obtain the reference to an existing object and thus join the computation by notifying existing members of its joining.

3.3 Sample Code

Using our programming model, one can easily implement applications using dynamic processes. We show one of such examples in Figure 1(a), a simple yet complete template for master-worker applications. The master initiates work on each worker, and results are collected using futures. The code can accommodate node joins using the `signal` primitive. Node failures are handled by catching exceptions. It is noteworthy that locks are not necessary, and that by using futures,

```
class Master:
  def __init__(self, jobs):
    self.nodes = []
    self.jobs = jobs

  def nodeJoin(self, node):
    self.nodes.append(node)
    self.signal() # notify join

  def run(self):
    assigned = {}
    while True:
      # dispatch work to available workers
      while len(self.nodes)>0
            and len(self.jobs)>0:
        node = self.nodes.pop()
        job = self.jobs.pop()
        # asynchrous RMI to worker
        f = node.work.future(job)
        assigned[f] = (node, job)

      # wait for any results
      readys = wait(assigned.keys())

      # if got signal, loop back
      if readys == None: continue

      #read ready results
      for f in readys:
        node, job = assigned.pop(f)
        try:
          print "done:", f.get()
          self.nodes.append(node)
        except RemoteException, e:
          # in case of a fault, rerun job
          self.jobs.append(job)

class Worker:
  def work(self, job):
    #do work on job...
    return results

  def run(self, mastername):
    #obtain reference to master and join
    master = RemoteRef(mastername)
    master.nodeJoin(self)
```

a. The core for a simple Master-Worker Program. Classes for the Master and the Worker are shown.

```
class Master:
  def __init__(self, jobs):
      self.jobs = jobs
      self.workers = []
      self.tabs = {}
      self.lock = Lock() # for mutex

  #invoked on new event with arg. e
  def handleEvent(self, e):
      #need mutual exclusion
      self.lock.acquire()
      try:
        #give job to new node
        if e.type == NEW_NODE:
            node = e.node
            #give new job, if any
            if len(self.jobs) > 0:
                job = self.jobs.pop()
                self.tabs[node] = job
                self.giveJob(node, job)
            else:
                self.workers.append(node)

        #handle result and give-out a new job
        elif e.type == JOB_DONE:
            print "done:", e.result
            node = e.node
            #give new job, if any
            if len(self.jobs) > 0:
                job = self.jobs.pop()
                self.tabs[node] = job
                self.giveJob(node, job)
            else:
                self.workers.append(node)

        #re-enque lost job on node failure
        #do not re-enqueue worker
        elif e.type == FAILURE:
            node = e.node
            job = self.tabs.pop(node)
            self.jobs.append(job)
      finally:
        self.lock.release()
```

b. Master-Program template in a typical master-worker framework

Fig. 1. Sample Code and Comparison with typical master-worker framework

the master's flow resembles that of a sequential program. With the active object model, where locks are also unnecessary, maintaining this flow is impossible as the master object is in method `run` the entire time; workers will never have a chance to run `nodeJoin`. In comparsion, Figure 1(b) shows the code for a typical master-worker framework. A handler, independently invoked on each event, has to branch of each event type, and the loop must perform explicit mutual exclusion. The event-driven code also destroys natural sequential flow of control.

3.4 Discussion

In our object semantics, atomic blocks do not encompass an entire method block, but rather between potentially blocking operations within a method. Yet we believe this is acceptable semantics. Atomic sections in real life applications are very short (e.g., checking if a given value exists in a map before insertion). Moreover, users are aware of blocking operations in advance (synchronous RMIs, access to futures, calling `wait`). Also, as common practice, it is not favorable to design atomic blocks such that they encompass blocking operations.

In the semantics however, livelocks may still occur, like in cases where a thread infinitely loops in a method without blocking. This is arguably as hard to debug as deadlocks. Currently, we defer this as future work.

Our signal mechanism sends the signal to *objects* rather than to *threads*. This design decision was motivated by the fact that in a distributed non-active object model, threads are ephemeral existences only used to gain parallelism. Thus, the programmer is more concerned about interacting with objects, than threads.

Finally, our model can address a wide range of applications beyond the master-worker model shown in the sample code. For example, more P2P-like applications like distributed island-GA applications, where each node performs GA and periodically do crossovers with other peer nodes, can also be easily expressed using RMIs to each other's object.

4 Implementation

In the following section, we will discuss how we implemented our framework to cope with the physical issues of a Grid environment. In particular, we had to resolve three issues: point-to-point communication in a WAN setting (NATs, firewalls), allowing nodes to join with ease, tolerating abrupt node failures.

4.1 Overlay Network Construction

In our framework, to realize point-to-point communication among all nodes, we automatically construct an overlay using TCP connections. Each participating node establishes connections with a small number of nodes chosen at random (about 10 connections). Analysis has shown that such a scheme will create a connected graph of all participating nodes with high probability [24]. However, some cluster completely filter incoming and outgoing packets, and thus there are no

means by which these resources may be connected. For these exceptional cases, we automatically perform SSH portforwarding over which TCP connections are forcefully established. Only for these cases, we require the user to specify the points between which SSH portforwarding is done. However, we view this as a very rare case and only requires one line in a configuration file.

Over this overlay, we implement a routing layer adapted from a reactive routing protocol, AODV [25]. It adapts well to dynamic graph changes and fits our setting where nodes join and leave at will; the lazy routing path construction does not entail broadcast storms in face of high churn. The routing metric is the RTT latency for each TCP link.

4.2 Dynamic Node Join

For nodes to join the computation, it must first become connected with the overlay. In order to do so, it needs a set of bootstrap peer node information, or *endpoints*, with which it will first connect. In a TCP overlay, this entails obtaining a set of initial (IP, port) pairs. We implemented an *endpoint server* that all nodes access before joining. Each node obtains a set of endpoints. It then adds its *own* endpoint to the server so that other nodes may connect to itself. We provide a number of options for this server. One is an HTTP server. The other is a server built on top of GXP[1] [26], a Grid shell. Using GXP, one may log into hundreds of remote servers via SSH and execute commands on them in parallel. It also provides a mechanism with which all nodes may communicate via SSH tunnels. Because all communication is done via SSH, this mechanism can be used even for resources that are not accessible by any other means.

4.3 RMI Fault Detection

In our context of an overlay network, a communication route between 2 points may constitute more than 1 TCP connection, and thus is not trivial to detect RMI faults. We assume that when a node fails, it closes all established TCP connections. For our implementation, an RMI is realized by two protocol messages, the RMI Request and the RMI Return message. Additionally, each RMI is identified by a globally unique id, an *RMIID*. In the implementation, we define an RMI to have failed if *either the RMI handler node fails, or if any of the TCP connections that the RMI Request message has traversed fails.*

On method invocation, an RMI Request Message is sent towards the object hosting node. As a node forwards the message, it creates a *path pointer* for the RMIID along the connection to the forwarded node.

After the method invocation has been handled by the object hosting node, an RMI Return Message is returned towards the invoker. The message is forwarded along the path on which the RMI Request message came. As it follows the path in reverse, all intermediate nodes erase the path pointer for the RMIID.

When a node fails, all nodes connected to it will detect a connection failure. Each node finds out if any RMI path pointer exists along the failed connection.

[1] http://www.logos.ic.i.u-tokyo.ac.jp/gxp/

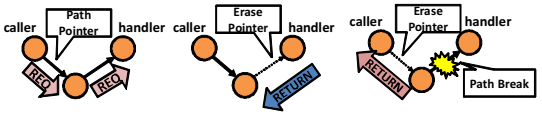


Fig. 2. The left and right figures show how an RMI Request and Return messages are sent along connections. The center figure depicts what would happen if an intermediate connection is lost.

If such a pointer exists, the node deletes the pointer and sends an RMI Return message carrying a failure exception, back along the stored path. By doing so, the intermediate node can notify the RMI invoker of the RMI failure, and the path created by the RMI Request message is effectively torn down. An image of the entire process is shown in Figure 2.

5 Evaluation

By using the program shown in Figure 1(a) as the base, we have implemented a number of master-worker type applications. In this section, we provide some micro-benchmarks to evaluate our overlay performance. We also evaluate its ability and effectiveness to run in a Grid environment with dynamic resources fluctuations. First, we explain the setup of our experimental environment. In Figure 3, we show the clusters used for our evaluations. Most cluster nodes are Core2Duo 2.13GHz, except for `kototoi` and `mirai` and `hiro` (Xeon 2.33GHz), `istbs` (Xeon 2.4GHz), and `tsubame` (Dual Core Opteron 2.4GHz). It is noteworthy that each cluster has different network administration settings. For example, due to the NAT configuration, most nodes in cluster `kyoto` and `imade` are not accessible from outside. Clusters `kototoi` have global IPs, yet due to the firewall at the gateway router, no incoming connections can be accepted. However, we were able to utilize all nodes in all clusters without any manual configuration; our bootstrapping scheme in Section 4.2 automatically bootstrapped all nodes to the peer-to-peer overlay. Exceptions are cluster `istbs` and `tsubame`, in which all its incoming and outgoing packets on most ports are filtered for security reasons. In order to utilize the two clusters, we enabled configurations for ssh forwarding from one gateway node in each cluster to a node in cluster `hongo`; all other nodes in the cluster connected to the gateway node within its cluster.

5.1 Micro-benchmark

We present our framework's microbenchmarks on our overlay, in particular, its latency overhead and its performance with data-intensive operations. We utilize clusters `chiba`, `hongo`, `kototoi`, `hiro`, `imade`, `kyoto`, and `mirai`. From a node in cluster `chiba`, we made RMIs on objects located on each of the other nodes. We show the latency of a `ping()` method, a no-op, of selected nodes from each cluster (denoted by `clusterXXX`) in Figure 4(a). The actual RTT value is paired to show the ideal minimum. Most nodes were reached within 3 hops on the overlay. The

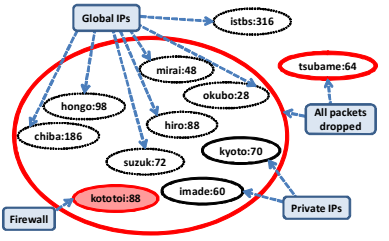
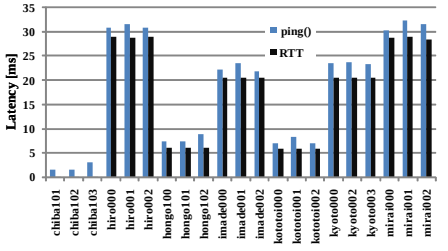
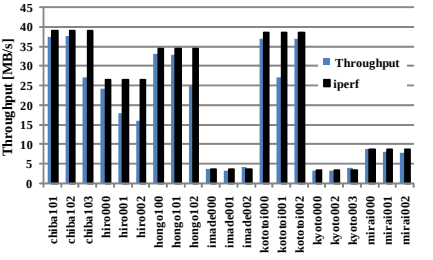


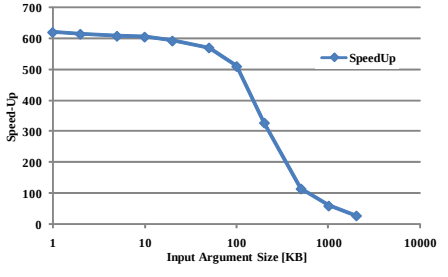
Fig. 3. Experiment cluster settings denoted by core counts



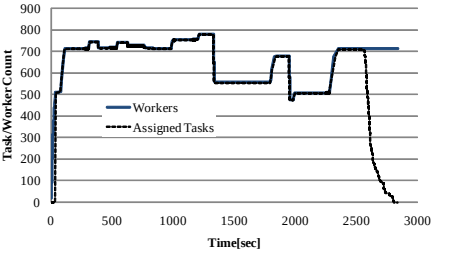
a. Latency of `ping()`



b. Throughput for `send_data()` for 100MB string



c. Speedup for parallel `send_and_wait()` for varying argument size



d. Active workers/assigned nodes for FT master-worker

Fig. 4. Benchmark Results

intra-cluster latency for cluster `chiba` was ˜150[us]. The hop latency overhead amounts to roughly 1.5[ms], sufficiently small for inter-cluster communication.

We do the same operation using the `send_data()` method, which is a no-op that takes 1 argument, and see the throughput of the data transfer. As its argument, we pass a long string of 100[MB]. The throughput is computed from the method completion time and is shown in Figure 4(b). The arguments have to be serialized (throughput: 78[MB/s]) for communication, and this reduces the maximum throughput for 1Gbit Ethernet links (overlapping serialization and sending would prevent the maximum throughput to drop to 40[MB/s], this remains to be our future work). The maximum possible point to point throughput, that accounts for serialization, calculated from iperf is paired to show the

ideal maximum. For nodes where 1Gbit Ethernet is available, a value close to this is obtained. `imade`, `kyoto`, `mirai` have gateway switches of only 100Mbit. `imade` and `kyoto` have particularly anemic bandwidth where even iperf registers 3.5[MB/s]. Within the same cluster, some nodes have much lower throughput (e.g., chiba103, hongo102, kototoi001). This is because these nodes take multiple hops on the overlay, and the store-and-forward routing diminishes the throughput on each hop.

We submit 10,000 invocations of a `send_and_wait()` method, same as `send_data()` but additionally sleeps 1[s], in a master-worker style to see the parallelism throughput when the argument size varies. This measures our framework's tolerance to parallelly handling jobs with large input data. We show the speed-up for 710 cores in Figure 4(c). The speed-up drops dramatically from around 50[KB]. This is expected as the master's maximum bandwidth(40[MB/s]) becomes saturated from the size of 56[KB] with 710 workers.

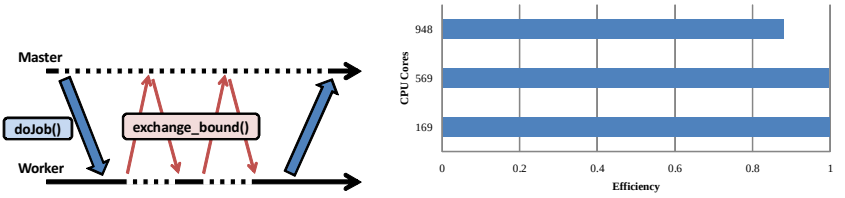
5.2 Fault-Tolerance

We evaluated our framework's ability to handle dynamic node insertions and failures. A single Master object dynamically distributes 10,000 tasks to Worker objects. For node addition, we used the method described in Section 4.2. For node failures, we simply killed the processes on nodes abruptly without warning. No tasks were lost during this experiment. We give the time series for the number of Workers running, and the number of tasks allocated by the Master to each Worker in Figure 4(d). The figure shows that as the number of workers fluctuates, the master schedules the tasks accordingly. The failure detection latency was in the order of milliseconds. All fault detection is done at the programmer level by detecting faults as exceptions as in Figure 1(a). Moreover, it is noteworthy that all participating nodes are interconnected on a TCP overlay and direct connection is not necessary for fault-detection due to the scheme in Section 4.3.

5.3 Real-Life Application

Many real-life distributed applications require parallel tasks to interact with each other. An example of such applications is a problem solver that uses branch-and-bound. In these applications, it is imperative that all parallel solvers share the latest bound information for efficient computation; these are applications that require periodic communication among nodes. Such applications are impossible to express in programmingless frameworks where inter-task communication is not permitted, or in divide-and-conquer type frameworks where communication is limited to immediate parent and children tasks. As discussed in [13], such applications can be expressed naturally in master-worker models, but there are virtually no frameworks that can handle the hostile network environment (NATs, firewalls, and IP filtering) in our experimental settings.

As our case study, we have taken on one such problem, the Permutation Flowshop Scheduling Problem. (P-FSP). P-FSP is a problem where n jobs have to be processed on m machines in the same order. This problem entails finding a



a. Master-worker RMI interaction b. Execution efficiency

Fig. 5. Evaluation of the Permutation Flowshop Solver

schedule which minimizes the makespan (execution time) with proof. The solver does parallel branch and bound in a master-worker model, where each worker receives a small section of the search space.

When a worker first joins, it first receives a task to solve. The worker and the master periodically (every 60 seconds) exchange bound information used for the branch-and-bound. When a worker finishes or aborts its given task, the master gives it its next task. The flow is expressed in Figure 5(a).

Because the computation would take months(perhaps years), fault-tolerance was a crucial part of the design. The master and the worker programs, which won the 3rd Grid Plugtest Competition [12], had already been implemented in C++. We used our framework to serve as the glue to integrate the two and deploy it on our platform. The code in Python took merely about 250 lines.

We ran the program on three different configurations: 168, 569, and 948 cores. The only necessary network configuration, in the 948 core case, was to specify the SSH portforwarding settings for clusters istbs and tsubame, which took a mere 6 lines. The rest of the deployment was taken care of automatically and successfully created a connected graph of all processes.

We present an evaluation using a relatively small randomly generated problem instance of ($n = 28$, $m = 20$). To measure the performance of our framework, rather than of the algorithm, we calculated the computation efficiency as $\frac{C}{NT}$, where N, T, and C resepctively denote the core count, the completion time, and the cummuative computation time across all cores. The results are shown in Figure 5(b). With 948 cores across 9 sites, 88% efficiency is maintained.

6 Conclusion

We have presented a programming framework that aims at simple and flexible programming in a Grid environment with limited network connectivity, dynamic node joins, and node failures. We provide simplicity, without the loss of generality, by extending a widely accepted object-oriented language, Python for wide-area parallel computing. Parallelism is expressed in the form of RMIs with the aid of futures for a natural transition from sequential programs. Accesses to objects are implicitly serialized without the fear of deadlocks, effectively eliminating locks. We provide simple means to add nodes to ongoing computation,

as well as to tolerate node failures without jeopardizing the computation. We automatically construct a TCP overlay and realize transparent communication among nodes even in the face of NATs and firewalls.

Taking a branch-and-bound optimization application as an example, we showed that our framework enables quick and effective development of parallel applications in large Grid environments with 900 cores, despite network hindrances like NATs and firewalls. A prototype for gluepy is currently available from its homepage: http://www.logos.ic.i.u-tokyo.ac.jp/~kenny/gluepy.

Acknowledgements

This research was partially funded by "New IT Infrastructure for the Information-explosion Era" of the MEXT Grant-in-Aid for Scientific Research on Priority Areas.

References

1. OpenPBS, `http://www-unix.mcs.anl.gov/openpbs/`
2. Ayyub, S., Abramson, D., Enticott, C., Garic, S., Tan, J.: Executing Large Parameter Sweep Applications on a Multi-VO Testbed. In: CCGRID 2007: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid, pp. 73–82. IEEE Computer Society, Los Alamitos (2007)
3. Thain, D., Tannenbaum, T., Livny, M.: Distributed computing in practice: the Condor experience. Concurrency - Practice and Experience 17(2-4), 323–356 (2005)
4. DAGMan, `http://www.cs.wisc.edu/condor/dagman/`
5. Taverna Project Website, `http://taverna.sourceforge.net/`
6. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: OSDI, pp. 137–150 (2004)
7. Egner, M.T., Lorch, M., Biddle, E.: UIMA Grid: Distributed Large-scale Text Analysis. In: CCGRID 2007: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid, pp. 317–326. IEEE Computer Society, Los Alamitos (2007)
8. van Nieuwpoort, R.V., Maassen, J., Wrzesinska, G., Hofman, R., Jacobs, C., Kielmann, T., Bal, H.E.: Ibis: a Flexible and Efficient Java-based Grid Programming Environment. Concurrency and Computation: Practice and Experience 17(7-8), 1079–1107 (2005)
9. Huet, F., Caromel, D., Bal, H.E.: A High Performance Java Middleware with a Real Application. In: Proceedings of the Supercomputing conference (2004)
10. Taura, K., Matsuoka, S., Yonezawa, A.: ABCL/f: A Future-Based Polymorphic Typed Concurrent Object-Oriented Language: Its Design and Implementation (1994)
11. Agha, G.: Actors: a Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge (1986)
12. 3rd Grid Plugtest Report, `http://www-sop.inria.fr/oasis/plugtest2006/plugtests_report_2006.pdf`
13. Aida, K., Osumi, T.: A Case Study in Running a Parallel Branch and Bound Application on the Grid. In: SAINT 2005: Proceedings of the The 2005 Symposium on Applications and the Internet, pp. 164–173. IEEE Computer Society, Los Alamitos (2005)

14. Aoki, H., Nakada, H., Tanaka, K., Matsuoka, S.: A Programming Environment with Dynamic Node Configuration for Hierarchical Grid: Jojo2. In: SACSIS (2006)
15. Linderoth, J., Goux, J.P., Yoder, M.: Metacomputing and the Master-Worker Paradigm. Technical Report ANL/MCS-P792-0200, Mathematics and Computer Science Division, Argonne National Laboratory (February 2000)
16. Nakada, H., Matsuoka, S.: A Java-based programming environment for hierarchical Grid: Jojo. In: CCGRID, pp. 51–58 (2004)
17. Shudo, K., Tanaka, Y., Sekiguchi, S.: P3: P2P-based Middleware Enabling Transfer and Aggregation of Computational Resources. In: CCGRID 2005: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid, vol. 1, pp. 259–266. IEEE Computer Society, Los Alamitos (2005)
18. Wrzesinska, G., van Nieuwpoort, R.V., Maassen, J., Kielmann, T., Bal, H.E.: Fault-tolerant Scheduling of Fine-grained Tasks in Grid Environments. International Journal of High Performance Computing Applications (IJHPCA) 20(1) (2006)
19. Frigo, M., Leiserson, C.E., Randall, K.H.: The Implementation of the Cilk-5 Multithreaded Language. In: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, pp. 212–223 (June 1998); Proceedings published ACM SIGPLAN Notices, 33(5) (May 1998)
20. Fuhner, T., Popp, S., Jung, T.: A novel framework for distributing computations dispyte distributing python tasks environment. Journal of Computational Electronics 5(4), 349–352 (2006)
21. van Nieuwpoort, R.V., Kielmann, T., Bal, H.E.: Efficient Load Balancing for Wide-area Divide-and-conquer Applications. In: PPoPP 2001: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming, pp. 34–43. ACM, New York (2001)
22. Denis, A., Pérez, C., Priol, T.: Padicotm: an open integration framework for communication middleware and runtimes. Future Gener. Comput. Syst. 19(4), 575–585 (2003)
23. Maassen, J., Bal, H.E.: Smartsockets: Solving the Connectivity Problems in Grid Computing. In: HPDC 2007: Proceedings of the 16th international symposium on High performance distributed computing, pp. 1–10. ACM, New York (2007)
24. Horita, Y., Taura, K., Chikayama, T.: A Scalable and Efficient Self-Organizing Failure Detector for Grid Applications. In: GRID 2005: Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing, pp. 202–210. IEEE Computer Society, Los Alamitos (2005)
25. Perkins, C.: Ad-hoc On-demand Distance Vector Routing. In: MILCOM 1997 panel on Ad Hoc Networks (November 1997)
26. Taura, K.: GXP: An Interactive Shell for the Grid Environment. IWIA 00, 59–67 (2004)

A Fully Parallel LISP2 Compactor with Preservation of the Sliding Properties

Xiao-Feng Li, Ligang Wang, and Chen Yang

Managed Runtime Optimization, Intel China Research Center, Beijing, China
{xiao.feng.li,ligang.wang,chen.yang}@intel.com

Abstract. Compacting garbage collector (GC) is widely used due to its good properties of in-place collection and heap de-fragmentation. In addition, it supports fast bump-pointer allocation and provides good access locality. Most known commercial JVM or CLR implementations use compaction algorithm in certain garbage collection scenarios, such as in full heap or mature object space collections. LISP2 compactor is one of the best-known GC algorithms. As multi-core architecture prevails, several efficient parallel compactors have been proposed. Nevertheless, there is no parallel LISP2 compactor available that can preserve all the sliding properties of its sequential counterpart. That is, to compact live data in-place into a single contiguous region in one end of the heap while maintaining the original object order. In this paper, we propose a fully parallel LISP2 compactor that keeps all the sliding properties. We also prove the correctness of the design. This parallel LISP2 compactor is fully parallel because all of its four phases are parallelized and the workloads are well balanced among the collector threads. The compactor supports fall-back compaction and adjustable boundaries that help deliver the best performance. We have implemented the parallel LISP2 compactor in Apache Harmony, a product-quality open source Java SE implementation. We evaluate and discuss the design on an Intel 8-core platform with representative benchmark.

Keywords: Garbage collector, compactor, parallelization.

1 Introduction

Garbage collection is a key component in managed runtime systems such as the runtime engines of Java, C# and scripting languages. In current known commercial JVM and CLR implementations, compacting garbage collector is unavoidably utilized in certain scenarios because of its advantages. For example, compacting GC reduces the heap fragmentation by packing data together while eliminating the unusable areas in between. This improves both heap space utilization and data locality. By leaving the free space contiguous, compacting GC also allows fast bump-pointer allocation. Finally, compacting GC can preserve the original object order in the heap as before the compaction, which is believed to have the best memory access locality.

The LISP2 compactor [3][6] has additional benefits. It does not rely on underlying OS virtual memory support, and it compacts the heap in-place without requiring significant extra space for collection, such as the block offset table used in some other

J.N. Amaral (Ed.): LCPC 2008, LNCS 5335, pp. 264–278, 2008.

compactors. One special benefit that is nonexistent in other compactors is that, the LISP2 compactor compacts the heap in the granularity of individual object, which provides good chances for individual object manipulations on the fly, such as to add or remove some data fields of the interested objects.

Although there have been several parallel compactors proposed [4][1][8][13], only one [4] tried to parallelize the LISP2 compactor, which partitions the heap into multiple regions, so that multiple GC threads (called collectors) can collect them independently in parallel. The problem with that compactor is that, it cannot compact the live data into a single contiguous region at one end of the heap, but leaves multiple object groups, one for every two neighboring partitions. This is a huge drawback to the original LISP2 compactor. Moreover, in that compactor, how to partition the heap pre-determines the available parallelism. The number of partitions strictly decides how many collectors can work in parallel, and the live data amount in a partition decides the work load of the collector assigned to that partition. The overall loads of the collectors are not dynamically balanced.

In this paper, we propose a fully parallel LISP2 compactor, all of whose phases are parallelized with balanced loads among the collectors. Furthermore this parallel compactor preserves all the good properties of LISP2 compactor.

1.1 Overview of LISP2 Compactor

The core algorithm of the LISP2 compactor consists of following phases for a collection:

Phase 1: *Live object marking.* This phase traces the heap from root set and marks all the live objects;

Phase 2: *Object relocating.* This phase computes the new address of every live object, and installs this value into the object header;

Phase 3: *Reference fixing.* This phase adjusts all the reference values in the live objects to point to the referenced objects' new locations;

Phase 4: *Object moving.* This phase copies the live objects to their new locations.

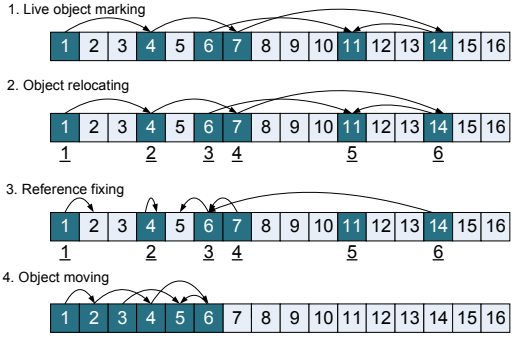


Fig. 1. Phases of LISP2 compactor (*An object is represented as a cell, and a live object is in dark color. Object reference is represented as an arrow pointing from the containing object to the referenced object. The numbers are the addresses of the objects, and the underscored numbers refer to the new target addresses of the objects.*)

Fig. 1 illustrates the phases of the LISP2 compactor[1].

In our experiments with the typical benchmarks, the four phases take similar execution time. That means, all of them should be parallelized in order to achieve good performance. In our design, we parallelize the four phases in four different ways according to the phase behavior while preserving the sliding properties.

The rest of the paper is organized as follows. Section 2 gives an overview of the parallel LISP2 compactor. Then we describe the design in details in Section 3. Section 4 introduces how we apply the parallel compactor into a real JVM to support adjustable boundaries. Section 5 evaluates and discusses the design with SPECJBB2005 benchmark. We discuss the related work in Section 5, and summarize the work in Section 6.

2 Design of Parallel LISP2 Compactor

In this section, we give an overview of the parallel LISP2 compactor design. We discuss the design phase by phase.

2.1 Live Object Marking

The phase of *live object marking* is to traverse the object connection graph. The parallelism granularity is naturally a node in the graph, i.e., an object. Although the parallelization properties for this phase have been studied by GC community for years, there are still a couple of design decisions to make for a GC algorithm.

Firstly, we need to decide the representation of the marking status of an object. A separate mark bit table requires atomic operations for marking, while marking the object header requires scanning the heap to find the marked objects. We choose to mark the object header due to its low overhead compared to that of atomic operations.

We also need to balance the loads of the marking tasks among multiple collectors. The idea is to assign the marking tasks evenly to the collectors at runtime, thus achieving dynamic load balance. After comparing the techniques of pool sharing, work stealing and task-pushing [14], we adopt the task-pushing technique because it can avoid atomic operations. Task-pushing composes a data-flow network between multiple collectors through task queues, as shown in Fig. 2.

Last thing to decide for parallel marking is the traversal order in the object connection graph. Since the live objects spread across the heap, it is possible that one traversal order has better access locality than another. Our experience is that the depth-first order has the best locality.

For the parallel LISP2 compactor, the live object marking phase traces the object connection graph in depth-fist order and marks object header with task-pushing load balance mechanism. Since the marking phase is common and studied in many different GC algorithms, we will not discuss it in the following text, but focus on other phases.

[1] Actually in our implementation there is an extra final phase that restores the object header information, which was replaced by a forwarding pointer in the object relocating phase. This phase takes negligible time compared to other phases and is not necessarily inherent to the LISP2 compactor algorithm. So we do not discuss it in this paper.

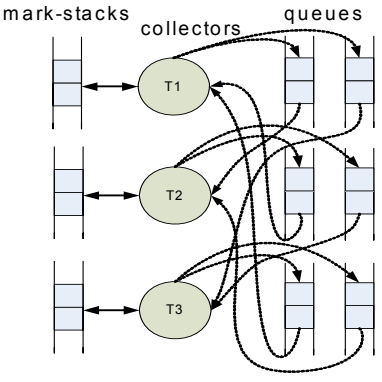


Fig. 2. Task-pushing load balance (*The dotted directed lines represent how the collectors push and pull the marking tasks through the queues. Normally the queues length is just one, which is virtually a shared variable between two collectors.*)

2.2 Object Relocating

This phase computes the objects' target locations, without really moving the objects. Since the new addresses decide where and how to move the objects, this phase is critical for the correctness and efficiency of the algorithm.

The new addresses of the live objects must ensure the preservation of their original heap order. To guarantee the correctness, one collector should never overwrite another collector's useful data. At the same time, we should decide a suitable parallel granularity for runtime efficiency.

Data races could exist between multiple collectors if they happen to compute the target address of the same object, or to relocate different objects to the same target address. We have to use atomic operations to eliminate the possible races in this phase. It is natural to use a group of objects as the parallelization granularity to avoid excessive atomic operations. In our design, we use heap block for the purpose. The heap is partitioned into fixed-size blocks, and each block has a block header for its metadata, i.e., block base address, block ceiling address, the state of the block, etc. The amount of metadata is a constant that is independent of the block size.

Only two atomic operations are needed for one block in the phase: one for taking the ownership of the block as a compacting source block, and the other for taking the ownership of the block as a compacting target block.

To ensure the correctness of the runtime execution process, we use a block state transition graph to guide the collectors to select proper blocks. More details will be discussed in Section 3.

2.3 Reference Fixing

Once the new address of an object is computed and stored in the object header, it is easy to parallelize the reference fixing phase. Each collector simply grabs a group of objects

(a block) in the heap and updates all the references in it as thread local data. This can be achieved by incrementing a global block index (or address) atomically. Since this phase is inherently highly parallelizable, we will not discuss it further in the following text.

2.4 Object Moving

To move the live objects in parallel is not as easy as to fix the references. The problem is due to the potential races between multiple collectors when they move objects from a source block to a target block. For example, they might write into a same target block, or write into a target block whose objects have not been moved away yet. The latter case happens when one collector's target block is another collector's source block.

In the object relocating phase, source block is used for the threads' synchronization control. That is, the collectors atomically grab the source blocks according to the heap order, and compute the target addresses of the live objects in the source block. Here in the object moving phase, we use the target block to control the moving. That is, only when a collector holds a block's ownership, can it move data to this block. The problem is how to guarantee that the data in the block has been moved away already before it is taken as a target block. A counter (*target_count*) is used for each block to solve the problem. As a source block, a block's live objects might be copied to more than one target blocks; *target_count* records the number of its target blocks. The value is set in the phase of object relocating. After that phase, there are three possible values for *target_count:*

1. For most blocks, *target_count* value is one, meaning all data from one block has been moved to a single target block.
2. Some blocks have their *target_counts* with value two, meaning part of the block data has been moved to one block, and the remaining part to another block.
3. There are also many blocks with value 0 in *target_count*. This happens when there are no live objects in those blocks.

During the object moving phase, *target_count* of a block is decremented by one when it finishes its data movement to one target block. When the *target_count* becomes zero, it means this block has no data left for moving, and it is ready to be used as a target block, i.e., a collector can move data from a source block into this block.

Since we use target block to control the parallel data moving, when a collector grabs the ownership of a target block, it should be able to find all of its source blocks. This requires a *source_list* for each block, which links all its source blocks. *target_count* and *source_list* jointly support the parallel object moving. More details will be discussed in Section 3. It should be noted that these two data structures require only two words in every block header, hence negligible space overhead.

2.5 Phases Composition

With all the phases parallelized, the last thing is to compose the phases into a complete collection process. This is straightforward in our design. Since the four phases are almost independent, we simply insert a barrier between two phases, where a new phase is only started after the old phase has finished.

There are some data passed from one phase to another:

1. Before the object relocating phase, the live objects have mark bit set in their object headers; then the collectors can iterate the heap to find all the live objects and compute their new locations;
2. The reference fixing phase needs the mark bit set as well, in order to find all the live objects to fix their references;
3. *target_count* and *source_list* should be set before the object moving phase. They are prepared in the relocating phase.
   After the moving phase, all these information are useless and can be cleared.

3 Parallelization Implementation

In previous section we have discussed the parallelization design. In this section we describe how we implement the design in details, and we focus on the phases of *object relocating* and *object moving*.

3.1 Object Relocating Implementation

During the compaction process, each block has two roles: It is a source block whose data are moved to new locations; it is also a target block where other live data are moved into. In the object relocating phase, each thread always holds a target block and a source block for target address computing. For each live object in the source block, the collector computes its target address in the target block. When the target block has no enough space, the collector grabs next target block. When the source block has no more live objects, the collector grabs another source block until all the blocks have been visited. Then the collector terminates its execution for this phase. When all the collectors finish the phase, they pass the barrier and enter next phase.

This phase decides if the following properties can be kept:

1. *Order preservation*: The blocks must be grabbed in heap order, and the objects' original order in the blocks is kept;
2. *Compaction*: The target addresses are contiguous in the heap;
3. *Load balance*: No collector is idle if there are remaining blocks for object relocating;
4. *Parallel efficiency*: The collectors do not conduct any redundant work except that required for object relocating.

To achieve the goals, the key idea is to use a state transition graph for each block to guide the relocating process. Each block is assigned with one of the following four states:

Block States	Meaning
UNHANDLED	Initial state of all blocks (neither a source block nor a target block.)
IN_COMPACT	Objects are under target addresses computation (i.e., the block is a source block)
COMPACTED	Objects' target addresses have been computed
TARGET	The block is a target block.

The collectors operate on the blocks according to the state transition graph shown in Fig. 3. And the state transition rules are given below.

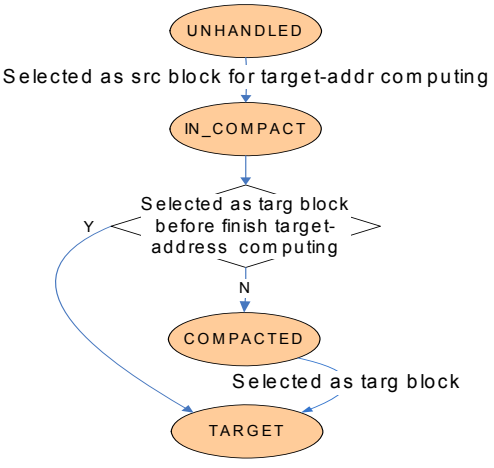


Fig. 3. Block state transition graph

Executing in parallel, each collector grabs from the heap a source block and a target block according to the heap address order. The rules for block state transitions are:

1. All the blocks are UNHANDLED at the beginning.
2. The collectors compete for an UNHANDLED source block in the heap order. If a block is grabbed, its state is set IN_COMPACT. Other failing collectors continue to compete for next source block in the heap order.
3. When a source block finishes all its objects relocating, its state is set to be COMPACTED, and the thread continues to grab a new source block.
4. At the same time, all the collectors compete for a target block in the heap order that is COMPACTED. If a block is grabbed, its state is set to be TARGET.
5. If a collector fails to grab a COMPACTED block in the heap order before its own source block, the thread uses its source block as its target block, and sets its state from IN_COMPACT to TARGET.

During the process, *target_count* and *source_list* are created and maintained accordingly.

Fig. 4 gives an example illustrating the *source-list*s built after the phase. In the figure, block #4 as a source block stays in both the *source-list*s of block #1 and #4, so the *target-count* of block #4 is 2, while that of block #9 is 1.

Based on the rules, we prove that the target address of any live object is no bigger than its original address in Theorem 1 below. This is a requirement for the parallelization correctness of the compactor algorithm.

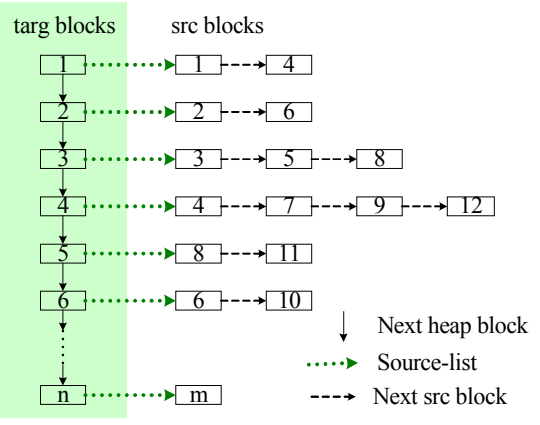


Fig. 4. *source-list* built in the object relocating phase

Theorem 1. After the object relocating phase, the target address of a live object is no bigger than its original address.

Proof. According to the state transition graph, a source block has IN_COMPACT state (transitioned from the UNHANDLED state); and a target block has TARGET state (transitioned from either COMPACTED or IN_COMPACT state). We prove the theorem in the following two cases depending on the target block state transition.

Case 1: A target block becomes TARGET from COMPACTED state. This means the collector can grab the target block before reaching to its own source block in heap address order, so a live object's target address in the destination block must be smaller than its original address in the source block;

Case 2: A target block becomes TARGET from IN_COMPACT state. This means the collector uses its own source block for the target block, i.e., the same block acts as both source and target block. In this case, the target address of a live object must be no bigger than its original address, since there are normally dead objects in the block, and the live objects are "moved" downwards to the block start. If there are no dead objects, the target address is the same as its original address. In this case, the object is not moved.

These two cases cover all the situations, so the theorem is proved. □

3.2 Object Moving Implementation

After the phase of fixing object references, the collectors are ready to move the live objects to their new locations. It is the phase doing the real compaction. The basic idea is similar to the object relocating phase, i.e., the collector always holds a source block and a target block; but the roles are flipped for the source and target blocks. In the relocating phase, the collectors' synchronization is mainly controlled by the grabbing of the source block ownership, while in this phase, that is done though the grabbing of the target block ownership.

We use a shared global variable *current_target* for the central control, which represents the last target block in the heap order that has been grabbed by the collectors. The algorithm is as following:

1. Set *current_target* to be the first block in the heap. (Use *CT* to represent *current_target*.)
2. Each collector *Ti* atomically picks up a source block *SBi* from the *source-list* of *CT*, and copies live objects from *SBi* to their new addresses in *CT*;
3. When a collector finishes copying all the live objects in *SBi* to *CT*, it atomically decrements the *target-count* of *SBi* and picks up another source block from *CT*'s *source-list*. (Note since block *SBi* can be the source block of more than one target blocks, the collector actually only copies those live objects that have target addresses in *CT*. The reaming live objects in *SBi* will be copied when their targeted block is processed.)
4. When the source blocks in *CT*'s *source-list* are run out, the collector looking for a source block chooses a new target block as *CT* according to the rules below, and loops back to Step 2.

When a collector is looking for a new block as *CT*, an eligible candidate has to satisfy either of the following conditions:

- The block's *target-count* is 0; (It means all live objects in it have been copied already.)
- The block's *target-count* is 1 while the first block in its *source-list* is itself.

With the rules, we prove in Theorem 2 that this phase never introduces race condition, i.e., no live object is overwritten before it is copied to a new location. This guarantees the correctness of the parallel compaction.

Theorem 2. In object moving phase, no live data are overwritten before they have been copied to their new locations.

Proof. According to the *current_target* block eligibility conditions, we prove the theorem in two cases:

Case 1: If the block's *target-count* is 0, all its live data have been copied or the block has no live data. This is a trivial case;

Case 2: When its *target-count* is 1 and the first block in its *source-list* is itself, it will be taken as the source block of itself. According to Theorem 1, the target address of every live object in this block is no bigger than its original address. When the data are copied to their new locations within the same block by a single thread, it is assured that no data loss can happen if the data are copied in order. When all the live data in this block are copied, Case 2 becomes Case 1, which has been proven already.

These two cases cover all the situations, so the theorem is proved. □

To illustrate the object moving phase, we give an example in Fig. 5 based on the *source-list*s built in Fig. 4. The target blocks are taken one by one in the heap order, and the source blocks are also taken one by one in the *source-list*s of the target blocks. All the collectors keep busy in the process.

Up to this point, we have described the design and implementation of the proposed parallel compactor. Note that the parallelization algorithms used for the four phases are

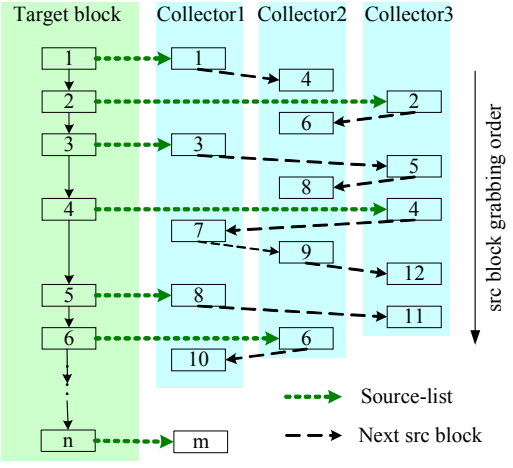


Fig. 5. Object moving phase illustration

not closely inter-dependent, i.e., as long as the necessary data for later phases are prepared by the earlier phases as described in subsection 2.5, the implementations are not constrained by what are described here. Actually we have implemented the phases of live object marking and object relocating in different ways in Apache Harmony.

We also want to emphasize that the number of collectors is configurable. A user can decide the number according to the heap size and/or the available number of cores. He or she can also choose to use different number of collectors for different phases. The option has also been implemented in Apache Harmony.

4 The Compactor Algorithm in Real GC

The parallel LISP2 compactor can be used as a standalone collector, or work with other collection algorithms. In this section, we describe how we use it in a real GC to support fallback compaction [9], and to support adjustable space boundaries.

4.1 Applied in a Generational GC

When it is used in a generational GC, the parallel LISP2 compactor is commonly used for major collections due to its in-place compaction advantages. Minor collections are usually conducted by a copying collector implementing semi-space or partial-forward algorithms.

As shown in Fig. 6, in a typical generational GC, the heap is partitioned into two parts, mature object space (MOS) and nursery object space (NOS). In order to achieve the best performance with a fixed-size heap, NOS size should be as big as possible to utilize as much the available free space. So we want to leave as small as possible the reserved free space in MOS for NOS copying. We should be able to adjust the boundary

Fig. 6. LISP2 compactor applied in generational GC

between MOS and NOS for the purpose. This has two requirements on the parallel LISP2 compactor:

1. It should always compact the live objects to the low end of MOS space, leaving a big contiguous free space to NOS;
2. When the reserved free space in MOS cannot accommodate all the NOS survivors in a minor collection, the collection can fallback to a major collection on the fly.

The first requirement can be satisfied trivially since it is one of the compactor's properties. The second requirement demands additional support from the compactor.

When a fallback happens, some NOS survivors have already been forwarded to MOS, some are still in NOS. Those forwarded survivors have two copies in the heap: the new copy in MOS and the original copy in NOS, resulting in an inconsistent heap state. To make it consistent, when the compactor marks an object in the live object marking phase, it checks if the object has been forwarded. If it has been forwarded, the collector only marks the new copy. Meanwhile, it updates any references to the old copy to point to the new one. Then after the live object marking phase, there would be no references pointing to the old copies, and heap consistency is maintained. The phases afterwards can be executed as usual.

4.2 The Compactor with a LOS Collector

In some GC designs, large objects are managed separately in a large object space (LOS). Fig. 7 is a typical heap layout that has LOS area. LOS is put at the low end because it is common to put NOS to the high end of the heap. (NOS is in the non-LOS part here. When there is MOS, MOS stays between LOS and NOS. MOS and NOS together are called non-LOS.) When LOS is fully occupied by large objects, a major collection is triggered.

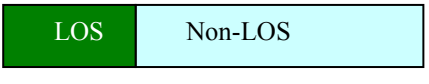


Fig. 7. LISP2 compactor with LOS

To leave little free space in LOS may trigger frequent expensive major collections, while too much free space reserved in LOS may waste the space. We need to support an adjustable boundary between LOS and non-LOS (called *LOS-boundary*). But since the compactor compacts live objects to the low end of non-LOS starting from the boundary, extra supports are needed for such an adjustment.

Fortunately, the parallel LISP2 compactor can support adjustable *LOS-boundary* if we know the new boundary value before the compaction. The idea is to specify the new boundary as the logical start address of the non-LOS area, as illustrated in Fig. 8. When

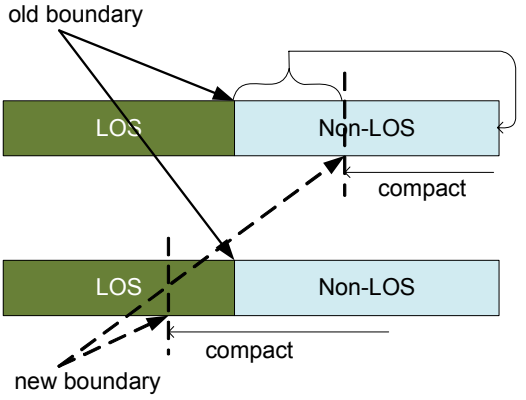


Fig. 8. Adjustable boundary with LOS

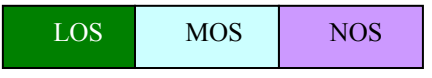


Fig. 9. Apply the compactor in Apache Harmony

shifting the boundary to non-LOS side, we logically connect the areas between the old boundary and the new boundary to the high end of non-LOS area; then the compaction can be done as usual starting from the new boundary. The original low end area is compacted to the high end of non-LOS. On the other hand, if we want to shift the boundary to LOS side, we set the first target block starting from the new boundary, then we can compact non-LOS as usual.

With the techniques described here, we can use the parallel LISP2 compactor to achieve good performance in a product-quality GC in Apache Harmony, which has the heap layout as shown in Fig. 9. The boundaries between LOS, MOS, and NOS can be adjusted at runtime to get highest heap utilization.

5 Evaluations and Discussions

We implemented the parallel LISP2 compactor in Apache Harmony and it is in the current main trunk source tree. To evaluate the design and implementation, we collected the data with SPECJBB2005 on an 8-core platform with Intel Core 2 2.8GHz processors. We ran the benchmark using 1GB heap size by default.

5.1 Scalability

We collected the GC total pause time and the time spent in different phases as shown in Fig. 10. In the experiments, we specified to use the original sequential and the parallel LISP2 compactor for the collections. The parallel compactor ran with 2, 4, 8 collectors.

It is seen from the figure that the overall pause time has been reduced steadily from 100% to 70%, 43% and 27% respectively. Fig. 10 also gives the speedups of the phases, which in average are 1.4x, 2.3x, and 3.7x respectively with 2, 4, 8 collectors.

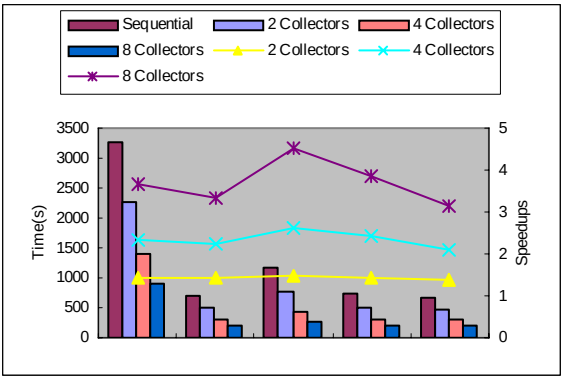


Fig. 10. GC time and speedups in phases with SPECJBB2005

Although the results above are good enough according to our experience in parallel GC, there are still opportunities for improvements. For example, in the object relocating and object moving phases, every source and target block requires an atomic operation to acquire. Almost all the blocks that have live data have been acting as source and target block once, so the number of the atomic operations is proportional to the number of blocks. It can be reduced if we use a bigger block size. The current block size is set to be 32KB, which is quite small compared to some other GCs.

5.2 Adjustable Boundaries

As we described in Section 4, our parallel LISP2 compactor can adjust its boundaries adaptively. We collected data to show the importance of this support for SPECJBB2005's performance, as shown in Fig. 11.

The current GC in Apache Harmony has two different NOS collection algorithms. One is the semi-space collector; the other is the partial-forward collector. To demonstrate the effectiveness of the adjustable boundaries, we let the NOS collector to simply copy the survivors in NOS to MOS in minor collections. In Fig. 11, we can see that the

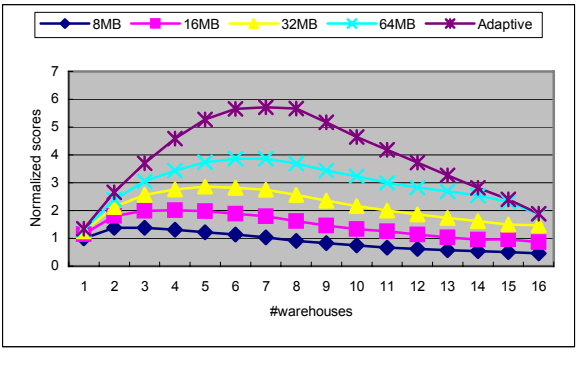


Fig. 11. SPECJBB2005 performance with different NOS sizes

adaptive NOS size can achieve better performance than all of other fixed NOS size settings. For example, it is almost 5x better than that of 8MB NOS size.

We can not demonstrate the effectiveness of the adjustable LOS boundary with SPECJBB2005, because it is not large object intensive.

6 Related Work

LISP2 compactor is known for its simplicity, but its parallelization is not straightforward if we want to keep the sliding properties, such as the contiguous free space, object order preservation, in-place collection, and object-level compaction granularity.

To the authors' knowledge, Flood et al. [4] is the only previous work trying to parallelize LISP2 compactor. Their collector pre-partitions the heap into a number of independent regions, and compacts them separately in parallel within the regions. The resulted free space is noncontiguous. This is a big loss of the LISP2 compactor's advantages, although they can alternate the compacting direction for each region so as to form [(n+1)/2] contiguous free areas. Nonetheless, their work achieved rather good scalability with SPECJBB and Javac applications. The speedups were around ~5x on 8-processor platform.

There are a couple of other non-LISP2 compactors proposed by the community. The threaded reference compactor suggested by Morris [10] and Jonkers [7] is inherently sequential due to its nature of scanning the heap back and forth to build the threaded reference in the heap order.

Abuaiadh et al [1] proposed a three-phase parallel compactor that uses a block-offset array and mark-bit table to record the live objects moving distance in blocks. When it moves the objects in the granularity of a heap block, it wastes about 3% space collecting SPECJBB. When it moves live objects in the granularity of individual object, the compaction time is increased by more than ~30%.

Kermany and Petrank [8] proposed the Compressor that requires two phases to compact the heap; Wegiel and Krintz [13] designed the Mapping Collector with nearly one phase. Both approaches depend on the virtual memory support from underlying operating system. The former one leverages the memory protection support to copy and adjust pointer references on a fault, and the latter one releases the physical pages that have no live data.

7 Summary

In this paper, we design and develop a fully parallel LISP2 compactor that compacts the live objects to a contiguous area in one end of the heap. It preserves all the sliding properties of the sequential LISP2 compactor. This parallel LISP2 compactor is fully parallel because all of its phases are parallelized. We have proved the correctness, implemented the parallel LISP2 compactor in Apache Harmony and evaluated it with a representative benchmark. Our result demonstrates that the collector can achieve 3.7x speedup on an 8-core platform (before the compactor is fully tuned).

Our current work and next step is to continue the fine tuning and leverage the compactor in a JIT-assisted GC, where the JIT can help in object allocation and release. We are also investigating how to reduce the sequential part in the GC implementation hence to achieve better scalability.

References

1. Abuaiadh, D., et al.: An efficient parallel heap compaction algorithm. In: The ACM Conference on Object-Oriented Systems, Languages and Applications (2004)
2. Borman, S., Sanitation, S.: Understanding the IBM Java Garbage Collector, http://www.ibm.com/
3. Cohen, J., Nicolau, A.: Comparison of compacting algorithms for garbage collection. ACM Transactions on Programming Languages and Systems 5(4), 532–553 (1983)
4. Flood, C., et al.: Parallel garbage collection for shared memory multiprocessors. In: The USENIX JVM Symposium (2001)
5. HotSpot Virtual Machine Garbage Collection, http://java.sun.com/javase/technologies/hotspot/gc/index.jsp
6. Jones, R.E.: Garbage Collection: Algorithms for Automatic Dynamic Memory Management. Wiley, Chichester (1996)
7. Jonkers, H.B.M.: A fast garbage compaction algorithm. Information Processing Letters (July 1979)
8. Kermany, H., Petrank, E.: The Compressor: Concurrent, incremental and parallel compaction. In: PLDI (2006)
9. McGachey, P., Hosking, A.L.: Reducing generational copy reserve overhead with fallback compaction. In: ISMM 2006, pp. 17–28 (2006)
10. Morris, F.L.: A time- and space-efficient garbage compaction algorithm. Communications of the ACM 21(8), 662–665 (1978)
11. Richter, J.: Garbage Collection: Automatic Memory Management in the Microsoft .NET Framework (November 2000)
12. Tuning the memory management system, http://edocs.bea.com/jrockit/geninfo/diagnos/memman.html
13. Wegiel, M., Krintz, C.: The Mapping Collector: Virtual Memory Support for Generational, Parallel, and Concurrent Compaction. In: ASPLOS 2008, Seattle, WA (March 2008)
14. Wu, M., Li, X.-F.: Task-pushing: a Scalable Parallel GC Marking Algorithm without Synchronization Operations. In: IEEE IPDPS 2007 (2007)

A Case Study in Tightly Coupled Multi-paradigm Parallel Programming

Sayantan Chakravorty[1], Aaron Becker[1], Terry Wilmarth[2], and Laxmikant Kalé[1]

[1] Department of Computer Science, University of Illinois Urbana-Champaign
[2] Center for Simulation of Advanced Rockets, Univ. of Illinois Urbana-Champaign

Abstract. Programming paradigms are designed to express algorithms elegantly and efficiently. There are many parallel programming paradigms, each suited to a certain class of problems. Selecting the best parallel programming paradigm for a problem minimizes programming effort and maximizes performance. Given the increasing complexity of parallel applications, no one paradigm may be suitable for all components of an application. Today, most parallel scientific applications are programmed with a single paradigm and the challenge of multi-paradigm parallel programming remains unmet in the broader community.

We believe that each component of a parallel program should be programmed using the most suitable paradigm. Furthermore, it is not sufficient to simply bolt modules together: programmers should be able to switch between paradigms easily, and resource management across paradigms should be automatic. We present a pre-existing adaptive runtime system (ARTS) and show how it can be used to meet these challenges by allowing the simultaneous use of multiple parallel programming paradigms and supporting resource management across all of them. We discuss the implementation of some common paradigms within the ARTS and demonstrate the use of multiple paradigms within our feature-rich unstructured mesh framework. We show how this approach boosts performance and productivity for an application developed using this framework.

1 Introduction

A parallel programming paradigm defines how concurrent tasks in a parallel program access data and interact with each other and how those interactions are expressed by the programmer. Such paradigms are created to address the needs of specific classes of problems in parallel computing, making implementations easier to develop. A programmer often develops a parallel algorithm with a certain parallel programming paradigm in mind. Programmers have a wide range of paradigms to choose from, such as message passing (e.g. MPI), shared address spaces (e.g. Global Arrays[28], UPC[9], OpenMP[7], HPF[11]), message-driven (active messages, actors, Charm++) and stream processing. This variety exists because not all paradigms are suitable for all problems. Choosing the correct

J.N. Amaral (Ed.): LCPC 2008, LNCS 5335, pp. 279–291, 2008.

paradigm for an application yields benefits in terms of lower programming effort, elegant, easily maintained code, and better performance.

As parallel applications become increasingly complex with multiple constituent algorithms, no single paradigm is suitable for all the different parts of an application. Most programmers currently choose a paradigm that is suitable for the bulk of the application and force the rest into the same paradigm. This results in reduced programmer productivity via inelegant code, potentially more errors, longer development time and lower maintainability. It can also result in poor performance when the inherent parallelism in the problem could be more naturally and fully expressed in some other paradigm.

Developing each application component with the most suitable paradigm would require a multi-paradigm parallel programming system in which different paradigms can be tightly coupled. In such a system, *the application would allow different paradigms to be used concurrently.* In addition, *the components of an application would not be restricted to a single paradigm*, nor would components of a certain paradigm be restricted to a subset of the physical processor space. Without a tightly coupled multi-paradigm environment, performance and productivity are adversely affected when software components cannot be cleanly expressed within one paradigm. Moreover, this multi-paradigm system should enable *resource management across all the paradigms on all the processors.* This is important for scaling applications to even moderately sized machines, since resource management issues like computational load imbalance and communication bottlenecks are often the biggest roadblocks to scaling.

A programmer faces further challenges when selecting a non-mainstream paradigm that may be ideally suited to her problem. There is a huge barrier to entry for alternative paradigms caused by the relative dominance of MPI. In the absence of decisive performance benefits, the best way for new models to gain traction is to interoperate cleanly with existing models, both to allow the use of existing libraries and to facilitate reuse of new code.

2 Background

Approaches to multi-paradigm parallel programming roughly fall into four categories: 1) multi-paradigm parallel languages; 2) extensions to existing parallel programming models; 3) interoperability libraries; and 4) run-time systems.

Multi-paradigm parallel programming languages have primitives that let a user take advantage of different paradigms in a tightly coupled fashion. One major disadvantage of this approach is that existing software implemented in various paradigms is difficult or impossible to reuse, and conversely, software developed in such languages may not be reusable in other applications developed with other languages. As an example of a multi-paradigm language, mpC[23] is a C superset that provides *network objects* to describe data and communication layouts in a parallel environment. It supports both task and data parallelism and enables both computation and communication optimizations.

Extensions to parallel programming models enhance existing models to allow the use of additional paradigms. Efforts to merge OpenMP and MPI such as Extended OpenMP (EOMP) [30] and MPI+OpenMP [6] fall into this category. Codes developed using mixed-mode techniques have been quite successful in some cases [29] Most extensions involve only two paradigms without a general framework for adding more. Moreover, different paradigms are often executed as different processes or kernel threads, increasing the cost of switching between paradigms. Although MPI+OpenMP attempts dynamic load balancing by moving OpenMP threads among processes [6], it is restricted to moving them among processors on the same node and cannot do truly global resource management.

Interoperability libraries provide interfaces between systems that implement different parallel models. Fortran M uses MPCL (message-passing compatibility library) to interface with other message passing libraries and HPF [12]. Here, the freedom to use multiple paradigms is marred by the additional complexity of the interface. Fortan M tackles the resource management issues by allowing access to its resource management facilities through MPCL. Other attempts to bolt multiple paradigms together, such as PVM with Solaris Threads, resulted in poor performance and undue code complexity [25].

Parallel Adaptive Run-time Systems (ARTS) attempt to provide the set of tools that are required to implement different parallel programming models. Existing models and languages can be implemented on top of such a run-time system and can interoperate using a common substrate. ARTS provide resource management capabilities common to all models, thereby relieving the programmer of this task. TPVM[10] was an early extension to PVM that implemented a thread-oriented, event-driven run-time and the notion of virtual processors. It consistently outperformed PVM in several experiments.

Our approach relies on Converse [19], an ARTS which fully realizes the tightly coupled interoperability of multiple paradigms. Converse meets many of the goals mentioned in a recent report from Berkeley [3], including the need for models to be independent of the number of processors, and it alleviates cognitive load on programmers by performing automatic resource management.

3 A Multi-paradigm Runtime System

Object-based virtualization [18] is a flexible approach to implementing a variety of interoperable programming paradigms. It encourages the decomposition of a computation into a large number of interacting objects called *virtual processors* (VPs). The task of mapping VPs to physical processors is handled by the ARTS, which can change the mapping at run-time by migrating VPs between processors. The ARTS is also responsible for message delivery between VPs. A scheduler on each processor selects which local VP executes next. The scheduler is message-driven and only schedules VPs that have pending messages.

Object-based virtualization does not dictate the paradigm used within a VP, so different VPs may use different paradigms. Converse [19] provides the tools for implementing different paradigms in a message-driven system with

object-based VPs. Apart from providing a scheduler on each processor, Converse also provides a messaging framework with primitives for point-to-point communication and multicasts, and methods for handling a message on the receiving processor. Converse also offers user-level threads[32] with low context switch overhead and the ability to migrate threads between processors. Local Converse threads share the scheduler with incoming messages.

Programming paradigms developed using the Converse ARTS give programmers the ability to efficiently compose independently-developed components into a single application or higher-level component. Components developed using separate paradigms can overlap their execution in time and over processors. Since multi-paradigm VPs share the same address space on a processor, a flow of control can switch paradigms cheaply via function calls or local messages within a processor. Therefore different paradigms implemented on Converse can be tightly coupled within a single application.

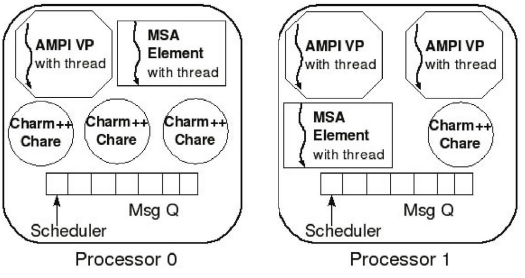


Fig. 1. The Converse ARTS allows multiple VPs with different paradigms on the same processor

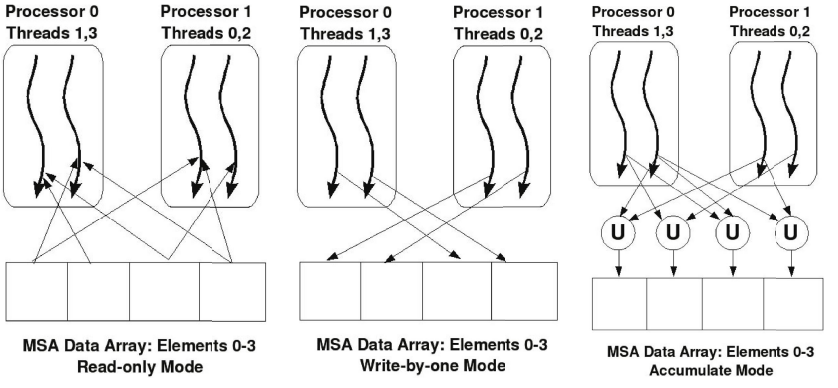
Numerous models have already been implemented on the Converse ARTS. These include the *message-passing* model via Adaptive MPI, the *message-driven* model via Charm++, and a phase-based *distributed shared memory* model called *Multi-phase shared arrays* (MSA). Figure 1 shows VPs using these models while sharing the same processor. Similarly, *global address space* languages are supported via an implementation of ARMCI. An *orchestration* model called Charisma allows for clear expression of control and data flows between serial components.

Object-based virtualization enables a number of performance benefits such as adaptive overlap of computation and communication [18], dynamic measurement-based runtime load balancing [31] and dynamic communication optimizations [22]. Since all paradigms in an application use Converse, resource management need not be limited to one paradigm. For example, 1) the load balancer takes into account work loads of the VPs belonging to all paradigms while trying to balance load; 2) if two VPs using different paradigms send each other many small messages, the communication optimization library can merge these into fewer larger messages.

Charm++ and MPI have both been described extensively, and their relative merits and deficiencies have been explored [2,4,13,14,20,26]. Therefore we will not describe them further except to say that both have proven suitable for developing complex parallel applications. Charm++ is implemented on top of Converse, and AMPI [17] is an implementation of MPI on Converse.

Together with Multiphase Shared Array (MSA), a simple shared memory model which we will now describe, these models make up the bulk of our unstructured meshing framework.

MSA consists of shared arrays that can change between three possible access modes. An MSA array can be constructed from any user-specified type. The lifetime of an MSA array is divided into phases, with all threads accessing the array in the same access mode during each phase. Phase boundaries are marked by synchronization. The phase based nature of MSA programs means that they can never have deadlocks or race conditions. The three possible access modes (shown in Figure 2) are:



(a) Read only mode (b) Write by one mode (c) Accumulate mode

Fig. 2. The three different access modes of a MSA array

Read-only mode: All threads can only read from the MSA array during this phase. Each element can be read by multiple threads.

Write-by-one mode: In this mode, all threads are permitted to write to the MSA array, but no element can be written to by multiple threads.

Accumulate mode: All threads can update the MSA array and multiple threads can update a single element. For each element, the data provided by different threads is accumulated using a user defined associative commutative operation such as addition, multiplication, max, set union and set intersection.

MSA represents a compromise between the convenience of a global address space and the performance and correctness problems associated with unfettered access to shared data [5]. The restrictions imposed by the access phases allow for more efficient communication while also preventing common shared memory programming hazards. MSA has proven useful in codes varying from matrix multiplication to distributed hashtables to molecular dynamics and has provided significant advantages to parallel programmers across many problem domains.

The Converse runtime has also been used to implement a variety of other parallel programming paradigms. One example is Charisma [15], an orchestration

language that lets the programmer specify the control and macro data flows of a parallel program separately from the sequential portions. Charisma is built on top of Charm++. The user expresses the global message flow in the orchestration code without fragmenting it among all the different types of objects in a complicated parallel application. The orchestration code can express all commonly used communication patterns, including point-to-point, broadcast, multicast, reductions, scatter and gather. The sequential portions are normal C++ code. A Charisma program can be combined with any library written using one of the parallel programming paradigms supported by the Converse runtime system.

Aggregate Remote Memory Copy Interface (*ARMCI*) [27] supports high performance remote memory copy on multiple platforms. It offers blocking and non-blocking versions of data transfer operations, synchronization operations and memory allocation and deallocation routines. ARMCI is used as the foundation for a number of global address space languages such as Global Arrays [28] and Co-Array Fortran [8]. ARMCI is implemented on the ARTS by encapsulating each ARMCI process within a threaded Charm object [16]. The ARTS can perform intelligent resource management for any application using ARMCI.

4 ParFUM: An Example of Multi-paradigm Programming

ParFUM [24] is a framework for the parallelization of unstructured mesh applications. It provides the programmer with a rich set of features such as mesh partitioning, communication between mesh partitions, mesh adaptivity, mesh locking, collision detection and data transfer. Due to the complexity of these features, each was implemented in the parallel paradigm most suited to it. These differences in paradigm are largely hidden from the user, to whom the application appears to be completely within the message-passing style. We describe some of these features and our programming model choices for their implementation below. We also discuss situations in which multiple parallel programming paradigms were used to provide a single feature.

In ParFUM, each mesh partition is associated with a single VP, and a driver routine is invoked by each of these VPs. In most applications, mesh nodes and elements (collectively, *entities*) along partition boundaries require data from entities on neighboring partitions to compute local solutions. ParFUM provides functionality for adding local read-only copies of remote entities, or *ghosts*, to the partition boundary. A single collective ParFUM call updates all ghost entities with data from the original entities on neighboring partitions.

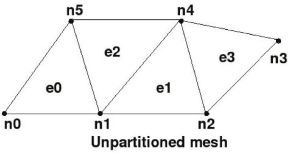


Fig. 3. An unpartitioned mesh

ParFUM also provides synchronization primitives to update the values of shared nodes during a simulation. The user code in the driver routine is typically written in a message passing style with blocking ghost update calls and synchronization routines such as barriers and reductions, and makes use of

adaptivity and locking mechanisms which use the message-driven style in a manner transparent to the programmer. As a result, both serial codes and pre-existing MPI codes can be easily modified to use the ParFUM library and its features.

4.1 Mesh Partitioning

Step 1: Compute a mapping of elements to partitions that produces approximately balanced partitions and minimizes the number of boundary elements. We start with an arbitrary mapping as an input to PARMETIS [21], a third-party MPI library for parallel partitioning that we use without modification via AMPI. As shown in Step 1 in Figure 4, PARMETIS takes in the connectivity of the mesh elements and produces a mapping of elements to partitions. Here multi-paradigm programming enables us to use a library developed by subject matter experts without having to re-implement it in another paradigm.

ParFUM divides the entities of a serial mesh into one partition per VP, using a memory-efficient parallel partitioner to handle large meshes with large numbers of partitions. Figure 3 shows a simple 2D mesh with triangular elements that is to be partitioned between two VPs. We use this simple mesh to illustrate the parallel partition algorithm:

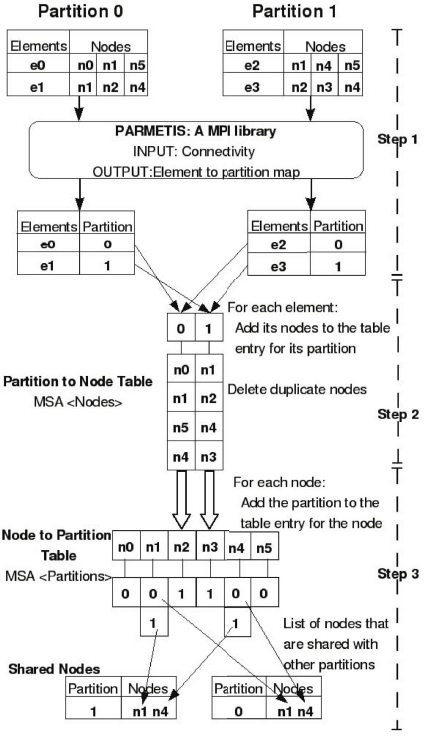


Fig. 4. Steps to partition a mesh

Step 2: Create partitions and send them their entity data. This is easy for elements because the mapping tells us exactly which partition each element belongs to. However, a node belongs to all partitions with an element adjacent to that node (for e.g.. $n1$ and $n4$ belong to both partitions in the example in Figure 4), so a node's ownership information is scattered across an unknown number of VPs on different processors. Collecting this information is simplified with a global table indexed by VP which stores all nodes owned by that VP's partition. This *Partition-to-Node* table has a list of nodes for each VP. Step 2 in Figure 4 shows that for each element, its nodes are added to the *Partition-to-Node* table at the entry for the element's partition (the partition to which PARMETIS has mapped this element in Step 1), with duplicate nodes being deleted.

Thus, in the first phase of this step the *Partition-to-Node* table is populated by all the VPs and in the second phase it is read by each VP. MSA, with its

separate accumulate and read modes, is ideally suited for this. The *Partition-to-Node* table is implemented as a MSA array of node lists in accumulate mode. A message passing implementation would have been more complex since a VP does not know how many nodes to expect from other VPs. Before passing the nodes, each VP would have to tell all the others how many to expect. This would lead to less readable code, higher communication, and poorer performance.

Step 3: Find the nodes shared between different pairs of partitions. This can be calculated by creating a global table (called the *Node-to-Partition* table) that maps each node to all the partitions to which it belongs. As shown in Step 3 of Figure 4, for every node owned by a VP, the VP adds itself to the node's entry. After all the VPs have finished writing to the table, each VP looks at the entry each of its nodes to determine the other partitions sharing that node. This lets ParFUM build up a list of nodes shared by every pair of neighboring partitions. MSA is an ideal fit for Step 3 for the same reasons as in Step 2.

4.2 Mesh Adaptivity

ParFUM implements two types of mesh adaptivity: *incremental* and *bulk* mesh modification. Both approaches provide low-level primitives for *edge bisect*, *flip*, and *edge contract* operations. In the incremental case, these are self-contained parallel primitive operations that leave the mesh in a consistent state, updating all ghost layers and adjacencies as needed. The faster, lightweight bulk operations currently under development in ParFUM perform en masse mesh modifications before updating the ghost layers and adjacencies.

These primitives lock the affected mesh entities so that multiple operations can simultaneously modify adjacent areas of the mesh, using the ParFUM locking functionality described earlier. Once the affected region of the mesh is locked, modification of the mesh can proceed.

An example of the edge bisect primitive is shown in Figure 5. When such an operation takes place across a partition boundary, as shown by the thicker line in the figure, the communication is very localized and specific. In (a), we highlight an element that we wish to bisect along its longest edge, which happens to be on the boundary between partitions A and B. The first step is to lock the region of the mesh that will be modified. This is shown in (b) which highlights the original element, the neighbor across the edge to be split, and one adjacent element for each of these (which will need to have adjacency data updated). Locking requires the first element to send a message to all the affected elements and then suspend to wait for a response about the success or failure of the lock. If locking is successful, two new elements and a node will

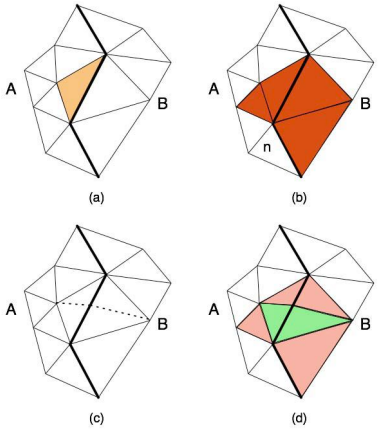


Fig. 5. Parallel edge bisection

then be added by the operation. The new node will bisect the longest edge, and the two new elements will be along the new edge created between the new node and the node n in (b). This new topology is shown in (c). Because the node is shared, a record must be made of it on both partitions A and B. The bisect is performed on the first side of the edge to be bisected, and all information about the new element and node on that side is then transmitted to the adjacent element on partition B. This side completes the second half of the operation, updating all relevant adjacency data, and in return sends information about the new element created back to partition A where the adjacency of the new element is updated. In (d), the lightly shaded elements have connectivity and adjacency updates performed on them, while the darker shaded elements are new elements added to the mesh.

Due to the unpredictable nature of modification messages, it is impossible for a partition to predict when one of its neighbors will invoke adaptivity functions. To accomplish this with MPI, we would need a polling loop consisting of a wild card receive. Once a message is received, its type is checked and it is processed accordingly. This amounts to a re-implementation of some capabilities of the Converse scheduler. However, the message-driven paradigm is ideally suited for this problem. The receiving partition does not need to expect incoming messages and processes messages as they are received.

Charm++ is excellently suited to adaptivity algorithms, as operations are confined to regions of the mesh determined by the state of the solution at a particular point in time. These problems are highly irregular and dynamic, and as such are also a perfect match for the virtualization capabilities provided by Charm++. Having multiple partitions per processor makes load balancing straightforward when refinement over particular partitions increases their load. Mesh adaptivity is achieved in ParFUM by associating a special type of chare array, called a *bound* array, with the AMPI VPs. Thus, each partition has a chare array element associated with it which performs the message-driven aspects of mesh adaptivity. The "bound" aspect of these elements means that when migration takes place, a VP is bound to its associated chare array element such that they always migrate together.

5 Multi-paradigm Applications

ParFUM has been used to develop a number of parallel unstructured mesh applications [24]. These applications utilize the many features provided by ParFUM for faster development and better performance. Since each component of ParFUM was written using the paradigm or paradigms most suitable for it, ParFUM applications are examples of multi-paradigm parallel programming.

For example, TentPitcher is a novel algorithm for solving hyperbolic systems via the Spacetime Discontinuous Galerkin (SDG) method developed at the Center for Process Simulation and Design at UIUC [1]. In converting this algorithm to run in parallel, we made extensive use of the multi-paradigm capabilities of ParFUM.

This algorithm operates on a triangular mesh. Rather than picking a timestep interval and advancing each vertex in time by that interval, TentPitcher performs local solutions by choosing a single vertex at a local time minimum and moving it as far forward in time as possible based on causality constraints and error estimates. The result is a highly asynchronous algorithm in which many local solutions may be computed independently, with no global synchronization.

To efficiently solve problems with sharp features, such as shock propagation, high degrees of mesh refinement and coarsening are required. However, since this algorithm has no explicit timestepping, there is no natural opportunity to globally modify the mesh, as is standard practice in parallel adaptive finite element codes. In addition, typically only small areas of the mesh require modification and the vast majority of elements are unaffected, so doing global adaptivity will hurt performance. Therefore, we must perform all mesh modifications locally. Adaptivity operations like this are well suited to a Charm++ approach, where the programmer can send a lock/unlock or adaptivity messages and invoke the appropriate function on another partition.

This code mingles parallel programming paradigms with ease. MPI calls are used for bulk communication such as checkpointing and output, while Charm++ is utilized for locking and adaptive operations. Multiple paradigms coexist transparently, even within a single function. This leaves the programmer free to use whatever paradigm is most suitable at a very fine granularity, rather than choosing which paradigm is suitable at an application level.

6 Conclusions

Developing parallel software involves additional complexity over sequential software. The ability to develop modules in the most suitable parallel programming paradigm and to reuse them in applications that incorporate multiple paradigms improves programming productivity. The complex, adaptive multi-physics applications of the future require sophisticated and automated resource management. A multi-paradigm adaptive run-time system (ARTS) provides such support.

We demonstrated such an ARTS, called Converse, that allows multiple work units on each processor and interleaves their execution based on availability of remote data and messages. These abilities are crucial to our goal of efficiently supporting tightly coupled multi-paradigm interoperability in a parallel programming environment. The utility of this approach was illustrated by 1) describing multiple paradigms implemented using our ARTS, and 2) showcasing a parallel unstructured mesh framework and two associated applications that leverage this multi-paradigm interoperability.

ARTS also have positive implications for new parallel programming paradigms. In order for new paradigms to come into use, programmers need to 1) hear about them, 2) learn how to use them, 3) find an implementation of them, and 4) not sacrifice re-usability of their code should they choose to adopt them. Incorporating new paradigms directly on the Converse ARTS lowers the barrier to entry for the adoption of new paradigms by satisfying three of those

criteria, providing the knowledge of and access to new paradigms in a way that will make them most immediately usable and subsequently re-usable in future codes.

We believe this approach is essential for productive and efficient parallel programming, particularly for the complex applications and petascale computing environments of the near future. We have been advancing the ARTS approach for over a decade, and hope that many new paradigms will be developed with it.

References

1. Abedi, R., Chung, S.-H., Erickson, J., Fan, Y., Garland, M., Guoy, D., Haber, R., Sullivan, J.M., Thite, S., Zhou, Y.: Spacetime meshing with adaptive refinement and coarsening. In: SCG 2004: Proceedings of the twentieth annual symposium on Computational geometry, pp. 300–309. ACM Press, New York (2004)
2. Almasi, G., Archer, C., Castanos, J.G., Gupta, M., Martorell, X., Moreira, J.E., Gropp, W., Rus, S., Toonen, B.: Mpi on blue gene/l: Designing an efficient general purpose messaging solution for a large cellular system
3. Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., Yelick, K.A.: The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Dept, University of California, Berkeley (December 2006)
4. Chiang, C.-C.: Low-level language constructs considered harmful for distributed parallel programming. In: ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference, pp. 279–284. ACM Press, New York (2004)
5. Choi, S.-E., Lewis, E.C.: A study of common pitfalls in simple multi-threaded programs. SIGCSE Bull. 32(1), 325–329 (2000)
6. Corbalan, J., Duran, A., Labarta, J.: Dynamic load balancing of mpi+openmp applications. Icpp, 195–202 (2004)
7. Dagum, L., Menon, R.: OpenMP: An Industry-Standard API for Shared-Memory Programming. IEEE Computational Science & Engineering 5(1) (January-March 1998)
8. Dotsenko, Y., Coarfa, C., Mellor-Crummey, J.: A multi-platform co-array fortran compiler. In: Proceedings of the 13th International Conference of Parallel Architectures and Compilation Techniques (PACT 2004), Antibes Juan-les-Pins, France (October 2004)
9. El-Ghazawi, T., Cantonnet, F.: Upc performance and potential: a npb experimental study. In: Proceedings of the 2002 ACM/IEEE conference on Supercomputing, pp. 1–26. IEEE Computer Society Press, Los Alamitos (2002)
10. Ferrari, A., Sunderam, V.S.: Multiparadigm distributed computing with TPVM. Concurrency: Practice and Experience 10(3), 199–228 (1998)
11. Foster, I., Avalani, B., Choudhary, A., Xu, M.: A compilation system that integrates high performance fortran and fortran M. In: Proceedings 1994 Scalable High Performance Computing Conference (1994)
12. Foster, I., Xu, M.: Libraries for parallel paradigm integration. In: Kalia, R., Vashishta, P. (eds.) Toward Teraflop Computing and New Grand Challenge Applications. Nova Science Publishers (1994)
13. Gursoy, A., Kalé, L.V.: Performance and modularity benefits of messagedriven execution. Journal of Parallel and Distributed Computing 64, 461–480 (2004)

14. Hardwick, J.: Porting a Vector Library: a Comparison of MPI, Paris, CMMD and PVM. Technical Report CMU-CS-94-200, Carnegie Mellon University (1994)
15. Huang, C., Kale, L.V.: Charisma: Orchestrating migratable parallel objects. In: Proceedings of IEEE International Symposium on High Performance Distributed Computing (HPDC) (July 2007)
16. Huang, C., Lee, C.W., Kalé, L.V.: Support for adaptivity in armci using migratable objects. In: Proceedings of Workshop on Performance Optimization for High-Level Languages and Libraries, Rhodes Island, Greece (2006)
17. Huang, C., Zheng, G., Kumar, S., Kalé, L.V.: Performance evaluation of adaptive MPI. In: Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006 (March 2006)
18. Kalé, L.V.: Performance and productivity in parallel programming via processor virtualization. In: Proc. of the First Intl. Workshop on Productivity and Performance in High-End Computing (at HPCA 10), Madrid, Spain (February 2004)
19. Kale, L.V., Bhandarkar, M., Brunner, R., Yelon, J.: Multiparadigm, Multilingual Interoperability: Experience with Converse. In: Proceedings of 2nd Workshop on Runtime Systems for Parallel Programming (RTSPP), Orlando, Florida, USA. LNCS (March 1998)
20. Kale, L.V., Bohm, E., Mendes, C.L., Wilmarth, T., Zheng, G.: Programming Petascale Applications with Charm++ and AMPI. In: Bader, D. (ed.) Petascale Computing: Algorithms and Applications, pp. 421–441. Chapman & Hall/CRC Press, Boca Raton (2008)
21. Karypis, G., Kumar, V.: Parallel multilevel k-way partitioning scheme for irregular graphs. In: Supercomputing 1996: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM), p. 35 (1996)
22. Kumar, S.: Optimizing Communication for Massively Parallel Processing. Ph.D thesis, University of Illinois, Urbana-Champaign (May 2005)
23. Lastovetsky, A.L.: Mpc: a multi-paradigm programming language for massively parallel computers. SIGPLAN Not. 31(2), 13–20 (1996)
24. Lawlor, O., Chakravorty, S., Wilmarth, T., Choudhury, N., Dooley, I., Zheng, G., Kale, L.: Parfum: A parallel framework for unstructured meshes for scalable dynamic physics applications. Engineering with Computers 22(3-4), 215–235
25. Leichtl, J., Crandall, P.E., Clement, M.J.: Parallel programming in multi-paradigm clusters. In: HPDC 1997: Proc. of the 6th IEEE Int. Sym. on High Performance Distributed Computing, Washington, DC, USA, p. 326. IEEE Computer Society, Los Alamitos (1997)
26. Marc Snir, S.O., et al.: MPI: The Complete Reference, vol. 1. The MIT Press, Cambridge
27. Nieplocha, J., Carpenter, B.: Armci: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In: Rolim, J., et al. (eds.) IPPS-WS 1999 and SPDP-WS 1999. LNCS, vol. 1586. Springer, Heidelberg (1999)
28. Nieplocha, J., Harrison, R.J., Littlefield, R.J.: Global arrays: A nonuniform memory access programming model for high-performance computers. J. Supercomputing (10), 197–220 (1996)
29. Smith, L., Bull, M.: Development of mixed mode mpi/openmp applications. In: Scientific Programming, 9(2-3/2001), Presented at Workshop on OpenMP Applications and Tools (WOMPAT 2000), San Diego, Calif., July 6-7, 2000, pp. 83–98 (2000)
30. Wang, J., Lai, J., Zhao, Y., Zhang, S.: Multi-paradigm and multi-grain parallel execution model based on smp-cluster. In: IEEE John Vincent Atanasoff 2006 International Symposium on Modern Computing, pp. 266–272 (2006)

31. Zheng, G.: Achieving High Performance on Extremely Large Parallel Machines: Performance Prediction and Load Balancing. Ph.D thesis, Department of Computer Science, University of Illinois, Urbana-Champaign (2005)
32. Zheng, G., Lawlor, O.S., Kalé, L.V.: Multiple flows of control in migratable parallel programs. In: 2006 International Conference on Parallel Processing Workshops, Columbus, Ohio, August 2006, pp. 435–444. IEEE Computer Society, Los Alamitos (2006)

ASYNC Loop Constructs for Relaxed Synchronization

Russell Meyers and Zhiyuan Li

Department of Computer Science
Purdue University, West Lafayette IN 47906, USA
{rmeyers,li}@cs.purdue.edu

Abstract. Conventional iterative solvers for partial differential equations impose strict data dependencies between each solution point and its neighbors. When implemented in OpenMP, they repeatedly execute barrier synchronization in each iterative step to ensure that data dependencies are strictly satisfied. We propose new parallel annotations to support an *asynchronous computation model* for iterative solvers. At the outermost level, the ASYNC_REDUCTION keyword is used to annotate the iterative loop as a candidate for asynchronous execution. The ASYNC_REGION contains inner loops which may be annotated by ASYNC_DO or ASYNC_REDUCTION. If the compiler accepts the ASYNC_REGION designation, it converts the iterative loop into a parallel section executed by multiple threads which divide the iterations of each ASYNC_DO or ASYNC_REDUCTION loop and execute them without having to synchronize through a conventional barrier. We present experimental results to show the benefit of using ASYNC loop constructs in multigrid methods and an SOR-preconditioned CG solver.

1 Introduction

Many important applications use iterative solvers to solve partial differential equations (PDE's). It has been found for quite some time that there exist a class of iterative solvers which are allowed to follow a loose data dependence relationship between each data point and its neighbors [3,4,7]. Under such an *asynchronous computation model*, the update of a data point does not need to strictly depend on the most updated values of its neighbors. Instead, Some older values of its neighbors can be used before the newest values become available. It may take more iterations for an asynchronous algorithm to converge or to achieve the same numerical accuracy as its synchronous counterpart. However, when implemented on parallel systems, especially those of a large size, the asynchronous algorithms suffer less from the interconnect latency than their conventional counterparts.

Unfortunately, current parallel languages and language extensions (such as OpenMP [5]) do not effectively support the asynchronous computation model. When implemented with OpenMP parallel annotations, for example, an iterative solver typically has a sequential outermost loop containing a number of parallel inner loops. Each parallel loop annotation implies a barrier synchronization

J.N. Amaral (Ed.): LCPC 2008, LNCS 5335, pp. 292–303, 2008.

point at the end of the loop, where all processors must meet before simultaneously proceeding to the next statement. Such barrier synchronization, executed repeatedly in each iterative step, dictates the conventional strict synchronous computation model, at the expense of performance penalty due to the interconnect latency. Barriers also severely limit the compiler's ability to generate efficient machine code.

In this paper, we propose three new loop annotations, called *ASYNC_DO*, *ASYNC_REDUCTION*, and *ASYNC_REGION*, respectively. At the outermost level, the ASYNC_REGION keyword is used to annotate the iterative loop as a candidate for asynchronous execution. If the compiler accepts the ASYNC_REGION designation, it converts the iterative loop into a parallel section executed by multiple threads. Embedded in ASYNC_REGION are inner loops which may be annotated by ASYNC_DO or ASYNC_REDUCTION, possibly accompanied by ordinary OpenMP parallel `DO` loops and sequential loops. If the ASYNC_REGION designation is accepted by the compiler, the threads will divide the iterations of each ASYNC_DO or ASYNC_REDUCTION loop and execute them without having to synchronize through a conventional barrier. The threads will also divide the iterations of an ordinary OpenMP parallel `DO` loop, but they will synchronize through a barrier. An OpenMP parallel section (such as parallel `DO`) embedded in ASYNC_REGION does not cause spawning a new set of threads, because ASYNC_REGION at the outer level is already executed by multiple threads.

Comparing to directly implementing asynchronous algorithm using P-threads or existing OpenMP loop constructs, the iterative solver written with the proposed new constructs gives the compiler the flexility to decide whether to implement the annotated candidates in the asynchronous manner. The programmer does not need to commit the iterative solver to the asynchronous execution model. We present experimental results to show the benefit of using ASYNC loops in 2D and 3D multigrid methods as well as an SOR-preconditioned conjugate gradient linear system solver.

2 ASYNC Loops

ASYNC_DO Loops. ASYNC_DO annotates a DO loop whose iterations can be executed in parallel by multiple processors without barrier synchronization. However, it is different from an OpenMP parallel `DO` with a `nowait` label, as will be clear later. Its syntax, analogous to that of an OpenMP parallel DO loop, is in the form of `!$ASYNC_DO` *parallel clause*, where *parallel clause* takes the same form and meaning as its counterpart in OpenMP *sans* the reduction clause [5]. Figure 1 shows an example on how to annotate parallel loops by ASYNC_DO. When the shown ASYNC_DO is embedded in an ASYNC_REGION accepted by the compiler, it will be transformed by the compiler into an iteration-partitioned loop shown in Figure 2. Notice the absence of barrier synchronization.

ASYNC_REDUCTION. The ASYNC_REDUCTION is supported by a relaxed barrier tree structure which allows a thread, depending on its thread ID,

```
!$ASYNC_DO default(shared)
!$         private(i1,i2,i3,u1,u2)
      do i3=2,n3-1
        do i2=2,n2-1
          do i1=1,n1
            u1(i1) = u(i1,i2-1,i3)
     >           + u(i1,i2+1,i3)
     >           + u(i1,i2,i3-1)
     >           + u(i1,i2,i3+1)
            u2(i1) = u(i1,i2-1,i3-1)
     >           + u(i1,i2+1,i3-1)
     >           + u(i1,i2-1,i3+1)
     >           + u(i1,i2+1,i3+1)
          enddo
          do i1=2,n1-1
            r(i1,i2,i3) = v(i1,i2,i3)
     >           - a(0) * u(i1,i2,i3)
     >           - a(1) * (u(i1-1,i2,i3)
     >                   + u(i1+1,i2,i3)
     >                   + u1(i1))
     >           - a(2) * (u2(i1)
     >                    + u1(i1-1)
     >                    + u1(i1+1))
     >           - a(3) * (u2(i1-1)
     >                    + u2(i1+1))
          enddo
        enddo
      enddo
```

Fig. 1. An ASYNC_DO loop inside MG residual calculation subroutine

```
      z_low = (my_id * bz(k)) + 1
      z_high = (my_id + 1) * bz(k)
      if(my_id .eq. 0) z_low = 2
      if(my_id .eq. (total_threads - 1))
         z_high = n3 - 1
      do i3 = z_low, z_high
        do i2=2,n2-1
          do i1=1,n1
            u1(i1) = u(i1,i2-1,i3)
     >           + u(i1,i2+1,i3)
     >           + u(i1,i2,i3-1)
     >           + u(i1,i2,i3+1)
            u2(i1) = u(i1,i2-1,i3-1)
     >           + u(i1,i2+1,i3-1)
     >           + u(i1,i2-1,i3+1)
     >           + u(i1,i2+1,i3+1)
          enddo
          do i1=2,n1-1
            r(i1,i2,i3) = v(i1,i2,i3)
     >           - a(0) * u(i1,i2,i3)
     >           - a(1) * (u(i1-1,i2,i3)
     >                   + u(i1+1,i2,i3)
     >                   + u1(i1))
     >           - a(2) * (u2(i1)
     >                   + u1(i1-1)
     >                   + u1(i1+1))
     >           - a(3) * (u2(i1-1)
     >                   + u2(i1+1))
          enddo
        enddo
      enddo
```

Fig. 2. A synchronization-relaxed loop generated from ASYNC_DO annotation

to deposit its partial term of the reduction result in the tree and continue its execution without obtaining the newly computed value. Figure 3 shows an example with eight threads. Threads are numbered 0 through 7 and every dot represents a lock structure. Once a pair of threads arrive at the appropriate lock (if one gets there first, it waits for the other), the left-sibling thread proceeds up the tree with the new information deposited by both threads, and the other thread is released and allowed to continue execution. Using this structure, we can greatly reduce the amount of blocking time of threads.

The ASYNC_REDUCTION annotation is analogous to an OpenMP parallel DO loop with a reduction clause, but with the relaxed barrier tree replacing the strict barrier. Figure 4 shows an example of a loop annotated by ASYNC_REDUCTION. When embedded in an ASYNC_REGION accepted by

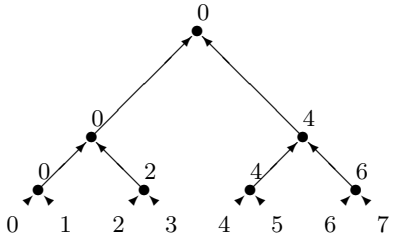


Fig. 3. A relaxed barrier tree structure

```
!$ASYNC_REDUCTION(+:d)
      do j=1, lastcol-firstcol+1
        d = d + p(j)*q(j)
      enddo
```

converted to

```
temp = 0.d0
do j = low_limit, high_limit
  temp = temp + p(j)*q(j)
enddo
call logbarrier(my_id, temp, d, 0)
```

Fig. 4. A loop converted from ASYNC_REDUCTION

the compiler, this loop will be transformed into an iteration-partitioned loop with a call to a runtime routine `logbarrier` which implements the relaxed barrier tree.

ASYNC_REGION. The ASYNC_REGION directive defines the lexical scope of the iterative solver and, by default, this scope has two barrier synchronization points, one at the entrance and the other at the exit. Within this scope, parallel threads partition the ASYNC_DO and ASYNC_REDUCTION loop iterations by following the "owner computes" rule such that, every time they reenter these loops, each thread will modify the same array sections as the last time.

Algorithm 1 provides a general template for writing an iterative solver using the asynchronous loops. The notation !$*a parallel loop header* in the algorithm could mean !$ASYNC_DO, !$ASYNC_REDUCTION, or any conventional OpenMP parallel construct. The iterative loop annotated by ASYNC_REGION

Algorithm 1. A General Template of Using ASYNC Loops

```
1: !$ASYNC_REGION
2: DO ITER = 1, number_iter
3: !$a parallel loop header
4: a parallel loop body
5: ...
6: !$a parallel loop header
7: a parallel loop body
8: ...
9: !$a parallel loop header
10: a parallel loop body
11: ...
12: END DO ITER
```

will be transformed by the compiler into an OpenMP parallel section (!$OMP parallel) to be executed by a number of parallel threads. Within the asynchronous region, an ASYNC_DO loop is transformed by the compiler into a DO loop whose iteration ranges are determined by the thread ID, as illustrated previously in Figure 2. No barrier synchronization is inserted. An ASYNC_REDUCTION loop is transformed into a DO loop that computes a partial reduction before invoking the relaxed barrier synchronization routine to add the partial term to the final result, as illustrated in Figure 4. A conventional OpenMP parallel `DO` or reduction loop will be transformed into a DO loop as stipulated in the OpenMP standard, with conventional barrier synchronization inserted. Sequential loops and statements within the ASYNC_REGION will be enclosed in a segment annotated by the `!$OMP master` directive to indicate that they are executed by the master thread only. At this point it should become clear to readers that one cannot implement asynchronous algorithms in OpenMP by simply adding the `nowait` label to a parallel `DO` loop.

It is important for the compiler to *align* the iteration ranges of the DO loops converted from the inner parallel loops (ASYNC or OpenMP) such that no two

threads write into the same array sections. This requires the compiler to perform array data flow analysis which has been studied quite extensively by previous work [8,10,11] and will be omitted in this paper due to space limitation. If the compiler is unable to perform the necessary array dataflow analysis for a particular asynchronous region, the ASYNC_REGION annotation will be ignored. An ASYNC_DO loop will be treated as an ordinary OpenMP parallel DO loop and an ASYNC_REDUCTION loop will be treated as an ordinary OpenMP reduction loop.

3 Benchmark Study

In this section, we introduce two benchmarks, namely MG and preconditioned CG, from the 2003 release of the NAS OpenMP parallel benchmarks (version 3.2) [1,9]. The MG program belongs to the class of iterative methods which use *relaxation methods* [12]. The CG program belongs to the class using *Krylov subspace methods* [12]. The Krylov subspace methods are known to benefit greatly from *preconditioning*, in terms of both numerical accuracy and computation efficiency.

Algorithm 2. Multigrid V-Cycle with Full Synchronization

1: DO iter = 1, number_iter
2: **for** $i = h_{max}...h_0, i = i/2$ **do**
3: !$OMP parallel do
4: Coarsen residual: $r^i = I^i_{i/2} r^{i/2}$
5: **end for**
6: !$OMP parallel do
7: Zero: $u^{h_0} = 0$
8: !$OMP parallel do
9: Smooth: $u^{h_0} = u^{h_0} + Sr^{h_0}$
10: **for** $i = 2...h_{max}/2, i = 2i$ **do**
11: !$OMP parallel do
12: Zero: $u^i = 0$
13: !$OMP parallel do
14: Prolongate: $u^i = I^{i/2}_i u^{i/2}$
15: !$OMP parallel do
16: Calculate Residual: $r^i = r^i - Au_i$
17: !$OMP parallel do
18: Smooth: $u^i = u^i + Sr^i$
19: **end for**
20: !$OMP parallel do
21: Prolongate: $u^{h_{max}} = I^{h_{max}/2}_{h_{max}} u^{h_{max}/2}$
22: !$OMP parallel do
23: Calculate Residual: $r^{h_{max}} = r^{h_{max}} - Au^{h_{max}}$
24: !$OMP parallel do
25: Smooth: $u^{h_{max}} = u^{h_{max}} + Sr^{h_{max}}$
26: END DO ITER

Relaxation methods such as SOR have been found to be effective preconditioners, We wrote a simple SOR preconditioner for CG.

MG Using ASYNC Loops. The MG program implements a multigrid method to solve the Poisson problem $\nabla u^2 = v$ with periodic boundary conditions. The benchmark places -1 and +1 values at twenty random grid points each, and zeros elsewhere. Each iteration consists of a full V-cycle [12]. We also derived a two-dimensional version of MG for a 2D grid size, but the algorithm remained the same. The high level organization of MG in OpenMP can be illustrated by the code skeleton in Algorithm 2. In the actual OpenMP code, the `!$OMP` annotations are within the subroutines shown in the algorithm such that within each subroutine call, one or more !$OMP parallel DO loops are executed, forcing a strict data flow.

To relax the data flow constraints, we annotate the entire iterative solver by ASYNC_REGION (which will then be converted to an OpenMP parallel section as discussed previously, suppose we decide that the asynchronous model is

Algorithm 3. Multigrid V-Cycle with ASYNC_DO loops

```
1: !$ASYNC_REGION
2: DO iter = 1, number_iter
3: for i = h_max...h_0, i = i/2 do
4:    !$ASYNC_DO
5:    Coarsen residual: r^i = I^i_{i/2} r^{i/2}
6: end for
7: !$ASYNC_DO
8: Zero: u^{h_0} = 0
9: !$ASYNC_DO
10: Smooth: u^{h_0} = u^{h_0} + S r^{h_0}
11: for i = 2...h_max/2, i = 2i do
12:    !$ASYNC_DO
13:    Zero: u^i = 0
14:    !$ASYNC_DO
15:    Prolongate: u^i = I_i^{i/2} u^{i/2}
16:    !$Barrier
17:    !$ASYNC_DO
18:    Calculate Residual: r^i = r^i - A u_i
19:    !$ASYNC_DO
20:    Smooth: u^i = u^i + S r^i
21: end for
22: !$ASYNC_DO
23: Prolongate: u^{h_max} = I^{h_max/2}_{h_max} u^{h_max/2}
24: !$Barrier
25: !$ASYNC_DO
26: Calculate Residual: r^{h_max} = r^{h_max} - A u^{h_max}
27: !$ASYNC_DO
28: Smooth: u^{h_max} = u^{h_max} + S r^{h_max}
29: END DO ITER
```

beneficial). A careful analysis of the numerical property of the solver suggests that all but two of the synchronization points can be safely removed without severely slowing the convergence. The necessary synchronization points occur immediately after the prolongation operation, but before the residual calculation. Hence, we change all embedded OpenMP parallel DO loops to ASYNC_DO loops. As discussed previously, the ASYNC_DO loops will result in a partition of the parallel loop iterations, but without implied barrier synchronization. We reinsert barriers immediately before the residual calculation. Figures 1 and 2 (in Section 2) show details for the residual calculation loop. The high-level organization of the ASYNC_REGION is shown in the code skeleton in Algorithm 3 below.

The reinserted synchronization points are necessary because, if stale values are used from the interpolated grid, the residual will undoubtedly be higher. Once this higher residual is applied to correct the grid, the difference (in norm) of the current grid to the previous one will also be larger. This error will propagate through each V-cycle iteration, so that the residual will continue to grow indefinitely after a few iterations. The smoothing operation exists to even out sharp differences in the residual, but the effects of a larger magnitude of residual values in general will still exist. Our experiments without the reinserted barriers have turned out poor numerical accuracies and hence confirmed the necessity of these synchronization points.

3.1 SOR-Preconditioned CG Using ASYNC Loops

A *preconditioned* conjugate gradient method is given in Algorithm 4 below [12]. Preconditioning a system of linear equations is a way to transform the original system into one that is likely to be easier to solve with an iterative solver. Both the efficiency and robustness of iterative techniques can be improved by a good preconditioner. The number of iterations to execute CG are expected to be reduced after preconditioning. The preconditoner M (in steps 1 and 6) is chosen such that M^{-1} is a good approximation of A^{-1}. Also, the system $Mz = r$ needs to be much easier to solve than the original system $Ax = b$, for example, using Jacobi, Gauss-Seidel, or Successive Overrelaxation. Here, we apply the preconditioner M to the system $Ax = b$ from the left, i.e. $M^{-1}Ax = M^{-1}b$.

Algorithm 4. Preconditioned Conjugate Gradient

1: Compute $r_0 = b - Ax_0$, $z_0 = M^{-1}r_0$, $p_0 = z_0$
2: **for** $j = 0, 1, ...$ until convergence **do**
3: $\quad \alpha_j = (r_j, z_j)/(Ap_j, p_j)$
4: $\quad x_{j+1} = x_j + \alpha_j p_j$
5: $\quad r_{j+1} = r_j - \alpha_j Ap_j$
6: $\quad z_{j+1} = M^{-1}r_{j+1}$
7: $\quad \beta_j = (r_{j+1}, z_{j+1})/(r_j, z_j)$
8: $\quad p_{j+1} = z_{j+1} + \beta_j p_j$
9: **end for**

Notice in statement 6 of algorithm 4, we compute $z = M^{-1}r = M^{-1}(b - Ax) = M^{-1}b - M^{-1}Ax$. Thus z represents the residual of the transformed system.

The CG benchmark from NAS estimates the smallest eigenvalue in magnitude of a matrix A using the inverse power method with shifts. During each iteration, a solution of a system of the form $Ax = b$ is obtained by calling a CG subroutine. Because here the CG method is being used as a solver inside of another iterative method, it is invoked a fixed number of times. The input matrix is of dimension 14000, and the conjugate gradient method is stopped when the residual norm falls below 10^{-8}. We implemented a point successive overrelaxation (SOR) scheme to serve as a preconditioner for CG. Because the structure of the input matrix is symmetric positive definite but random, the point SOR solver is an appropriate preconditioner. The preconditioner is stopped when the residual norm is less than 10^{-6}.

4 Experimental Results

We performed experiments by running the chosen benchmarks on a Sun E10000 which has 54 Ultrasparc II processors (each clocked at 400 Megahertz) and 56 GB

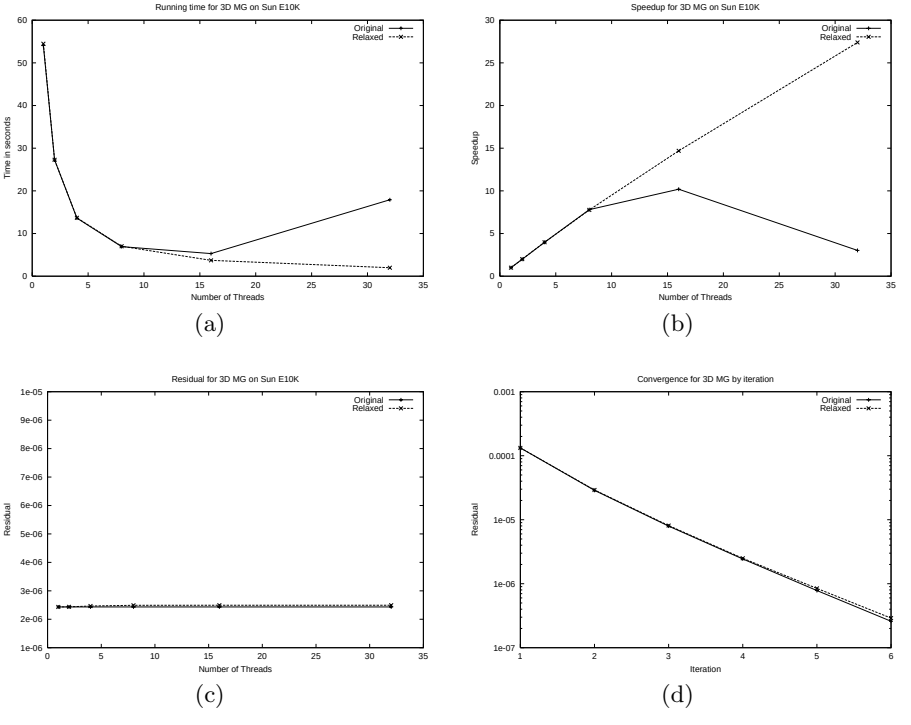


Fig. 5. Comparison between strict and relaxed data flow models with MG. (a) Running time using standard 3D grid (b) Parallel speedups (c) Final residual norms (d) Convergence Rates.

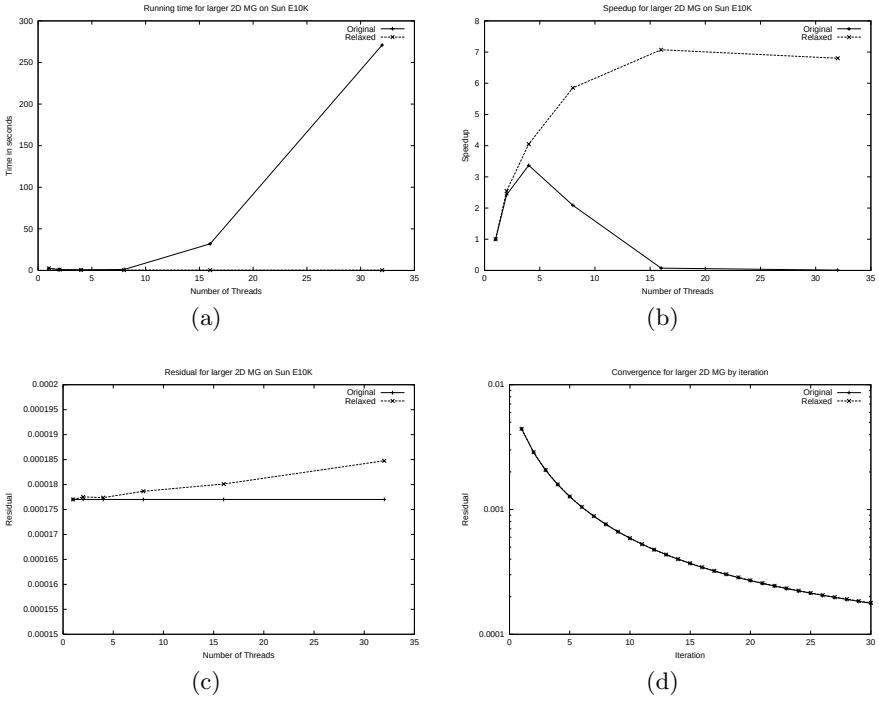


Fig. 6. (a) Running time of MG using a 2D grid (b) Parallel speedup (c) Final residual norms (d) Convergence rates

of memory. For each data set, tests were run on a differing number of processors, ranging from 1 to 32 in powers of two. With the exception of the residual vs. the number of iterations, all performance data reported here is an average over 10 runs. Since we are still exploring the range of applications for ASYNC loops, the conversion from such loops to OpenMP codes is currently performed manually.

3D Multigrid. We first use a 256 x 256 x 256 grid size with 4 iterations for the MG benchmark, in order to be consistent with the original benchmark specification for class A problems. Figures 5(a)-5(c) compare the performance and residual after executing four iterations, while figure 5(d) shows that both versions approach convergence at the same rate in the number of iterations shown.

After just four iterations, a satisfactory residual has been reached. We see that the speedup behavior of the relaxed version is superior to that of the original OpenMP version, in particular with more than eight threads. In addition, although the final residual values are slightly higher than that of the original version, the residuals for the relaxed version are approximately of the same order of magnitude in comparison.

2D Multigrid. To further study the performance, we use a 512 x 512 grid size for the two dimensional problem. Figures 6(a)-6(c) compare the performance and

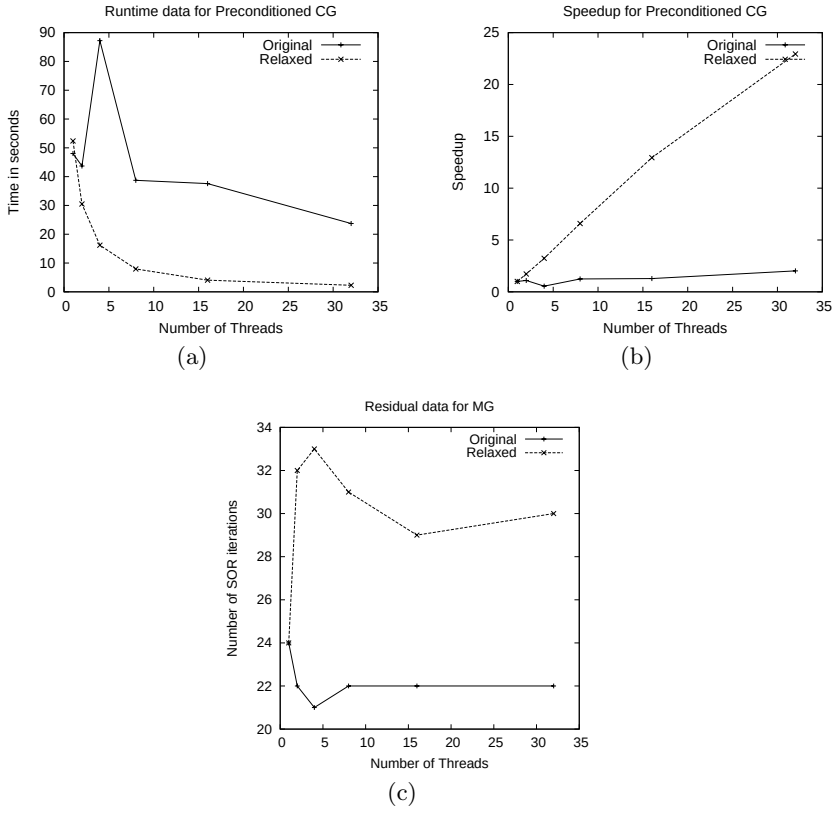


Fig. 7. Comparison of methods with SOR-preconditioned CG (a) Running times (b) Parallel speedups (c) Number of iterations of first invocation of SOR preconditioner

residual after executing 30 iterations, and figure 6(d) shows that both versions approach convergence at the same rate in the number of iterations given. The results also show that the original version performs quite poorly in terms of parallel speedup. The relaxed version clearly performs better.

SOR-Preconditioned CG. We tested the use of SOR as a preconditioner embedded in the basic conjugate gradient method (see Algorithm 4). The SOR preconditioner was run both in its original OpenMP version with strict data flow and in the ASYNC_DO version with relaxed data flow. Figures 7(a) and 7(b) show that the parallel speedups of the relaxed version are much improved over the original version. It is not yet clear why the original version experiences a jump in the running time for four threads. One might observe that the relaxed data flow causes the preconditioner to converge in a greater number of iterations, as figure 7(c) demonstrates. Even as such, the improvement in efficiency over the original version is quite significant.

5 Related Work

To the best of our knowledge, this work is the first to propose parallel language constructs to identify loops for execution based on asynchronous algorithms. Although the asynchronous model could be implemented using P-threads or existing OpenMP directives, it would not give the same flexility to the compiler as our scheme does, and the way the program is composed would be more tedious and more error-prone to modify. A considerable amount of prior work, on the other hand, has been conducted on theories of asynchronous iterative algorithms in the past decades [3,4,7]. Most publications seem to have focused on developing general convergence criteria and tightening convergence conditions, although several have devoted themselves to specific iterative methods such as Jacobi, Gauss-Seidel and SOR [2,3,6]. We have not found prior experimentation with mutigrid methods using asynchronous algorithms, which we have presented in this paper. Applying the general theory of asynchronous computation model to a concrete iterative numerical method remains a nontrivial problem.

6 Conclusion

The number of processors in parallel computers have been steadily increased in recent years. The largest computational clusters now boast over ten thousand processors. Interprocessor data communication is therefore becoming a more serious performance bottleneck. We have proposed three kinds of ASYNC loop constructs to support the asynchronous computation model for iterative solvers, which, when applied successfully, can significantly reduce data communication overhead. The experimental results with 3D and 2D MG benchmarks and with SOR-preconditioned CG benchmark show excellent improvement of the ASYNC versions over the conventional OpenMP versions of these parallel programs in terms of the parallel execution efficiency. Moreover, the convergence rate has remained approximately the same. While these results are highly encouraging, we observe that deciding which synchronization points to remove remain a nontrivial task which involves careful consideration of the numerical properties of the given iterative solver. We believe that the proposed new loop constructs make it easier for programmers to implement and fine tune asynchronous algorithms.

For our future work, we will further investigate other asynchronous algorithms and hope to see sufficient successes to motivate a full implementation of the proposed ASYNC loops in a parallelizing compiler.

Acknowledgement

This work is sponsored in part by National Science Foundation through grants ST-HEC-0444285 and CPA-0702245. The authors thank the reviewers for their careful reviews and helpful suggestions. We also thank Ananth Grama for proposing the use of relaxed barrier tree for reduction.

References

1. Bailey, D., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Fineberg, S., Fredrickson, P., Lasinski, T., Schreiber, R., Simon, H., Venkatakrishnan, V., Weerantunga, S.: The NAS Parallel Benchmarks. Technical Report RNR-94-007, NASA (March 1994)
2. Barlow, R.H., Evans, D.J.: Parallel algorithms for the iterative solution to linear systems. The Computer Journal 25(1), 56–60 (1982)
3. Baudet, G.M.: Asynchronous iterative methods for multiprocessors. J. ACM 25(2), 226–244 (1978)
4. Bertsekas, D.P., Tsitsiklis, J.N.: Convergence rate and termination of asynchronous iterative algorithms. In: ICS 1989: Proceedings of the 3rd international conference on Supercomputing, pp. 461–470. ACM, New York (1989)
5. OpenMP Architecture Review Board. OpenMP Application Program Interface. 2.5 edition (May 1990)
6. Bru, R., Migallón, V., Penadés, J., Szyld, D.B.: Parallel, Synchronous and Asynchronous Two-stage Multisplitting Methods. Electronic Transactions on Numerical Analysis 3, 24–38 (1995)
7. Chazan, D., Miranker, W.L.: Chaotic relaxation. Linear Algebra and Its Application 2, 199–222 (1969)
8. Gu, J., Li, Z.: Efficient interprocedural array data-flow analysis for automatic program parallelization. IEEE Trans. on Software Engineering 26, 244–261 (2000)
9. Jin, H., Frumkin, M., Yan, J.: The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. Technical Report NAS-99-011, NASA (October 1999)
10. Moon, S., Hall, M.W.: Evaluation of predicated array data-flow analysis for automatic parallelization. SIGPLAN Not. 34(8), 84–95 (1999)
11. Moon, S., Hall, M.W., Murphy, B.R.: Predicated array data-flow analysis for runtime parallelization. In: International Conference on Supercomputing, pp. 204–211 (1998)
12. Saad, Y.: Iterative Methods for Sparse Linear Systems, 2nd edn. SIAM, Philadelphia (2003)

Design for Interoperability in STAPL: pMatrices and Linear Algebra Algorithms*

Antal A. Buss, Timmie G. Smith, Gabriel Tanase, Nathan L. Thomas, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger

Parasol Lab, Dept. of Computer Science, Texas A&M University
{abuss,timmie,gabrielt,nthomas,bmm,amato,rwerger}@cs.tamu.edu

Abstract. The Standard Template Adaptive Parallel Library (STAPL) is a high-productivity parallel programming framework that extends C++ and STL with unified support for shared and distributed memory parallelism. STAPL provides distributed data structures (`pContainers`) and parallel algorithms (`pAlgorithms`) and a generic methodology for extending them to provide customized functionality. To improve productivity and performance, it is essential for STAPL to exploit third party libraries, including those developed in programming languages other than C++. In this paper we describe a methodology that enables third party libraries to be used with STAPL. This methodology allows a developer to specify when these specialized libraries can correctly be used, and provides mechanisms to transparently invoke them when appropriate. It also provides support for using STAPL `pAlgorithms` and `pContainers` in external codes. As a concrete example, we illustrate how third party libraries, namely BLAS and PBLAS, can be transparently embedded into STAPL to provide efficient linear algebra algorithms for the STAPL `pMatrix`, with negligible slowdown with respect to the optimized libraries themselves.

1 Introduction

Parallel programming is becoming mainstream due to the increased availability of multiprocessor and multicore architectures and the need to solve larger and more complex problems. To help programmers address the difficulties of parallel programming, we are developing the Standard Template Adaptive Parallel Library (STAPL) [1,15,16,18]. STAPL is a parallel C++ library with functionality similar to STL, the ANSI adopted C++ Standard Template Library [14] that provides a collection of basic algorithms, containers and iterators that can be used as high-level building blocks for sequential applications.

* This research supported in part by NSF Grants EIA-0103742, ACR-0081510, ACR-0113971, CCR-0113974, EIA-9810937, ACI-0326350, CRI-0551685, CNS-0615267, CCF 0702765, by the DOE and Intel. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. Buss is supported in part by Colciencias/LASPAU (Fulbright) Fellowship.

J.N. Amaral (Ed.): LCPC 2008, LNCS 5335, pp. 304–315, 2008.

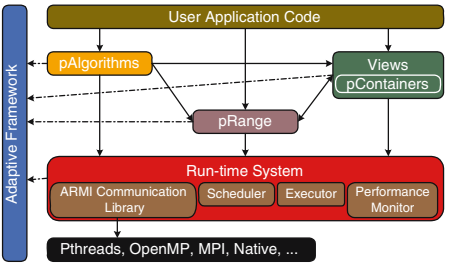


Fig. 1. STAPL software architecture

STAPL consists of a set of components that include `pContainers`, `pAlgorithms`, `views`, `pRanges`, and a run-time system (see Figure 1). `pContainers`, the distributed counterpart of STL containers, are provided to the users as shared memory, thread-safe, concurrent, extendable, and composable objects. `pContainer` data can be accessed using `views` which can be seen as generalizations of STL iterators that represent sets of data elements and are not necessarily related to the data's physical location, e.g., a row-major view of a matrix that is stored in column major order. Generic parallel algorithms (`pAlgorithms`) are written in terms of `views`, similar to how STL algorithms are written in terms of iterators. Intuitively, `pAlgorithms` are expressed as task graphs (called `pRanges`), where each task consists of a work function and `views` representing the data on which the work function will be applied. STAPL relies on the run-time system (RTS) and its communication library ARMI (Adaptive Remote Method Invocation [17]) to abstract the low-level hardware details of the specific architectures.

An important goal of STAPL is to provide a high productivity environment for developing applications that can execute efficiently on a wide spectrum of parallel and distributed systems. A key requirement for this is that STAPL must be interoperable with third party libraries and programs. First, STAPL programs must be able to take advantage of well known, trusted, highly optimized external libraries such as BLAS [13], PBLAS [5,6], LAPACK [2], parMETIS [12], etc. Second, it is equally important for a programming tool like STAPL to provide the ability to be used by other packages, including programs written in other languages (e.g., FORTRAN). For example, STAPL must provide interfaces to make `pAlgorithms` callable on third party data structures, such as FORTRAN/MPI distributed matrices. In this paper, we present a methodology for making STAPL interoperable with external libraries and programs. We describe how to *specialize* `pAlgorithms` to use third party libraries, and how to *invoke* `pAlgorithms` from other programming languages. Our strategy exploits unique features of `pContainers` and `views` that provide generic access to data and of `pAlgorithms` that allows them to be specialized to use optimized third party libraries when possible.

Since `pAlgorithms` are defined in terms of `views`, the same `pAlgorithm` can be used with multiple `pContainers` each with arbitrary physical distributions.

However, the genericity provided by `views` can complicate the use of third party libraries. For example, the PBLAS library [5,6] for matrix multiplication requires that the data be of a PBLAS recognized type (e.g., `float`) and laid out in local memory in contiguous chunks and in a block-cyclic manner across the processes executing the parallel application. Thus, a generic matrix multiplication `pAlgorithm` can only be specialized to use PBLAS if the data corresponding to the `pAlgorithm` input `views` can be shown to already satisfy these properties, or can be efficiently converted to do so. Our methodology for optimizing `pAlgorithm` performance is designed to address these problems. We do this by providing algorithm developers the tools to describe the problem input conditions under which tuned external libraries can legally be called and the mechanisms to properly invoke them.

We illustrate the process by showing how third party libraries, namely BLAS and PBLAS, can be transparently embedded into STAPL to provide efficient `pAlgorithms` for the `pMatrix`, a `pContainer` providing two-dimensional random access dense arrays with customizable data distributions. We also describe how `pAlgorithms` can be used by external codes. Our results on two architectures, a 640 processor IBM RS/6000 with dual Power5 processors and a 19,320 processor Cray XT4, show that generic `pAlgorithms` operating on `pMatrices` can be: (i) specialized to exploit BLAS and PBLAS and provide performance comparable to the optimized libraries themselves, and (ii) used by external codes with minimal overhead.

2 Related Work

The interoperability of software components and libraries is a broad field of study in software engineering. For the purposes of this paper, we will focus our discussion of related work to research covering library integration for high performance computing as well as recent work for library development and composition done within the context of generic programming in C++.

Breuer et al. [3] employ the current generic programming mechanisms in C++ to invoke external eigensolver routines on their distributed graph data structure. Specifically, their graph is mapped to the eigensolver's matrix concept via an adapter code module.

In [7], Edjlali et al. present Meta-Chaos to address the interoperability problem in a parallel environment. The approach is to define an application independent, linearized data layout that the various software components must use. It is useful for cases where the expected and actual layout of a data structure are very different.

Jarvi et al. [10] adapt the flood-fill algorithm in Adobe's generic image library to use the Boost graph library's sequential graph algorithms. This work uses a compiler that implements a prototype of the C++0x concept proposal [9]. They show that with minimal changes to either existing code base, concepts can adapt data structures from the image library to reuse generic Boost graph algorithms. This static (i.e., compile-time) adaptation incurs no runtime overhead.

Many active parallel language and library projects list interoperability as a key design goal, and some give the specification of calling conventions for the invocation of mainstream language (e.g., FORTRAN) libraries. However, it does not appear that much development effort has yet been spent demonstrating how the various projects' constructs can be used to promote code reuse and interoperability.

3 The pMatrix pContainer

In this section we briefly describe the `pMatrix`, a `pContainer` that implements a two-dimensional dense array. We concentrate on the aspects necessary to present our interoperability methodology. More details about the `pContainer` framework can be found in [15,16].

The declaration of a `pMatrix` requires the following template arguments:
`template <VType,Major=Row,BlocksMajor=Row,Partition=Default,Traits =Default > class p_matrix;`
The `VType` defines the type of elements stored in a `pMatrix`. A `pMatrix` is partitioned into sub-matrices (`components`) as specified by a `partition` class that defines in which sub-matrix each element of the `pMatrix` is stored. A total order over the elements of the matrix is defined by the `Major` and `BlocksMajor` types which specify the order among and within, respectively, the sub-matrices; the majors can be `Row` or `Column`.

Each sub-matrix is stored in a *location*, which is assumed to have execution capabilities, e.g., a location can be identified with a process address space. The mapping between sub-matrices and locations is performed by a `partition-mapper` (defined in the `Traits` template argument). Hence, together the `partition` and `partition-mapper` define a data distribution in STAPL. This can define a location/processor grid, as in [5]. For the `pMatrix`, we provide a generic *block cyclic* data distribution which can be customized to obtain other `partitions`, such as *blocked*, *blocked row-wise*, or *blocked column-wise*.

The `pContainer` framework is designed to allow the reuse of existing containers. This is supported by a component interface that can be implemented by `pContainer` developers as a light wrapper around their existing containers. For example, for `pMatrix` we support sub-matrices implemented as MTL [8], Blitz++ [19], or `malloc` allocated buffers.

The interface of the sub-matrix component must provide methods to allocate and manipulate the data and iterators to *natively* access the elements, which enables optimized data access for specialized `pAlgorithms`. Another interface requirement is to flag if the iteration order provided by the sub-matrix component iterators is the same as iterating through memory (e.g., `is_contiguously_stored =true/false`). This is necessary, for example, when the data of the sub-matrix component has to be passed to external libraries like BLAS.

A `view` over a `pMatrix` is a two-dimensional random access array typically representing an arbitrary sub-matrix of the `pMatrix`. A `view` can be partitioned into `sub-views` using a `partition` and a `partition-mapper`. A `sub-view` is

in every respect a `view`. The *default* `view` provided by a `pMatrix` matches the partition and the mapping of the `pMatrix` data. This view provides the most efficient data access since all the elements in a `sub-view` are in the same physical location. Starting from the default `view`, the user can obtain other `views`, such as `views` over a sub-matrix, `views` over rows (columns), `views` that provide the transpose, or `views` with an arbitrary block cyclic partition.

4 pAlgorithms

`pAlgorithms` in STAPL are specified as *task graphs*, where tasks are function objects (called *workfunctions*) that operate on `views` passed to them as arguments. Typically the workfunctions express the computation as operations on `iterators` over views. STAPL provides parallel versions of the STL algorithms, `pMatrix` algorithms, and `pGraph` algorithms, that operate on linearized `views`, `pMatrix` `views` and `pGraph` `views`, respectively.

Once the workfunction has been specified, the views passed as arguments may have to be rearranged to match the algorithmic requirements. This step is called `view` *alignment*. The workfunctions have *traits* that can specify requirements of the algorithm, e.g., the return type of the work function, the reduction operator to be used on it, the way the input `views` are accessed, such as read-only (`R`), write-only (`W`), or read-write (`RW`), etc. The latter traits are used by STAPL to allocate tasks to locations to improve performance.

STAPL provides support for querying and exploiting locality of `views`. Since a `view` describes a logical layout and partition of the data of a `pContainer`, accessing a data element through a `view` can involve a remote memory access which, in general, adds some overhead, even to local accesses. To mitigate this potential performance loss, STAPL can verify if a given `view` (i) is completely contained in a single address space, and (ii) if its iteration space is the same as the `pContainer`'s layout. Based on what conditions are matched, accesses to the data in the `views` can be optimized. For this reason, the `views` that are actually passed to the workfunctions may be of a type different than the one originally specified. For instance, it may be a `local_view`, whose data is in the local address space. This mechanism is exploited by the specializations described in the next section.

We now provide a `pAlgorithm` for matrix-matrix multiplication called `p_matrix_multiply_general` that given a view of an $m \times k$ matrix `A` and one of a $k \times n$ matrix `B`, computes $A \times B$ in the view of an $m \times n$ matrix C. The algorithm rearranges the input views corresponding to A, B, and C as blocked matrices, where blocks are compatible for matrix multiplication and accessed in column-major. The tasks of the algorithm perform the block-by-block multiplications necessary to compute the final result. A sketch of the workfunction is provided in Figure 2. The triple loop algorithm is written to access the views according to the view's major to exploit possible locality/access opportunities.

Note the `typedefs` in the class public interface that define the type of access performed in the `views`. Each `view` in the argument list has a corresponding

```
struct mat_mult_wf {
  typedef void result_type; // workfunction returns void
  // vA and vB are read-only, vC is write-only
  typedef access_list<R,R,W> view_access_types;

  template<class ViewA, class ViewB, class ViewC>
  void operator()(ViewA& vA, ViewB& vB, ViewC& vC) {

    // Triple loop rearranged to exploit possible locality
    for(size_t j = 0; j < cols_of_B; ++j)
      for(size_t k = 0; k < rows_of_B; ++k)
        for(size_t i = 0; i < rows_of_C; ++i)
          vC(i,j) += vA(i,k) * vB(k,j);
  }
};
```

Fig. 2. Example of workfunction for `p_mat_mul` algorithm showing traits for return type and access type of arguments (`R` for read-only, `W` for write only), and the templated function operator

flag indicating if it is read-only (`R`), write only (`W`), or read-write (`RW`). These access specifications are used to decide where the tasks will be executed, how the data will be accessed during the execution of the algorithm, and what types of optimizations can be applied.

5 Interoperability for Linear Algebra Computations

In this section, we describe our methodology for interoperability between STAPL's `pAlgorithms` and other libraries. Here, we apply it to parallel matrix multiplication; however, the approach is general and indicative of how other code bases interact with STAPL. We first look at how `p_matrix_multiply` transparently invokes PBLAS and BLAS routines when a set of compile time and runtime constraints are satisfied. We show how these constraints are specified and subsequently enforced. Finally, we show how an application not written in STAPL can invoke `p_matrix_multiply` through an interface provided by the `pMatrix`.

Algorithm 1. p_matrix_multiply(A, B, C)

```
1. if input conforms to PBLAS then
2.    call PBLAS
3. else
4.    decision =  redistribute | general
5.    if decision == redistribute then
6.       p_copy A,B,C to temporary PBLAS conformable storage
7.       call PBLAS
8.       p_copy temporary result to C
9.    else
10.      call general_matrix_multiply (use BLAS in sequential sections if possible)
11.   end if
12. end if
```

```
template<typename ViewMatA,typename ViewMatB,typename ViewMatC> void
p_matrix_multiply(ViewMatA& vA, ViewMatB& vB, ViewMatC& vC) {

  algorithm_impl::pdblas_conformable<ViewMatA, ViewMatB, ViewMatC> check_conformability;
  if (check_conformability<COLUMN_MAJOR, COLUMN_MAJOR, COLUMN_MAJOR>(vA, vB, vC) ||
      check_conformability<COLUMN_MAJOR, ROW_MAJOR, COLUMN_MAJOR>(vA, vB, vC) {
    p_matrix_mult_pblas(vA, vB, vC);
  } else {
    p_matrix_multiply_general(vA,vB,vC);
  }
}
```

Fig. 3. Specializing the p_matrix_multiply algorithm

5.1 Optimizing `p_matrix_multiply` with PBLAS and BLAS

Algorithm 1 shows pseudocode for `p_matrix_multiply`. First, the input is tested to determine if PBLAS can be called. If it cannot, the algorithm may employ an approach outlined in [18] to temporarily redistribute the data so that it is amenable to PBLAS invocation. This has been shown [6] to often be the best approach (assuming memory is available). Otherwise, STAPL's `p_matrix_multiply_general`, described in the previous section, is invoked and BLAS is used in serialized sections of the computation when the input conforms to BLAS interfaces.

Specializing the Parallel Computation. For brevity, we only show the test for invoking `pdgemm`, the PBLAS routine for double precision data. Specializations for other types follow the same approach. The conditions to use `pdgemm` are partially tested at compile time and partially at runtime. The conditions are the following: 1) the type of the data elements has to be `double`, 2) the input `views`, after aligning, have the property that each `sub-view` is contained in a single address space, 3) the `partition` of the `pMatrices` has to be block-cyclic and the majors of the blocks and within the blocks have to be the same, 4) the `partition-mappers` define a common computing grid for all the matrices, and 5) that the block parameters of the distributions make the three matrices distributions compatible with what `pdgemm` requires. The latter conditions depend on the type of the major of the matrices, and a proper invocation of `pdgemm` has to be picked up to cover the eight combinations of the transposition flags for A, B, and C.

Figure 3 shows the specialization of `p_matrix_multiply` to invoke PBLAS. The code shows only the condition checking for two specializations, one when the input `pMatrices` will be passed as-is to `pdgemm`, and another when the transposition flag for matrix B has to be set appropriately since B is stored row-major and `pdgemm` accepts by default column-major. `check_comformability` checks all the conditions listed above. The template parameters are used to select the proper set of checks for condition 5.

Specializing the Sequential Computation. For BLAS, we use a different approach than PBLAS for algorithm customization. There is still a runtime constraint for BLAS conformability. It shares the requirement with PBLAS that the subviews refer only to elements on the execution location (contiguous storage and traversal sequence requirements are checked statically as shown below).

The runtime check, however, is not explicitly located within the sequential workfunction but instead is implicitly performed by STAPL before each workfunction invocation. The workfunction invocation can determine the result of this test by checking whether the types of the views passed to it are `local_views` (see Section 4). This approach employs additional C++ language constructs to provide a more structured approach to incremental algorithm specialization. It allows new cases to be added over time without the need to modify existing code.

This relies on C++'s class template partial specialization mechanisms and the `enable_if` [11] template utility. Partial specialization allows one to define a *primary template* (i.e., the general matrix multiplication algorithm) which is used unless a template specialization is defined which is a better fit for the given template arguments (i.e., view types). `enable_if` allows us to specify when a specialization is appropriate to use based on *type traits*, by exploiting the *Substitution failure is not an error* (SFINAE) condition in C++.

```
struct matmul_wf {
  //function operator invoked by STAPL task graph
  template<typename VA, typename VB, typename VC>
  void operator()(VA& vA, VB& vB, VC& vC) {
    mat_mult_algorithm<VA, VB, VC> algorithm;
    algorithm(vA, vB, vC);
  }

  //Define nested class template struct mat_mult_algorithm
  //and specialize behavior as desired.

  //The generic (default) algorithm
  template<typename VA, typename VB, typename VC, typename Enable = void>
  struct mat_mult_algorithm {
    void operator()(VA& vA, VB& vB, VC& vC)
    { ... } // General algorithm
  };

  //1st specialization
  template<typename VA, typename VB, typename VC>
  struct mat_mult_algorithm<VA, VB, VC,
    typename enable_if<and_<dblas_capable_view_set<VA, VB, VC>
                            column_major<VA>, native_traveral<VA>,
                            column_major<VB>, native_traveral<VB>,
                            column_major<VC>, native_traveral<VC>
                          > >::type > {
    void operator()(VA& vA, VB& vB, VC& vC)
    { ... } // Setup matrix descriptors and CALL BLAS dgemm
  };

  //2nd specialization
  template<typename VA, typename VB, typename VC>
  struct mat_mult_algorithm<VA, VB, VC,
    typename enable_if<and_<dblas_capable_view_set<VA, VB, VC>
                            column_major<VA>, native_traveral<VA>,
                            column_major<VB>, transposed_native_traveral<VB>,
                            column_major<VC>, native_traveral<VC>
                          > >::type > {
    void operator()(VA& vA, VB& vB, VC& vC)
    { ... } // Setup matrix descriptors and CALL BLAS dgemm
  };
};
```

Fig. 4. Specializing the matrix multiplication workfunction to use BLAS

Note that our workfunction's function operator forwards the input views to a nested class template for execution. This is because only class templates (and not function templates) support partial specialization in C++. As with PBLAS specialization, the approach used to specialize the sequential matrix multiplication will also benefit from new concept features likely to be part of the next language standard.

Figure 4 shows the pseudo-code for BLAS specialization. `dblas_capable_view_set` checks, at compile time, if data is local and contiguous in memory. BLAS assumes matrices are stored in a column-major fashion. Hence, with these guarantees, it can be invoked directly with its standard parameters. The second specialization is similar, but handles the case when a column-major view of B represents a transposed traversal over the container's native order (i.e., the container is row-major). Here, the transposition flag must be set for matrix B when invoking the BLAS `dgemm` routine.

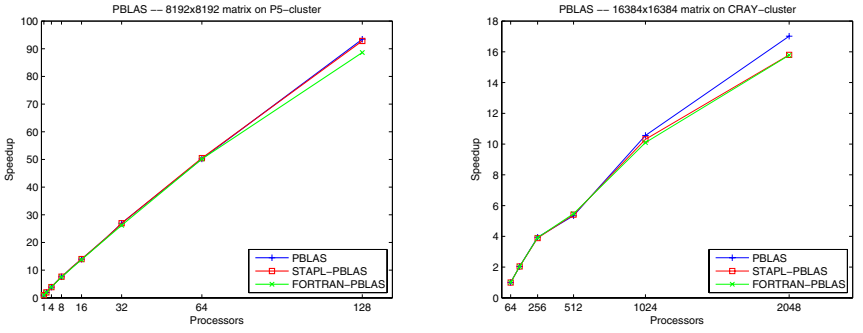
5.2 External Invocation of p_matrix_multiply

STAPL supports the use of `pAlgorithms` by applications that have been developed outside STAPL by providing a wrapper function for each `pAlgorithm`. The wrapper accepts pointers to the calling program's data instead of the `views` required by the `pAlgorithm` interface, and then constructs a `pContainer` for each argument using a special constructor that explicitly takes pre-allocated memory. The `views` obtained from these `pContainers` are then passed to the STAPL `pAlgorithm`. The `pAlgorithm` transparently accesses the external data through the `view` interface. When the `pAlgorithm` returns, the destructors of the `pContainers` do not attempt to free the memory since they are aware that it is externally managed, and control is passed back to the calling application.

6 Experimental Results

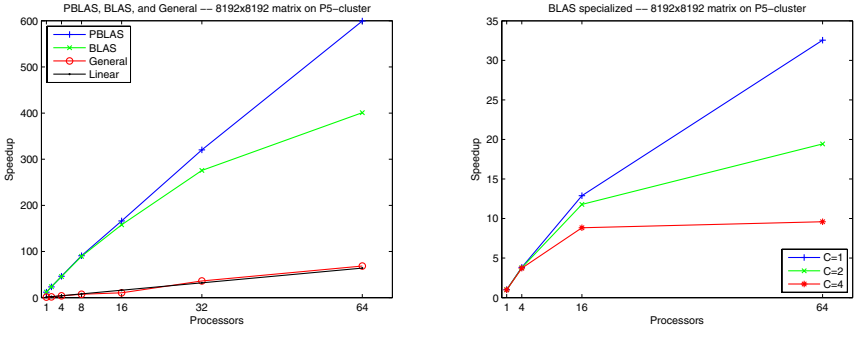
In this section, we present experimental results to evaluate the performance of our specialization methodology. We run experiments on two different architectures: a 640 processor IBM RS/6000 with dual Power5 processors available at Texas A&M University (called P5-CLUSTER), and a 19,320 processors Cray XT4 at NERSC (called CRAY-CLUSTER). More extensive results are available in [4].

The first set of experiments compares the performance of the STAPL `p_matrix_multiply` algorithm in the case where PBLAS specialization can be used (STAPL-PBLAS line) with a direct invocation of PBLAS (PBLAS line). We also show the performance of a FORTRAN/MPI program that allocates the data and invokes the STAPL matrix multiplication algorithm (FORTRAN-STAPL). The results are shown in Figure 5(a) for P5-CLUSTER, and Figure 5(b) for CRAY-CLUSTER. The plots show the speed-up with respect to running PBLAS sequentially on P5-CLUSTER, and with respect to 64 processors on CRAY-CLUSTER (to fit the large input in memory). The plots show that the overhead of the runtime check to determine if the PBLAS specialization can be invoked is negligible



(a) (b)

Fig. 5. Comparison of speed-ups for a direct PBLAS invocation (PBLAS), a PBLAS specialized STAPL `pAlgorithm` (STAPL-PBLAS), and a FORTRAN/MPI program invoking a STAPL `pAlgorithm` (FORTRAN-PBLAS). Results are shown for two architectures/data sizes: (a) P5-CLUSTER with 8192×8192 matrices and (b) CRAY-CLUSTER with 16384×16384 matrices. The baseline is direct PBLAS.



(a) (b)

Fig. 6. P5-CLUSTER with 8192×8192 matrices: (a) Speedups of the unspecialized (General) and BLAS specialized (BLAS) versions of the algorithm, and (b) speed-ups of the BLAS specialized workfunction when varying the number of `components` per location

up to a thousand processors. For larger processor counts, for which the actual multiplication takes less than a second, STAPL exhibits a visible overhead. The knee in Figure 5(b) arises since `pdgemm` does not scale as well if the number of processors is not a perfect square.

As mentioned in Section 3, the partition of a `pContainer` is decoupled from its actual distribution across the address spaces (locations) of the processes carrying out the computation (done by the `partition-mapper`). Thus, more than one `pContainer component` can be placed into a single address space. Since PBLAS forbids the use of such a data layout, when the number of sub-matrices is greater than the number of locations, then the runtime check for using PBLAS fails and the specialization for using BLAS within the workfunction is then tested.

Next we analyze the performance of the BLAS specialization. Figure 6(a) shows the speed-ups of the STAPL algorithm when no specialization is used (General line) and when the BLAS specialization is used (BLAS line). The baseline is the execution time of the matrix-matrix multiplication algorithm implemented in the workfunction of Figure 2. The performance of the unspecialized algorithm is dramatically slower than the BLAS specialization, showing that major performance gains are possible by using highly optimized third party libraries. The general algorithm starts exhibiting super-linear speed-up for 32 processors, when data fits in cache. The plots include the execution of PBLAS whenever the data layout allows us to use it, using the same baseline for the speed-up. It can be seen that the performance of the BLAS specialized algorithm is comparable to the PBLAS specialized algorithm, which is an interesting result from a productivity point of view.

Finally, Figure 6(b) shows the speed-up achieved by the BLAS specialized workfunction with respect to the execution of PBLAS on one processor. The plots report three experiments varying the number C of components per location (for $C = 1$ we forced the BLAS specialization to be invoked instead of PBLAS). We show results for perfect square processor grids, since this allows us to make fair comparisons. As can be seen, the speed-up decreases as the number of components increases. This is due to the increased number of memory copies and communications executed by the algorithm.

7 Conclusion

In this paper, we addressed the problem of interoperability in STAPL. We showed how STAPL can take advantage of third party parallel and sequential libraries by combining compile time and runtime checks. We illustrated the methodology by implementing a matrix-matrix multiplication algorithm that can exploit the availability of PBLAS and BLAS when the proper conditions are met. Our results show that the overhead of specialization is negligible, and, when proper specialization can be utilized, that STAPL performance is comparable to that of PBLAS. We also showed how STAPL can be used by other languages by providing the proper constructors for `pContainers` to embed foreign data structures within STAPL with negligible overhead.

References

1. An, P., Jula, A., Rus, S., Saunders, S., Smith, T., Tanase, G., Thomas, N., Amato, N., Rauchwerger, L.: STAPL: A standard template adaptive parallel C++ library. In: Proc. of the International Workshop on Advanced Compiler Technology for High Performance and Embedded Processors (IWACT), Bucharest, Romania (2001)
2. Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.: LAPACK Users' Guide. Society for Industrial and Applied Mathematics, 3rd edn. Philadelphia, PA (1999)

3. Breuer, A., Gottschling, P., Gregor, D., Lumsdaine, A.: Effecting parallel graph eigensolvers through library composition. In: 20th International Parallel and Distributed Processing Symposium, 2006. IPDPS 2006, p. 8 (March 2006)
4. Buss, A.A., Smith, T.G., Tanase, G., Thomas, N.L., Olson, L., Fidel, A., Bianco, M., Amato, N.M., Rauchwerger, L.: Design for interoperability in STAPL: pMatrices and linear algebra algorithms. Technical Report TR08-003, Dept. of Computer Science, Texas A&M University (August. 2008)
5. Choi, J., Dongarra, J.J., Ostrouchov, L.S., Petitet, A.P., Walker, D.W., Whaley, R.C.: Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. Scientific Programming 5(3), 173–184 (Fall, 1996)
6. Demmel, J., Dongarra, J., Parlett, B.N., Kahan, W., Gu, M., Bindel, D., Hida, Y., Li, X.S., Marques, O., Riedy, E.J., Vömel, C., Langou, J., Luszczek, P., Kurzak, J., Buttari, A., Langou, J., Tomov, S.: Prospectus for the next lapack and scalapack libraries. In: PARA, pp. 11–23 (2006)
7. Edjlali, G., Sussman, A., Saltz, J.: Interoperability of data parallel runtime libraries with meta-chaos. Technical Report CS-TR-3633, University of Maryland (1996)
8. Gottschling, P., Wise, D.S., Adams, M.D.: Representation-transparent matrix algorithms with scalable performance. In: ICS 2007: Proceedings of the 21st annual international conference on Supercomputing, pp. 116–125. ACM, New York (2007)
9. Gregor, D., Stroustrup, B., Widman, J., Siek, J.: Proposed wording for concepts. technical report n2617=08-0127. ISO/IEC JTC 1, Information Technology Subcommittee SC 22, Programming Language C++ (2008)
10. Järvi, J., Marcus, M., Smith, J.: Library composition and adaptation using c++ concepts. In: GPCE 2007: Proceedings of the 6th international conference on Generative programming and component engineering (October 2007)
11. Järvi, J., Willcock, J., Hinnant, J., Lumsdaine, A.: Function overloading based on arbitrary properties of types. C/C++ Users Journal 21, 25–32 (2003)
12. Karypis, G., Schloegel, K., Kumar, V.: ParMeTis: Parallel Graph Partitioning and Sparse Matrix Ordering Library, Version 2.0. University of Minnesota, Dept. of Computer Science (September 1999)
13. Lawson, C.L., Hanson, R.J., Kincaid, D.R., Krogh, F.T.: Basic linear algebra subprograms for fortran usage. ACM Trans. Math. Softw. 5(3), 308–323 (1979)
14. Musser, D., Derge, G., Saini, A.: STL Tutorial and Reference Guide, 2nd edn. Addison-Wesley, Reading (2001)
15. Tanase, G., Bianco, M., Amato, N.M., Rauchwerger, L.: The STAPL pArray. In: Proceedings of the 2007 Workshop on Memory Performance (MEDEA), Brasov, Romania, pp. 73–80 (2007)
16. Tanase, G., Raman, C., Bianco, M., Amato, N.M., Rauchwerger, L.: Associative parallel containers in STAPL. In: Adve, V., Garzarán, M.J., Petersen, P. (eds.) LCPC 2007. LNCS, vol. 5234, pp. 156–171. Springer, Heidelberg (2008)
17. Thomas, N., Saunders, S., Smith, T., Tanase, G., Rauchwerger, L.: ARMI: A high level communication library for STAPL. Parallel Processing Letters 16(2), 261–280 (2006)
18. Thomas, N., Tanase, G., Tkachyshyn, O., Perdue, J., Amato, N.M., Rauchwerger, L.: A framework for adaptive algorithm selection in STAPL. In: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), Chicago, IL, USA, pp. 277–288. ACM, New York (2005)
19. Veldhuizen, T.L.: Arrays in blitz++. In: Caromel, D., Oldehoeft, R.R., Tholburn, M. (eds.) ISCOPE 1998. LNCS, vol. 1505, pp. 223–230. Springer, Heidelberg (1998)

Implementation of Sensitivity Analysis for Automatic Parallelization*

Silvius Rus, Maikel Pennings, and Lawrence Rauchwerger

Parasol Lab, Department of Computer Science, Texas A&M University
rus@google.com, {pennings,rwerger}@cs.tamu.edu

Abstract. Sensitivity Analysis (SA) is a novel compiler technique that complements, and integrates with, static automatic parallelization analysis for the cases when program behavior is input sensitive. SA can extract all the input dependent, statically unavailable, conditions for which loops can be dynamically parallelized. SA generates a sequence of sufficient conditions which, when evaluated dynamically in order of their complexity, can each validate the dynamic parallel execution of the corresponding loop. While SA's principles are fairly simple, implementing it in a real compiler and obtaining good experimental results on benchmark codes is a difficult task. In this paper we present some of the most important implementation issues that we had to overcome in order to achieve a fairly successful automatic parallelizer. We present techniques related to validating dependence removing transformations, e.g., privatization or pushback parallelization, and static and dynamic evaluation of complex conditions for loop parallelization. We concern ourselves with multi-version and parallel code generation as well as the use of speculative parallelization when other, less costly options fail. We present a summary table of the contributions of our techniques to the successful parallelization of 22 industry benchmark codes. We also report speedups and parallel coverage of these codes on two multicore based systems and compare them to results obtained by the Ifort compiler.

1 Introduction

1.1 Automatic Parallelization - Current State of the Art

The recent introduction of multi-core based architectures to the mass market has brought program parallelization of the existing code base to the forefront. In fact, there seems to be a degree of urgency from the part of the major vendors to enable their users to exploit the coarser level parallelism offered by these new micros with their existing software base. Parallelizing compilers are a key enabling

* This research supported in part by NSF Grants EIA-0103742, ACR-0081510, ACR-0113971, CCR-0113974, ACI-0326350, CRI-0551685, CCF-0702765, CNS-0615267, by the DOE, IBM, and Intel. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DOE-AC02-05CH11231.

J.N. Amaral (Ed.): LCPC 2008, LNCS 5335, pp. 316–330, 2008.

technology in this domain because they offer the advantage of automation and thus high productivity.

Parallelizing compilers must focus, at least as a necessary first step, on discovering which loops can be executed in parallel (ideally as a `doall`). Data dependence analysis techniques as simple as the GCD test [16] and as sophisticated as the Omega test [8] have been employed to statically prove the independence of memory references within a loop. After some limited success it had become clear that sparse, dynamic programs could not be automatically parallelized using these static techniques alone because their memory reference pattern is input dependent. The proposed solution was dynamic (run-time) analysis with the advantage of high accuracy (most symbolic data is instantiated) but with the drawback of run-time overhead. The dynamic approach has taken two directions: (a) a continuation of the static compilation analysis at run-time, and (b) a memory reference trace based analysis approach. In the first approach, symbolic expressions that could not be evaluated statically are postponed for run-time evaluation which then decides the (in)dependence of a loop. For example, if the static analysis cannot conclusively perform a standard data dependence test, e.g., a GCD test, because some of its parameters can be evaluated, we can always perform it at run-time when all information becomes available. In the second approach, more general and better suited for codes using indirection, the memory references are recorded and analyzed at run-time either before a loop is executed (inspector–executor mode [15]) or after an optimistic (speculative) parallel execution [11]. The complexity of this method is proportional to the number of dynamic references and thus is potentially expensive.

Overall, the static and run-time approaches to automatic parallelization have progressed independently without significant integration. Partial, but insufficient, static analysis was not used effectively to simplify run-time analysis. An improvement over this state of the technology was presented in [12]. Instead of performing a reference-based test, the technique, named Hybrid Analysis, uses an aggregated reference representation and performs dynamic analysis using set and interval operations very similar to those performed statically by a compiler. This often results in a significant reduction of run-time overhead.

A step further in automatic parallelization has been the re-formulation of the loop independence analysis into sufficient conditions (predicates) for which a loop can be parallelized. These conditions represent the *sensitivity* of parallelization to some input (dynamic) conditions. For example, in [9] the authors showed some limited examples of how sufficient predicates could be extracted by simplifying Presburger formulas with uninterpreted function symbols. These predicates are returned to the programmer for evaluation (for interactive compilation). Further research [2,5,6,12] showed how to extract simple scalar conditions from relatively simple array data dependence predicates for a limited number of cases.

We have used a similar approach and recently presented Sensitivity Analysis (SA) [13] as a general framework to analyze memory references and used it to extract parallel loops from sequential programs. SA seamlessly bridges static and dynamic analysis of memory references. When the compiler cannot draw

definitive conclusions about interesting properties of a memory reference pattern, SA can generate a set of sufficient conditions which, when evaluated, can (in)validate these interesting properties. Examples of such interesting properties are (in)dependent memory references, privatizable references, reductions, etc.

1.2 Automatic Parallelization with Sensitivity Analysis

In [12,13] we have shown how our compiler using SA is able to extract most available loop level parallelism from various benchmark codes using a mix of advanced static analysis and aggressive optimizations that are validated dynamically with minimal overhead. This has resulted in fairly good speedups. In [12,13] we have explained with some detail how the overall SA framework functions. However, obtaining good results requires us to apply and refine many general techniques that together contribute to good speedups.

For example, we mentioned that SA generates a set of sufficient conditions that can be evaluated dynamically and validate parallelization. However, the work (run-time overhead) involved in the dynamic evaluation of these predicates can vary greatly. Thus an ordering of their evaluations from simple to complex is crucial (somewhat similar to evaluating complex predicates) for obtaining good performance. In fact, based on performance models we can stop evaluating predicates if the effort outweighs the benefit of parallelization.

Further examples are simple algorithm substitution transformations. Exchanging a serial reduction with a parallel one can enable the parallelization of large loops. These transformations have to be proven correct though, and, in the case of complex or input sensitive memory reference patterns, this may not be possible statically. We use the same SA approach to generate dynamic conditions to validate parallelizing code transformations.

Contribution. In this paper we present some important aspects describing how the general framework of Hybrid Analysis (presented elsewhere [12,13]) has been used and implemented in our parallelizing compiler (which is a derivative of the UIUC-Polaris compiler).

2 A Brief Introduction to Sensitivity Analysis

The Memory Reference Representation
There are three main concepts in our analysis. First, we introduce a powerful memory reference representation, the USR (uniform set representation). It was described in detail in [12] under the name RT_LMAD. In essence it can represent memory references of a program as an expression whose leafs are sets of LMADs (linear memory access descriptors) or enumerated sets of references which are composed (internal nodes of the expressions) through program operations (conditionals, loops, subroutine calls, etc.) A crucial advantage of this representation is that it is closed under composition - it can represent any memory reference pattern symbolically, at program level. When USRs cannot be evaluated to the

exact sets of addresses they represent at compile time, they can be embedded in the generated code and computed at run time, in the presence of actual input values. However, in most cases we do not need to compute the actual memory reference pattern, but rather prove a relation, which is generally easier.

Memory Reference Aggregation and Classification
The second concept in SA is memory reference aggregation, which ensures scalability of interprocedural analysis at the cost of losing dependence direction information. Memory references are aggregated bottom up on the Control Dependence Graph (CDG) within a subroutine, and on the call graph inter-procedurally.

The process starts at leaf CDG nodes, which are simple statements. The set of memory locations *read* and *written* by the statement is computed from the statement type and symbolic expressions. This set is parameterized by symbolic variables referenced by the statement.

The sets corresponding to successive statements are then computed using set union, intersection and difference. All these operations are performed on USRs [12]. Special nodes in the CDG require more elaborate set operations, all of which are well defined and closed on USRs: predication, union across iteration space and symbolic translation.

These simple, node-local transformations on the CDG are applied repeatedly until the memory reference pattern has been completely summarized across the whole program.

Dependence Relations Based on Reference Summary Sets
While summarizing references, we also classify them into three disjoint sets [2]: Read Only (RO), Write First (WF) and Read Write (RW). They represent the specific data flow information needed for dependence analysis. The RO summary set records all memory locations only read (not written) within a section of code, the WF summary set records all memory locations that are written first and then possibly read and written, and the RW summary set records all other memory locations referenced from within a context. Computing the RO, WF and RW sets requires only the USR operations discussed in the previous section. An example is given in Fig. 1.

Every time we reach a loop header in the aggregation process, we compute the cross iteration data dependence relations. If there are no dependences, then all the loop iterations can be executed in parallel. This is the most effective automatic parallelization method, as it scales with the number of iterations, thus it is likely to remain efficient as the underlying hardware evolves towards a larger number of processing units.

```
1   ... = A(6:15)        RO = [6 : 15], WF = ∅, RW = ∅
2   A(1:10) = ...        RO = ∅, WF = [1 : 10], RW = ∅
3   ...                  RO = [11 : 15], WF = [1 : 5], RW = [6 : 10]
```

Fig. 1. Memory reference classification example

To express cross iteration dependence relations, we compute the set of memory locations that are referenced in two different iterations, and are written in at least one. At this point in the analysis, we have already computed RO_i, WF_i and RW_i, the per iteration reference sets.

One such dependence set is

$$DS = \cup_{i=1}^{n} RO_i \bigcap \cup_{i=1}^{n} WF_i$$

Similar dependence sets are expressed for combinations of RO, RW and WF sets [12]. If we prove $DS = \emptyset$, then no cross-iteration dependences may exist.

Sensitivity of Dependence Relations to Parameters
Finally, the third concept used in SA is the transformation of the USRs representing the aggregated memory references into a **Sensitivity Graph (SG)**, i.e., a boolean expression representing the parallelization conditions.

In many cases, proving the dependence set empty is trivial. It often results from a set intersection such as $[1:10] \cap [11:20]$, which evaluates to $\emptyset$ through symbolic calculus, at compile time. In other cases, proving the dependence set empty is not possible at compile time either because it depends on input data, e.g., $DS = [1:n] \cap [m:100]$ or because the relation is just too complicated for the compiler to evaluate.

We build SGs from dependence equations based on USRs by using a divide and conquer approach, which, at each step, breaks the dependence equation $DS = \emptyset$ into several simpler equations based on set identities [13]. For instance, equation $A \cup B = \emptyset$ is broken into $A = \emptyset$ and $B = \emptyset$. This algorithm is applied recursively until we reach equations involving only intervals, such as $[m:n] \cap [p:q]$. Such equations are translated into simple predicates based on bound comparison, e.g., $n < p$ *or* $q < m$.

We then extract a minimal (modulo the symbolic calculus capabilities of the compiler) run time check that guarantees that the loop is parallel. We then generate parallel code predicated by this condition. We use the SG [13] representation for these conditions. When they cannot be evaluated at compile time to a boolean value, they are embedded in the generated code and evaluated at run time, in the presence of actual values.

The aggregation and equation solving processes can deal with multidimensional strided reference patterns. In some cases, the divide and conquer process cannot extract a precise predicate from a dependence equation. In such cases, we approximate sets with predicated multidimensional strided intervals, and continue the analysis with affine sets, which are easier to compare. The predicates are added to the dependence condition. We can afford to make up optimistic, speculative predicates, since they are verified at run time through SG evaluation.

3 Engineering an Automatic Parallelizer

In the previous section we have provided an overview of the general approach to parallelization: We aggregate and, at the same time classify memory references

(WF, RO, RW) at the program level into a set representation (USR) and then formulate the independence condition $DS = \emptyset$ (empty dependence set). Then, the compiler verifies the conditions for which this equation holds true by recursively descending on the equation $DS = \emptyset$ and, using boolean logic, generating a conjunction (OR) of simpler equations. Some of these equations can be proven true for all inputs, i.e., statically true, and others result in some constraints for the equation to be true. From these constraints (conditions) the compiler generates predicates (code) that are evaluated at run-time and can validate the parallelization of a loop. The constraints are expressed as sets of expressions which can be represented as a graph, the sensitivity graph (SG). This method was presented as SA (sensitivity analysis) [13].

Parallelism enhancing transformations. Our overall goal is to uncover as much parallelism (`doall` type only) as possible and exploit it when beneficial. To this we apply our SG based technique not only to prove that the original loops in a program are independent but also to validate code transformations that increase the intrinsic amount of parallelism. We will show how we can use our SA to perform powerful **dependence removing transformations**, e.g., reduction parallelization, pushback parallelization and array privatization. These are not new techniques, but the use of SA in their implementation makes them more powerful, i.e., more often successful.

Efficient Run-time Evaluation of Parallelization Conditions. After applying the dependence removing transformations the compiler needs to generate efficient parallel code. The outcome of the static Sensitivity Analysis may be the SG (sensitivity graph) which may be varying degree of complexity and which needs to be efficiently evaluated dynamically. It is important to perform the dynamic evaluation efficiently because this evaluation represents pure overhead. The novelty of our implementation lies in the way we generate efficient code for this dynamic validation.

We will present some of the more important aspects of this process, e.g., the generation of predicates that pre-validate parallel loop execution and the use of speculation and post execution validation. Sometimes we cannot extract a condition that can be evaluated before a loop is executed because it depends on the computed data. (There may be a cycle between address and data computation). In this case, we have to resort to speculative execution [11]. This invokes other efficiency issues such as checkpointing (if used). Here too we use our program representation and SA to improve performance.

It is worth mentioning that our entire analysis framework is interprocedural. For the evaluation of USRs at run-time we have developed a library to which we generate calls. Similarly, when we employ LRPD we use a specialized library.

Let us now take a closer look at some powerful techniques.

3.1 Transformations to Remove Dependences

Conditional and Selective Array Privatization. Array privatization can be complex and expensive. In general, it means allocating a private array in

each thread of execution. This replication can become quite costly if the array is big. In the most general case it is required to first copy-in from the shared array and then, after processing, copy-out the last value written. These two operations (copy-in and copy-out last value) can be very expensive because they do not scale. Thus we optimize them by performing selective copy-in and last value copy-out. In the case of relatively sparsely referenced arrays this can save significant time.

We can use USRs to express these in/out sets precisely in a general way and thus improve performance. Briefly, here is our approach:

By the time we reach a loop header, we have already classified all memory locations referenced within each iteration i into disjoint sets (USRs) RO_i, WF_i and RW_i. Using only set operations, we put together the following descriptors as per-iteration USRs.

$$USR\ to\ privatize = WF_i \cap (\cup_{k \neq i} WF_k) \tag{1}$$

$$USR\ to\ copy\ in = RO_i \cup RW_i \tag{2}$$

$$USR\ to\ copy\ out = WF_i - (\cup_{k=i+1}^{n} WF_i) \tag{3}$$

In practice, we compute a single USR for the iteration space of each thread, or a single USR for the whole iteration space of the loop. The formulas assume that we have already proved that $\cup_{i=1}^{n} WF_i \bigcap \cup_{i=1}^{n} (RO_i \cup RW_i) = \emptyset$, which we do as part of solving the dependence equation. This essentially means that the only dependences left can be eliminate by privatizing overlaps of WF_i across iterations on different threads.

The first descriptor contains the set of memory references that must be privatized because they are written to in at least two iterations. We chose to generate an OpenMP PRIVATE directive whenever this USR is not provably empty at compile time. This means we are possibly allocating too much private storage, since sometimes not all the elements in the array must be privatized. However, the alternative is to use an indirection table for just those locations that must be privatized, which introduces both complexity and overhead.

Although we privatize entire arrays, we perform selective and conditional copy in. Only those locations that are read before being written inside the loop are used in a memory copy operation from the shared object to the private copies. They are only copied if it turns out, at run time, that the values are needed inside the loop, based on actual control flow predicates. We wrote a simple *memcpy like* routine that uses a USR to control which locations get copied.

Conditional and Selective Array Reduction. Although implementations vary greatly, array reduction conceptually starts with an initialization of all the elements participating in the reduction with the null element of the reduction operator. The loop is then executed in parallel. Upon exit from the parallel section, elements updated by more than one thread are merged using the reduction operation. We use USRs to describe the extent of the initialization and merge

phases, and wrote simple library routines that use USRs to control the exact locations that are initialized and merged respectively.

$$USR\ to\ initialize = USR\ to\ reduce = \cup_{i=1}^{n} \left[RW_i \cap \left(\cup_{k=1}^{i-1} RW_k \right) \right] \quad (4)$$

Conditional Parallelization of Pushback Sequences. We have shown [14] how to recognize sequences of pushback operations that can be parallelized by using private storage, which simply need to be copied at the end of the loop to a specific location of the shared array. We use USRs WF_i to describe the extent of the writes to private storage, and a library function to perform the actual copies (the same used for copy in and copy out).

Not only do they get relocated efficiently, but this makes the transformation more general, since USRs can describe arbitrarily complex patterns. Previously, only pushback sequences made of contiguous locations could be parallelized.

3.2 Sensitivity Graph (SG) Evaluation

The outcome of the static Sensitivity Analysis may be either a definitive answer at compile time or the Sensitivity Graph (SG), a boolean expression that needs to be efficiently evaluated at run-time. It represents a conjunction (logic OR) of sufficient conditions which all can validate a loop to be parallel (including the associated dependence removing transformations). The SG can be of various complexities. They can be a:

a Boolean expression that can be evaluated in constant time.
b Boolean expression that can be evaluated in time proportional to some fraction of the size of the program data. For example, a triply nested loop with iteration spaces N,M,K can be parallelized by performing N (or N*M or K*N) work. This situation arises many times when aggregation works well in only some of the dimensions of the analyzed data structures.
c Boolean expression that can be evaluated in time proportional to data size.

In this latter case (c), some of the transformations of the equations ($DS = \emptyset$) involving the globally aggregated USRs into simpler ones has failed. This can happen when the recursive simplification of the $DS = \emptyset$ equation is not very successful or, in an extreme example, when the code uses indirection arrays. In effect, we need to generate code to dynamically evaluate the USR's (which can be seen as a program slice). The compiler will generate code for the evaluations

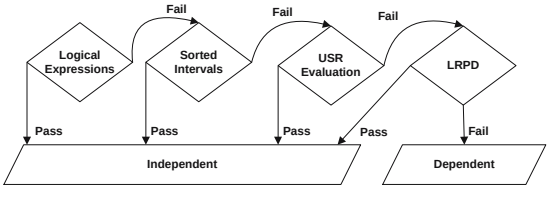


Fig. 2. Cascade of sufficient run time tests in increasing order of complexity

of these conditions and sort them in order of their estimated complexity (similar to the predicate of a branch condition). For illustration purposes, we have named the resulting code a *cascade of sufficient conditions* (Fig. 2).

There are four types of run time operations involved in the evaluation of the SG: (1) evaluation of elementary conditional expressions (constant time), (2) interval trees (some fraction of data size, simple operations), (3) actual evaluation of USRs (fraction of data size, complex operations) and comparison to the empty set and (4) reference-by-reference LRPD [11]. The estimated complexity of these tests ranges from $O(1)$ tests as the one in Fig. 3 to $O(n)$ dynamic reference instrumentation as is the case in Fig. 4. The evaluation of USRs at run time generally consists of fewer, but more complex operations than the reference-by-reference LRPD. In some cases they may either degenerate into inefficient enumerations or take conservative decisions that can lead to false negatives. The LRPD test has overhead proportional to the dynamic reference count, but is optimal for cases where aggregation and equation inversion are not possible (Fig. 4). It is always applicable, precise, and has a more predictable complexity. Perhaps the most important aspect of the "heavy" methods (USR evaluation or LRPD test) is that they have to be performed in parallel so that the overall obtained speedup scales with the number of processors.

There are two ways to validate parallel execution: Before the loop execution (similar to an inspector) or after its execution. In the latter case we have to use speculative execution [11].

In most cases, we can adopt either method and (hopefully) select the most efficient one. The correct choice involves a more complex cost model which is beyond the scope of this discussion. Presently, we choose speculation over pre-verification only if (1) a parallel inspector cannot be extracted (see next section) or (2) if we cannot extract a light inspector (a slice made of only scalar definitions). The actual test code generation consists of a syntax-based translation from the SG grammar to Fortran.

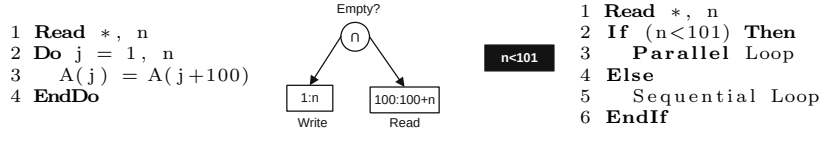


Fig. 3. Example of an input-sensitive memory reference pattern and corresponding code after parallelization

Fig. 4. A Hybrid Analysis extreme: in general, no test can solve this problem faster than the reference-by-reference LRPD test

In both cases, we reuse the test results by means of inspector hoisting, SG and USR common subexpression recognition, and run time test result memoization. We apply loop invariant hoisting to USRs and SGs by performing aggressive invariance analysis on their sets of input variables. Invariance problems on USRs resulting from subscripted subscripts are formulated as dependence problems on the subscript arrays, which *are solved by the same SA algorithm applied to the subscript array.* This is achieved by representing the exact referenced memory regions of the subscript array as USRs themselves, and thus identifying the exact subregion of the subscript array that affects the shape or size of the memory pattern on the host array. An interesting problem arises when a more expensive test such as LRPD can be hoisted out of a loop, but a simpler O(1) version is loop variant. At this time we (simplistically) hoist tests as far away as possible and build cascades from tests at the same loop nesting level.

Even when we cannot hoist tests out of their loops, by transforming reference based tests into simple boolean operations, we reduce the run time overhead by a constant but significant factor. For instance, in MDG/INTERF_do1000, switching from reference based test to SGs improved the speedup on 4 threads from 1.5 to 3.3. (Scalability did not change). There are three main reasons for this improvement. First, simple SGs evaluation require very little extra memory, in most cases a few scalar variables. The code we insert consists of accumulation of conditions such as $indep = indep.AND.x > 0$. Second, we insert the code at the earliest common postdominator of the SSA definition sites of the operands. Since in most cases there is just one operand other than the dependence decision accumulator, we insert code right after that operand is written to, which gives excellent temporal locality. If the definition is an unconditional scalar assignment, the operand is likely to be in a register. Also, simple SGs only perform logic operations. In contrast, the LRPD and USR run time libraries, even highly optimized, may use large amounts of extra memory and execute bookkeeping operations including loads/stores, branches and sub-word manipulation.

3.3 Speculative Execution

Sometimes we cannot extract a condition that can be evaluated before a loop is executed because it depends on the computed data. (There is a cycle between address and data computation). In this case, we have to resort to speculative execution [11]. This invokes other efficiency issues such as checkpointing (if used). We identified previously the conditional pushback sequence pattern, which is perhaps the simplest such example. Other cases are more complex and do not follow a preset pattern. It should be noted that even when the dependence relation can be precomputed before the loop it may be worth executing the loop in parallel speculatively in order to reduce the overhead. A more detailed discussion about these choices can be found in [7].

If speculative parallelization is necessary, we take advantage of our novel representation and SA techniques to reduce overheads. We can compute the exact extent (as a USR) of memory that must be either saved at a checkpoint before the speculative loop, or committed from private speculative storage after the

loop. The actual memory operations are implemented as calls to our selective memory copy routine used for copy in, copy out and pushback parallelization.

3.4 The Value Evolution Graph and Pushback Sequences

The *Value Evolution Graph (VEG)* [14] can represent the data flow in recurrences used as array indices which have no closed form solutions. The graphs are pruned based on control dependence predicates and produce tighter value ranges than abstract interpretation methods. These value ranges and their relations (overlapping, mutual exclusive) are used throughout our analysis, when building USRs and when extracting SGs.

Additionally, VEGs can be used to detect monotonic reference patterns in the code text. Unlike previous pattern recognition methods, we can analyze partially aggregated and classified memory descriptors (USRs). This single generic approach both extends and unifies in a single framework most cases which were previously solved using various, different, pattern matching techniques. It allows for the parallelization of important classes of memory reference patterns, e.g., sequences of pushback operations with complex footprints.

4 Experimental Results

Our experiments show that our techniques extract almost all the available parallelism at the highest granularity possible, which results in significant speedups on 22 codes from the PERFECT and various SPEC benchmark suites.

Table 1. Automatic parallelization coverage and speedups. PERFECT and SPEC92/95 speedups were measured on a 2-way Intel Core Duo. SPEC2000/2006 speedups were measured on an 8-way Sun server with 4 AMD dual core processors.

	Coverage		**Speedup**	
Suite	Polaris/SA	Intel	Polaris/SA	Intel
PERFECT	95%	14%	1.51	1.02
SPEC92/95	98%	29%	1.44	1.10
SPEC2000/2006	88%	62%	2.87	1.66

Table 2. Parallelization coverage breakdown (a) between compile time and run time (b) as contribution of each compiler technique. The coverages in (b) overlap because parallelizing some loops required several techniques. Coverage is measured as the percentage of the original sequential execution time that was parallelized.

Technique	**PERFECT**	**SPEC**
CT	58.00	87.25
RT: Speculative	11.90	1.42
RT: Non-Speculative	26.60	4.08
Total	95.50	92.75

(a)

Technique	**PERFECT**	**SPEC**
SG: Simple Expressions	20.10	2.08
SG: Interval Trees	9.20	0.00
SG: LRPD	3.00	1.42
SG: USR Evaluation	13.80	4.08
Hybrid Priv, Red	12.60	0.50
VEG	12.80	0.91
Pushback Recognition	9.60	0.92

(b)

Table 3. Run time tests actually executed to decide whether the dependence structure on array *MX* prohibits or allows parallelization of loop *DYFESM/MXMULT_do10*. %S represents the time spent in the test as a percentage of the execution time of the loop.

Test	Type	Accuracy	Success	% S
Parallel/Sequential	Simple Expression	Sufficient	Fail	0.005
	USR Evaluation	Necessary&Sufficient	Pass	0.025
Indep. Update/Reduct.	Simple Expression	Sufficient	Fail	0.005
	USR Evaluation	Necessary&Sufficient	Fail	0.030
Indep. Write/Priv.	Interval Trees	Necessary&Sufficient	Pass	0.005

Table 4. Loop parallelization in PERFECT codes. % = percentage of total application execution time. DD Test = type of data dependence test required (CT = compile time, RT = run time, SE = simple logical expressions, IT = interval trees, UE = USR evaluation, LRPD = LRPD run time test) Priv = type of privatization required (A = array privatization). Red = type of reduction required. PB = pushback required. IP = loop contains subprogram calls. EX = execution type (IE = nonspeculative, SP = speculative execution). Intel = parallelized automatically by the Intel Compiler (version 9.1, *-parallel -par_threshold100*).

Code	Loop	%	DD Test	Priv	Red	PB	IP	EX	Intel
ADM	RUN_do20,...,100	44	RT:SE,UE	CT,A	-	-	√	IE	-
	D*DTZ_do30	31	CT	CT,A	CT	-	√	-	-
	DKZMH_do20,50	11	CT	CT,A	-	-	√	-	-
	WCONT_do40	5	CT	CT,A	CT	-	√	-	-
ARC2D	STEPF*_do*	29	CT	CT	-	-	-	-	√
	*PENT*_do*	14	CT	CT	-	-	-	-	√
	FILERX_do15	14	RT:SE,UE	CT,A	-	-	-	IE	-
	RHS*_do*	10	CT	CT	-	-	-	-	√
	TK*_do1	8	CT	CT	-	-	-	-	-
BDNA	ACTFOR_do240,500	89	CT	CT,A	CT	-	-	-	-
DYFESM	MXMULT_do10	73	RT:IT,UE	RT:IT,A	RT:IT,UE	-	√	IE	-
	SOLVH_do20	9	RT:SE	RT:IT,A	-	-	√	IE	-
	FORMR0_do20	7	RT:IT,UE	RT:IT,A	RT:IT,UE	-	√	IE	-
	SOLXDD_do4,10,30,50	9	RT:IT	RT:IT,A	RT:IT	-	√	IE	-
FLO52	*FLUX*_do*	55	CT	CT	-	-	-	-	√
	PSMOO_do40,80	21	CT	CT	-	-	-	-	-
	EULER_do*	15	CT	CT	CT	-	-	-	√
MDG	INTERF_do1000	93	RT:SE	CT,A	CT	-	√	SP	-
	POTENG_do2000	6	CT	CT,A	CT	-	√	-	-
OCEAN	FTRVMT_do109	41	RT:SE	CT	-	-	-	IE	-
	IN_do10	15	CT	-	-	-	-	-	-
	OUT_do10	15	CT	-	-	-	-	-	-
	CSR,RCS_do20	7	CT	CT	-	-	-	-	-
	ACAC,SCSC_do30,40	6	CT	CT,A	-	-	-	-	-
SPEC77	GLOOP_do1000	48	CT	CT,A	CT	-	√	-	-
	GWATER_do1000	24	RT:LRPD	CT,A	CT	-	√	SP	-
	SICDKD_do1000	4	CT	CT,A	-	-	√	-	-
TRACK	EXTEND_do400	50	CT	CT,A	-	√	√	-	-
	FPTRAK_do300	46	CT	CT,A	-	√	√	-	-
	NLFILT_do300	2	RT:LRPD	CT,A	-	-	√	SP	-
TRFD	OLDA_do100	67	CT	CT,A	-	-	-	-	-
	OLDA_do300	28	CT	CT,A	-	-	-	-	-
	INTGRL_do140	3	RT:IT	RT:IT,A	-	-	-	IE	-

Table 1 presents full application speedups, measured by dividing the sequential execution time of the whole application by its parallel execution time including the runtime overhead, if any.

Table 5. Loop parallelization in SPEC codes. (Legend in Table 4.).

Code	Loop	%	DD Test	Priv	Red	PB	IP	EX	Intel
APPLU	JACL*_do#1	34	CT	CT	-	-	-	-	-
	RHS_do#1,2,3,4	20	CT	CT	-	-	-	-	√
APSI	RUN_do*	25	RT:SE,UE	CT,A	-	-	√	IE	-
	D*DTZ_do40	40	CT	CT,A	CT	-	√	-	-
	DKZMH_do30,60	12	CT	CT,A	-	-	√	-	-
	WCONT_do40	6	CT	CT,A	CT	-	√	-	-
	HYD_do20	5	CT	CT	CT	-	-	-	-
MGRID	RESID_do600	52	CT	CT	-	-	-	-	√
	PSINV_do600	27	CT	CT	-	-	-	-	√
	RPRJ3_do100	7	CT	CT	-	-	-	-	√
	INTERP_do400,800	8	CT	CT	-	-	-	-	√
	COMM3_do100,200,300	5	CT	CT	-	-	-	-	-
SWIM	SHALLOW_do3500	48	CT	CT	CT	-	-	-	-
	CALC1_do100	14	CT	CT	-	-	-	-	√
	CALC2_do200	17	CT	CT	-	-	-	-	√
	CALC3_do300	19	CT	CT	-	-	-	-	√
WUPWISE	MULDEO_do100,200	47	CT	CT,A	-	-	√	-	-
	MULDOE_do100,200	46	CT	CT,A	-	-	√	-	-
HYDRO2D	FILTER_do*	42	CT	CT	-	-	-	-	√
	FCT_do*	18	CT	CT	-	-	-	-	√
	ARTDIF_do*	14	CT	CT	-	-	-	-	√
	TRANS*_do*	12	CT	CT	-	-	-	-	√
	TISTEP_do*	6	CT	CT	-	-	-	-	√
	S1,S2_do100	4	CT	CT	-	-	-	-	-
MATRIX300	LBMK14_do20	13	CT	CT	-	-	-	-	-
	SGEMM_do*	86	CT	-	-	-	√	-	-
MDLJDP2	FRCUSE_do20	76	CT	CT	CT	-	√	-	-
	FRCBLD_do20	11	CT	CT	CT	√	√	-	-
	POSTFR_do*	8	CT	CT	CT	-	-	-	-
	PREFOR_do*	5	CT	CT	-	-	-	-	-
NASA7	VPETST_do110	26	CT	CT	-	-	√	-	-
	GMTTST_do120	24	RT:UE	CT	-	-	√	IE	-
	CFFT2D*_do130,150	17	RT:LRPD	CT	-	-	-	SP	-
	BTRTST_do120	10	CT	CT	-	-	√	-	-
	CHOTST_do120	9	CT	CT	-	-	√	-	-
	EMIT_do5	6	CT	RT:IT,A	-	-	-	IE	-
ORA	MAIN_do9999	99	CT	CT	CT	-	√	-	-
SWM256	CALC1_do100	31	CT	CT	-	-	-	-	√
	CALC2_do200	38	CT	CT	-	-	-	-	√
	CALC3_do300	30	CT	CT	-	-	-	-	√
TOMCATV	MAIN_do100/2,120/2,60,...	96	CT	CT	CT	-	-	-	√

Two main factors are behind these good speedups: high granularity and high coverage. The VEG, the USR and SG are all interprocedural and flow sensitive (though they use approximations), which makes our analysis apply to large program slices, resulting in higher granularity. Our hybrid approach pushed coverage over 90%. It also increased granularity significantly, since many outer loops could be proved parallel only at run time. A detailed discussion of the speedup numbers can be found in [13].

Table 2 presents the effect of each technique towards our goal of achieving highest parallelization coverage possible. It is important to note that our hybrid framework solves the parallelization problems uniformly at both compile-time and run-time, using SGs. The techniques presented in this paper contribute substantially to the coverage and granularity of parallelization. Comprehensive reports for a large set of loops are available at: `http://parasol.tamu.edu/compilers/ha`. An interesting case is loop *MXMULT_do10*, which accounts for

73% of the sequential execution time on *DYFESM*. This loop contains an array *MX* which shows multiple patterns on different subsections. The first part of the array is only written to, while the last part is a reduction. The write section is fully independent, but this is not known until run time. The reduction section is only proven a proper reduction (not an independent update) at run time. Table 3 presents our run time tests, their dynamic outcomes and their relative overhead for this loop.

Tables 4 and 5 show the occurrence of each static and dynamic dependence test, privatization, reduction, pushback, and speculative parallelization in various benchmark programs.

5 Conclusions

In this paper we have presented some of the more important issues involved in the implementation of the novel Sensitivity Analysis framework in our Polaris derived automatic paralleling compiler. We have shown that our powerful USR representation and our sensitivity analysis technique is useful not only in detecting independent loops but also in applying parallelism enhancing transformations (e.g, reduction and pushback parallelization, privatization). We have further shown that SA generates a flexible cascade of sufficient conditions applied in order of their estimated execution time complexity. Thus we allow a flexible cost-benefit analysis between the benefits of parallelization and the effort to obtain it. We further present the impact of our methods on 22 benchmark codes and report speedups that compare quite well with existing commercial compilers. These good results are due to our ability to uncover and efficiently exploit large granularity parallelism.

References

1. Agrawal, G., Saltz, J.H., Das, R.: Interprocedural partial redundancy elimination and its application to distributed memory compilation. In: SIGPLAN Conf. on Programming Language Design and Implementation (1995)
2. Hoeflinger, J.: Interprocedural Parallelization Using Memory Classification Analysis. Ph.D thesis, University of Illinois, Urbana-Champaign (August 1998)
3. Kai Chen, D., Torrellas, J., Yew, P.-C.: An Efficient Algorithm for the Run-time Parallelization of DOACROSS Loops. In: Proc. for Supercomputing 1994, Washington D.C (November 1994)
4. Leung, S., Zahorjan, J.: Improving the performance of runtime parallelization. In: Proc. of the 4-thACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (May 1993)
5. Moon, S., Hall, M.W.: Evaluation of predicated array data-flow analysis for automatic parallelization. In: Proc. of the 7-th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, New York, NY, USA (1999)
6. Moon, S., Hall, M.W., Murphy, B.R.: Predicated array data-flow analysis for runtime parallelization. In: Proc. of the 12th Int. Conf. on Supercomputing, Melbourne, Australia (July 1998)

7. Patel, D., Rauchwerger, L.: Principles of speculative run–time parallelization. In: Proc. 11th Annual Workshop on Programming Languages and Compilers for Parallel Computing (August 1998)
8. Pugh, W.: The Omega test: A fast and practical integer programming algorithm for dependence analysis. In: Supercomputing 1991, Albuquerque, N.M (November 1991)
9. Pugh, W., Wonnacott, D.: Nonlinear array dependence analysis. In: Proc. of the 3-rd Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers. Kluwer, Dordrecht (1995)
10. Quiñones, C.G., Madriles, C., Sánchez, J., Marcuello, P., González, A., Tullsen, D.M.: Mitosis compiler: an infrastructure for speculative threading based on precomputation slices. In: Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation, New York, NY, USA (2005)
11. Rauchwerger, L., Padua, D.A.: The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. In: Proc. of the SIGPLAN 1995 Conf. on Programming Language Design and Implementation, La Jolla, CA (June 1995)
12. Rus, S., Hoeflinger, J., Rauchwerger, L.: Hybrid analysis: static & dynamic memory reference analysis. Int. J. of Parallel Programming 31(3), 251–283 (2003)
13. Rus, S., Pennings, M., Rauchwerger, L.: Sensitivity analysis for automatic parallelization on multi-cores. In: Proc. of the ACM Int. Conf. on Supercomputing, Seattle, WA (2007)
14. Rus, S., Zhang, D., Rauchwerger, L.: The value evolution graph and its use in memory reference analysis. In: Proc. of the 13th Int. Conf. on Parallel Architectures and Compilation Techniques, Antibes Juan-les-Pins, France (October 2004)
15. Saltz, J.H., Mirchandaney, R., Crowley, K.: Run-time parallelization and scheduling of loops. IEEE Transactions on Computers 40(5), 603–612 (1991)
16. Wolfe, M.J.: High Performance Compilers for Parallel Computing. Addison-Wesley Longman Publishing Co., Inc., Boston (1995)
17. Yu, H., Rauchwerger, L.: Run-time parallelization overhead reduction techniques. In: Proc. of the 9th Int. Conf. on Compiler Construction, Berlin, Germany, March 2000. LNCS. Springer, Heidelberg (2000)

Just-In-Time Locality and Percolation for Optimizing Irregular Applications on a Manycore Architecture*

Guangming Tan[1,2], Vugranam C. Sreedhar[3], and Guang R. Gao[2]

[1] Department of Electrical and Computer Engineering, University of Delaware
[2] Key Laboratory of Computer System and Architecture, Institute of Computing Technology, Chinese Academy of Science
[3] IBM T. J. Watson Research Center, USA
{guangmin,gao}@capsl.udel.edu, vugranam@us.ibm.com

Abstract. This paper presents a new technique to optimize locality of irregular programs by leveraging parallelism on a massive many-core architecture – IBM Cyclops64 (C64). The key idea is to achieve *Just-In-Time Locality* which ensures that data are available *locally* for computation to use. The proposed percolation model for Just-In-Time Locality moves data proactively close to the computation and organizes the data layout such that locality is exploited effectively. The percolation model opens a door for exploiting locality through parallelism, which is an advantage of the future many-core architecture. We implemented the percolation strategy in the context of two irregular applications on C64. Our experimental results are very encouraging and we get an order of magnitude improvement in performance of irregular applications. We also drastically improve the scalability of the applications that we studied.

1 Introduction

Emerging future microprocessor chip technology unveils a new generation of many-core chip architecture that may contain 100 to 1,000 processing cores. In order to improve the performance and scalability of large-scale applications computer architects, system software designers and application scientists are realizing that they must work closely together to investigate how to exploit the computational power of such new many-core architecture. At a high level there are two kinds of applications—"regular applications" where data access and control flow follow regular and (statically) predictable patterns, and "irregular applications" where data access and control flow have statically (and often even dynamically) unpredictable patterns. Many irregular applications are often implemented using complex pointer data structures such as graph and queue, and recursive control flow is often used to traverse and manipulate such complex pointer data structures. It is difficult and often impossible to capture the data access patterns at compilation time for such applications. For architectures that support memory

* This work has been performed when the first author is a visiting scholar at Computer Architecture and Parallel System Laboratory (CAPSL) of University of Delaware. He is currently associated with Institute of Computing Technology.

J.N. Amaral (Ed.): LCPC 2008, LNCS 5335, pp. 331–342, 2008.

hierarchy, unpredicatable data access patterns often lead to higher off-chip memory access latency, which in turn can degrade the performance and scalability of such irregular applications.

The current threading library (e.g., Pthreads), a combination of compiler directives and libraries (e.g., OpenMP) and optimistic parallelization [1, 2, 3] were not designed to support programming for tolerating off-chip latency, or to handle efficient allocation and movement of data across hierarchy levels. It is often the case that in the underline thread execution model, a thread is enabled and activated as soon as all data and control dependencies are satisfied. Such thread execution models may do well for regular applications, where there is an inherent memory locality in the application. Unfortunately, irregular applications often do not have inherent memory locality and so the weaker model often performs poorly.

In this paper we exploit several characteristics of IBM Cyclops64 (C64) architecture [4] and its runtime threading model to drastically improve the scalability and performance of irregular applications. Our runtime threading model consists of two phases: (1) memory access phase and (2) computation phase. These two phases are orchestrated using Just-In-Time Locality and percolation model in such a way that we can amortize the latency of accessing non-local data across multiple hardware threads. The main contributions of this paper are as follows:

- We highlight the basic notion of Just-In-Time Locality - that has been studied conceptually in our past work on percolation model under the HTMT project [12, 14] to improve and exploit data locality in irregular applications. We show how to interleave computation and memory access such that a thread is enabled only when all of its data, control, and *locality* constraints are satisfied. To hide the latency of memory access we overlap and pipeline the computation phase and the memory access phase across multiple cores or hardward threads.
- We describe percolation programming technique to optimize two important irregular applications—the betweenness centrality algorithm and the dynamic programming algorithm.
- We have implemented our approach on C64 and using our approach we obtained a performance improvement of 4-50 times for betweenness centrality and of 1-2 times for dynamic programming.

The rest of the paper is organized as follows: In section 2, we propose the percolation programming technique in detail. Section 3 discusses how to program irregular program with percolation. Section 4 evaluates the performance of applying percolation model to two irregular applications – betweenness centrality and dynamic programming on a many-core chip architecture. In section 5, we discuss the existing related techniques. Finally, section 6 concludes this paper.

2 Percolation Model and Just-In-Time Locality

In this section we describe some of the key ideas behind our percolation model by exploiting some of the key characteristics of C64 architecture [4].

2.1 C64 Architecture

C64 is a many-core chip architecture that employs a large number of hardware thread units (processing cores), half as many floating point units, (on-chip) SRAM memory banks, an interface to the off-chip DDR SDRAM memory and bidirectional inter-chip routing ports. The C64 chip has no data cache and features a three-level memory hierarchy (Scratchpad memory, on-chip SRAM, off-chip DRAM). A portion of each thread unit's corresponding on-chip SRAM bank is configured as the scratchpad memory. Therefore, the thread unit can access its own scratchpad memory with very low latency through a "backdoor", which provides a fast temporary storage to exploit locality under software control. The remaining portion of each on-chip SRAM bank, together, forms the on-chip global memory that is uniformly addressable from all thread units. There are 4 off-chip memory controllers connected to 4 off-chip DRAM banks.

C64 incorporates efficient support for thread level execution. For instance, a thread can stop executing instructions for a number of cycles or indefinitely; and when asleep it can be woken up by another thread through the execution of a special instruction (casing a direct hardware interrupt). All the thread units within a chip connect to a 16-bit signal bus, which provides a means to efficiently implement barriers. C64 provides no resource virtualization mechanism: the thread execution is *non-preemptive* and there is no hardware virtual memory manager. The former means the OS will not interrupt the user thread running on a thread unit unless the user explicitly specifies a termination or an exception inside C64 chip that is *visible* to the programmer. From a programming model perspective such non-preemptiveness implies that it is important not to stall an execution of a thread once it is scheduled on to a hardware thread unit. In the rest of this section we will describe our percolation model that avoids such unnecessary stalls of running threads.

2.2 Percolation for Just-In-Time Locality

In our percolation threading model a thread has to satisfy two requirements before it can be enabled and ready to run: (i) data/control dependencies have to be satisfied and (ii) *locality* constraints have to be satisfied. The latter requirement essentially enforces that all data referenced by a thread should be local before a thread can be scheduled to run on a hardware thread unit. We represent a program as a directed acyclic task graph, where each node is a task, and a direct arc between two nodes represents a precedence relation between tasks. For regular programs, a user (or a compiler) can often automatically identify computation and memory access tasks, and schedule them such that the latency of memory accesses incurred by the memory access/movement tasks are tolerated. But for an irregular program like graph traversal, it is often difficult to automatically identify task level parallelism and their data access patterns to perform such latency tolerant scheduling statically. One solution is to let the users explicitly specify the task level parallelism. Recall that a necessary condition for a task to become enabled is that all data required by a (computation) task have been produced, and all control dependences are satisfied. In a task graph, a node s (i.e., a task) is enabled if all its predecessor nodes have completed and the required data and control dependences have be satisfied. We call a task that satisfies data and control dependence requirements

as being *logically enabled*. In our percolation model it is not sufficient for a logically enabled task to run. A logically enabled task often cannot immediately run since the data may still be in off-chip memory hierarchy or in the local memory of other cores. We introduce locality constraints in addition to data and control dependence requirements to overcome the latency gap through memory hierarchy. We call a logically enabled task as *locality enabled* if it also satisfies locality constraints. For a task to be locality enabled all data referenced by the task should become local before a task can begin execution. The locality requirement ensures that the corresponding code and data of the candidate task are resident in the same level of memory hierarchy where it is to be enabled.

C64 also supports explicit memory hierarchy and so we use multigrain parallelism to improve performance and scalability of applications. Coarse-grained parallelism is used to enable a thread at coarse-grain level where data and control dependencies are satisfied, and fine-grain parallelism is used to enable a thread at fine-grain level when locality constraint is satisfied. The coarse-grain tasks reflect logical parallelism in the user program. It is easy to map each task to an independent thread (core) if the depended data is available in a shared memory space. Our percolation model creates additional fine-grain parallelism within each coarse-grain task. It is important to note that our percolation-based multi-grain parallelism is different from conventional multi-grain parallelism techniques that are used to combine task and data parallelism [5]. The fine-grain parallel tasks in the percolation model are exploited as separating memory access from computation operations within a coarse-grain task. The advantages of separating memory from computation are: (1) in addition to the fine-grain parallelism, we can also pipeline the different phases of memory access and computation tasks to hide the overhead of memory access, and (2) it provides an opportunity to elaborate the memory access tasks so that they are aware of memory hierarchy and transform non-linear memory access with high latency to linear memory access with low latency. A separate memory access task may reorganize (gather) the dispersed references in advance in the pipeline of tasks. Within memory hierarchy, the tasks for locality requirement may involve either collecting the data toward the cores where the task is enabled, called inward percolation, or sending/migrating the data away from the cores, called outward percolation. A percolation task scheduler causes the data to meet the corresponding threads just in time at the vicinity of the cores where the computation task is to be carried out.

2.3 Discussion

The key idea behind percolation model is to bring data close to computation just in time so that the computation thread can run to completion. For non-preemptive thread units such as in C64, it is important to reduce the number of stalls during execution. In percolation model rather than delaying or suspending a thread during execution, we avoid scheduling such threads that do not have locality constraints satisfied. Percolation model is closely related to prefetching, except that in prefetching, prefetch instructions are inserted within a thread that is ready to be scheduled, and the hope is that at the time the thread needs the data it will be available for use. Often prefetching instructions are either inserted too soon (in which case the prefetched data is evicted from the cache) or too late. In both cases the corresponding thread will stall the (non-preemptive) thread

unit on which it is running. Since C64 has no caches nor does it support preemptive hardware thread units, delaying or suspending active threads or prefetching is not well suited to improve performance and scalability of applications.

3 Percolation Programming

In this section, we discuss in detail two irregular applications (betweenness centrality with graph traversal in SSCA2 [6] and non-serial polyadic dynamic programming (DP) [7]) and show how to program them for percolation. There are three parts to percolation programming: (1) Collect data from non-contiguous locations just in time to obtain Just-In-Time Locality in the on-chip memory for the computation phase; (2) Compute the relevant information based on Just-In-Time Locality on-chip data; and (3) Finally, map the information thus computed back to off-chip memory. The first and last phase form the memory tasks and are mapped to helper threads by the runtime system. The second phase is the main computation phase that is mapped to a computation thread. Due to space limitation, we only present the pesudo code of SSCA2. It is important to keep in mind that the memory tasks and computation tasks are pipelined by the runtime system to reduce the critical path. Also, it is upto the programmer to create coarse grain parallelism (that includes multiple computation tasks and memory tasks) depending on what is being computed within the application.

3.1 Percolation Programming for SSCA2

In this section we describe our percolation programming for SSCA2. Recall that there are two phases in SSCA2: the BFS phase (See Figure 1) and BT phase [6]. To simplify the presentation we only describe the percolation programming for the BFS phase.The first step in percolation programming is to identify the memory tasks. Let us denote the set of the vertices that is being extended in the current queue (the ith level of BFS tree) as $V_i = \{v_{i1}, v_{i2}, ..., v_{ik}\}$. Let $N_j = \{w_{j1}, w_{j2}, ..., w_{jk_j}\}, 1 \leq j \leq k$ denote the neighboring set of vertices of a vertex v_{ij}. According to the algorithm, the unvisited neighbor vertices w ($d[w] = -1$) is added to the current queue and the vertices that is being extended in the shortest path ($d[w] = d[v] + 1$) is added to the set of predecessor $P[w]$. Note that the layout of these N_j in the adjacency array may be non-contiguous and have variable stride. Note that if we compact all the neighbors in one level into one large set: $UN_i = \bigcup_{1 \leq j \leq n} N_j$, the compacted non-contiguous memory region can be considered as a contiguous linear array so that it is easy to partition it among parallel threads. In our implementation, we do *not* explicitly perform such a compaction operation in the off-chip memory, but inside we use the Just-In-Time Locality principle.

The inward percolation consists of computing the start address and size of the neighboring vertices region in adjacency array of each vertex, and collecting neighboring vertices that is dispersed in the off-chip memory address (adjacency array) into a contiguous on-chip memory address. We also collect the corresponding elements in d, σ into a contiguous on-chip memory address. Notice that there is a producer-consumer relationship between the collection of neighboring vertcies and collection of d, σ. Also, the memory references of d, σ are discrete because the distribution of the neighboring

```
1  BFS(int v) {
2   int dv = d[v]; //length of the shortest path
3   int sigmav = sigma[v]; //the number of the shortest path
4   for (i = 0; i < NumEdges[v]; i++) {
5     w = Adjacent[index[v]+i];
6     if (d[w] < 0) {
7       d[w] = dv + 1;
8       sigma[w] = 0;
9     }
10    if (d[w] = dv + 1)
11      sigma[w] = sigmav + 1;
12  }
13 }
```

Fig. 1. The sequential BFS codes without percolation

vertices obeys a law of power in a scale free graph. Due to this property of scale-free graph we believe that the thread speculation techniques are not effective in practice. Once we compute the relevant information (d, σ) we write them back to off-chip memory using yet another memory task.

The complete parallel percolation program for BFS phase is shown in Figure 2. Once the programmer defines the coarse-grain parallel tasks the runtime system automatically divides these tasks into multiple sub-tasks and pipelines them. Obviously, the dependence between the sub-tasks is inherited from that specified in the user program. In order to achieve the pipeline of computation, inward and outward memory tasks, the union set UN_i is partitioned into multiple sub-blocks. When computation tasks are processing the data in block i, inward percolation memory tasks gather the data in block $i+1$ and the outward percolation memory tasks scatter the results that are generated using the data in block $i-1$.

3.2 Percolation Programming for DP

The dynamic programming (DP) algorithm in RNA secondary structure prediction [8] belongs to a type of non-serial polyadic dynamic programming [7]. We can use a simple recursive formulation to represent the computation:

$$m[i,j] = \begin{cases} min_{i \le k < j}\{m[i,j], m[i,k] + m[k+1,j]\} & 0 \le i < j < n \\ a(i) & i = j \end{cases} \quad (1)$$

The irregular behavior comes from the non-consecutive data dependence and irregular iteration domain which is a triangular space. Basically, we could use a blocking strategy to fill the matrix. In [9], the computation of a block is decomposed into combination of the depended blocks. According to the dependence, the computational sequence of blocks is from down to up and from left to right in the matrix, and each block depends on the blocks on both the same row and the same column. When both $A(0,1)$ and

```
1  /*three pipelined phase: (1) off-chip memory read;
2  (2) compuation (accessing on-chip memory);
3   (3) off-chip memory write.*/
4  BFS(int v) {
5    int offset = 0;
6    int turn = 0;
7    int dv = d[v];
8    int sigmav = sigma[v];
9    SPAWN_TASK{
10   for (i = 0; i < buffsize; i++)
11         buff[turn][i] = Adjacent[index[v]+offset+i];
12    offset += buffsize;
13    turn ^= 1;};
14   BARRIER_WAIT();
15   while (offset < NumEdges[v]) {
16     //1. off-chip memory read
17     SPAWN_TASK{
18     for (i = 0; i < buffsize; i++)
19         buff[turn][i] = Adjacent[index[v]+offset+i];
20     offset += buffsize;
21     turn ^= 1;};
22     SPAWN_TASK{
23     for (i = 0; i < buffsize; i++) {
24          w = buff1[i];
25          buff2[turn][i] = d[w];
26          buff3[turn][i] = sigma[w];
27      }
28      turn ^= 1;};
29      //(2). compuation (accessing on-chip memory);
30      SPAWN_TASK{
31      for (i = 0; i < buffsize; i++) {
32         if (buff2[turn][i] < 0) {
33                 buff2[turn][i] = dv+1;
34                 buff3[turn][i] += 0;
35         }
36         if (buff2[turn][i] == dv+1)
37                 buff3[turn][i] += sigmav;
38      }
39      turn ^= 1;};
40      //(3). off-chip memory write
41      SPAWN_TASK{
42      for (i = 0; i < buffsize; i++) {
43         w = buff[turn][i];
44         d[w] = buff2[i];
45         sigma[w] = buff3[i];
46       }
47       turn ^= 1;};
48       BARRIER_WAIT();
49    }
50 }
```

Fig. 2. BFS codes with percolation on IBM C64

$A(1,3)$ are combined to calculate $A(0,3)$, an ideal memory layout should look like: $A(0,1)$ is row-wise and $A(1,3)$ is column-wise. One strategy of block data layout is to store each block as both row wise and as column wise array layout. However, this doubles the memory usage which is not practical. We assume that the matrix is stored as a row-wise linear array. Thus, the stride of accesses in different row or column within each block is not constant.

According to the data dependence, the computation of block $A(i,j)$ needs to access other blocks $A(i,i), A(i,i+1), ..., A(i,j-1)$ and $A(i+1,j), A(i+2,j), ..., A(j,j)$. For example the program accesses $< A(0,0), A(0,3) >$, $< A(0,1), A(1,3 >$, $< A(0,2), A(2,3) >$ and $< A(0,3), A(3,3) >$ during the calculation of $A(0,3)$. We assume that the triangular DP matrix is stored as a linear array in off-chip memory. The percolation transformation is responsible to transform the non-contiguous access of a block into a contiguous access in on-chip memory. When a block $A(i,j)$ is computing, the percolation threads gather the non-contiguous elements in off-chip memory into multiple contiguous space in on-chip memory, then scatter the results to the corresponding locations in off-chip memory. The pipeline achieves Just-In-Time Locality, and the percolated data will be used immediately by the computation task. The unused data will never exist in on-chip memory at that time even if the spatial locality of cache mechanism is satisfied. For the example of computing block $A(0,2)$, when the program is percolating block $A(1,2)$ into on-chip memory, a conventional spatial locality optimization strategy or speculation may load the elements in block $A(1,3)$. Obviously, the current computation does not need $A(1,3)$ at all. Since C64 provides user programmable scratchpad, the runtime system ensures that such unnecessary data is not loaded into the on-chip memory.

4 Evaluation

We have implemented Just-In-Time Locality and percolation for the two programs in our many-core architecture C64 execution-driven simulation platform. The toolchain on C64 consists of an optimized GCC compiler, a thread execution runtime systems TNT [10] (Pthread-like) and a TNT-based OpenMP [11]. In this section we present our empirical results and compare them with the corresponding OpenMP programs on C64.

4.1 Empirical Results for SSCA2 with Percolation

For SSCA2 we represent the problem size in term of *S*, where the number of vertices is 2^S. Comparing the result with the OpenMP implementation, we can see that the percolation process shows a significant performance and scalability improvements (see Figure 3). Figure 4 illustrates the performance and scalability as we increase the number of threads for three different scales (i.e., the problem size). Using our approach we achieve almost linear speedups for all test cases when the number of threads is less than 32. For the test case with a problem size $S = 8$, the performance stops increasing when the number of threads reaches 128 because the number of available parallel sub-tasks is less than the number of hardware thread units. However, we improve the performance when the problem size is increased, i.e for $S = 9$ and 10. The degree of a vertex

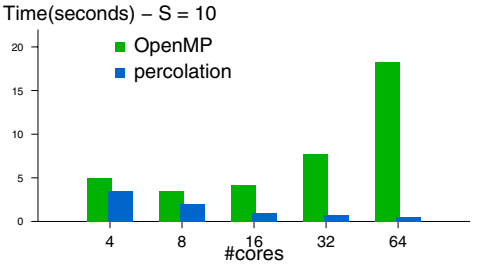


Fig. 3. Performance comparison of percolation with OpenMP. Left bars are openmp and the right bars are percolation.

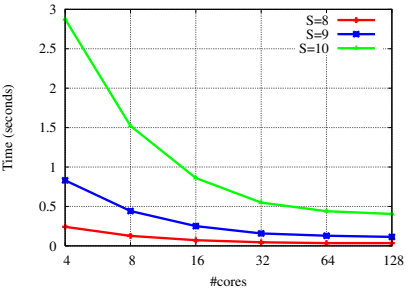
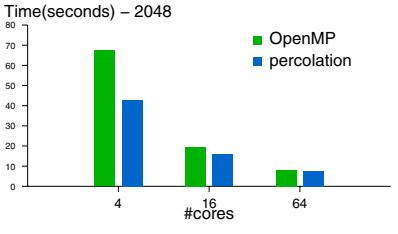
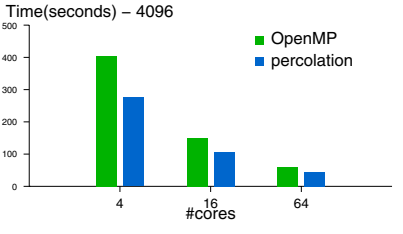


Fig. 4. Scalability of parallel betweenness centrality algorithm. The number of edges $E(n) = 8n$.



(a) matrix size = 2048×2048



(b) matrix size = 4096×4096

Fig. 5. Performance comparison of percolation with OpenMP. Left bars are OpenMP and the right bars are percolation.

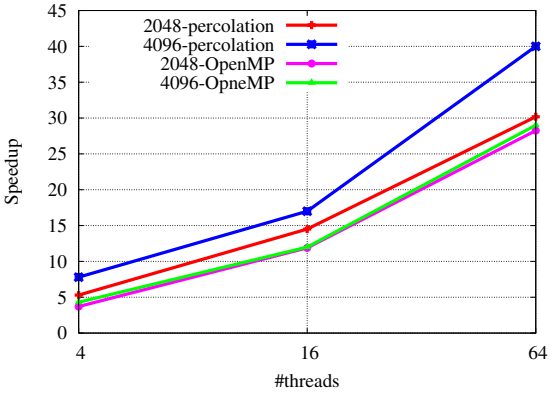


Fig. 6. Scalability results for matrix sizes of 2048×2048 and 4096×4096

determines the amount of parallelism that we can exploit. In percolation programming we leverage multi-grain parallelism to reduce the number of idle threads. On the other hand the maximum degree of a vertex is 64 for problem size $S = 8$. So the available

parallelism for this small problem size leads to a smaller performance on 128 threads. For $S = 9$ and 10, where the maximum vertex degrees are 94 and 348, we further improve the performance and scalability of the application.

4.2 Empirical Results for DP with Percolation

In order to highlight the advantage of separation of computation from memory operations, we compared with the performance of a baseline program implemented with directly blocking algorithm by OpenMP. As shown in Figure 5(a) and 5(b), the reconstructed program with percolation reduces the execution time for matrix sizes of 2048×2048 and 4096×4096. The effect of percolation for DP is not so significant with that for SSCA2. Note that in the implementation of DP we use blocking technique to re-organize the DP matrix so that it shows more inherent locality, which can not be observed in the irregular execution of SSCA2. Further, our percolation program improves the scalability slightly. Figure 6 reports the absolute speedup achieved by percolation and OpenMP programs.

5 Related Work

The percolation model has its deep root in the HTMT execution model proposed well over a decade ago as the basis of the world first (to the best of our knowledge) petaflops architect project [12]. The concept of percolation was developed early under HTMT project, and was first exposed in [13, 14]. Unlike the HTMT time, The work reported here is partly motivated by the challenging technology trend of modern large-scale many-core chip technology, and driven by the productive software technology available to us that is not available during HTMT project due to resource limitations.

In our parallel pipelining algorithm we overlap computation task with memory task. The concept of overlapping computation with I/O, network, and other long latency operations is an old concept. Prefetching techniques [15, 16, 17] and thread speculation [1, 18, 19] also use such overlapping concept. Most previous work on prefetching also focused on moving data (mostly contiguous data) from main memory to local memory (either to register or cache) prior to execution. In the previous prefetching or speculation, conceptually computation threads "pull" the data locally using prefetch instructions. In our method the local data determines which computation thread is ready to execute. In other words, data that is local to a core will "pull" computation thread to execute on the core. In prefetching there is no control on how much data to prefetch—prefetching too much or too less data can impact the performance. Besides, previous works do not discuss the impact of prefetching in the context of massive multithreading many-core. A variant of thread level speculation uses dependences by monitoring the reads and writes to memory locations. In producer-cosumer loop iterations, the speculative execution leads to a violation of dependence, then must roll back.

There have been several work on the optimization of irregular programs on parallel architectures. Recently Erez et.al [20] performed a comprehensive study of 4 irregular scientific computing applications on a streaming processor. Both their work and ours share the streaming programming style of gather-compute-scatter. The way to gather

data ahead makes our approach different from theirs. In [20] the streaming processor uses a DMA-style transfer, our approach utilizes the ample hardware thread units, where to hide the overhead of transformation is easier and requires less hardware cost.

6 Conclusion

Both computer architects and system developers is yet to evaluate the new many-core architectural features and show how such features can be effectively exploited when executing challenging irregular applications like graph traversal and dynamic programming in practice. This paper introduces Just-In-Time Locality and percolation model for improving the locality of irregular applications on C64 many-core architecture. However, our percolation model is not uncontroversial. For example, we did not discuss the role of "reusability" of data to be percolated. It is obvious that we should try to give higher priority to data that can have better reuse. Another is connection to load balancing: one obviousely need to coordinate the percolation (and Just-In-Time Locality) with the place where a thread is likely to be scheduled for execution. In an irregular code, we cannot expect load is evenly distributed among the processing cores - and runtime coordination of data movement and load balancing should be smooth. We can imagine cases where not all cores in a 100+ core chip can be kept usually busy all the time - as our experience has been for C64. In this case, some of the idle cores should be employeed to assist the percolation and coordination - an interesting research topic by its own right. Finally one fair doubt is the impact on percolation model on programmability - a question that does not has a short answer and we will have to leave it for future work.

Acknowledgment

We would like to thank all reviewers and the shepherd for improving this paper. The authors would like to acknowledge Russo Andrew and Ge Gan at CAPSL for their help. This work is partially supported through the support from NSFC (60633040), IBM, ET International, the National Science Foundation (CNS-0509332).

References

1. Steffan, J.G., Colohan, C.B., Zhai, A., Mowry, T.C.: A scalable approach to thread-level speculation. In: Proceedings of the 27th Annual International Symposium on Computer Architecture (2000)
2. Rauchwerger, L., Zhan, Y., Torrellas, J.: Hardware for speculative run-time parallelization in distributed shared memory multiprocessors. In: Proceedings of the 4th International Symposium on High-Performance Computer Architecture, p. 162 (1998)
3. Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., Chew, P.: Optimistic parallelism requires abstractions. In: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, pp. 211–222 (2007)
4. Zhu, W., Sreedhar, V.C., Hu, Z., Gao, G.R.: Synchronization state buffer: Supporting efficient fine-grain synchronization on many-core architectures. In: The 34th International Symposium on Computer Architecture (2007)

5. Gordon, M., Thies, W., Amarasinghe, S.: Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In: International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA (2006)
6. Bader, D.A.: Hpcs scalable synthetic compact applications 2 graph analysis (2006), http://www.highproductivity.org/SSCABmks.htm
7. Grama, A., Gupta, A., Karypis, G., Kumar, V.: Introduction to Parallel Computing. Addison Wesley, Reading (2003)
8. Zuker, M., Mathews, D.H., Turner, D.H.: Algorithms and Thermodynamics for RNA Secondary Structure Prediction: A Practical Guide. Kluwer Academic Publishers, Dordrecht (1999)
9. Tan, G., Feng, S., Sun, N.: Locality and parallelism optimization for dynamic programming algorithm in bioinformatics. In: SC 2006: Proceedings of the, ACM/IEEE conference on Supercomputing, p. 78. ACM, New York (2006)
10. Cuvillo, J., Zhu, W., Hu, Z., Gao, G.R.: Tiny threads: a thread virtual machine for the cyclops-64 cellular architecture. In: Fifth Workshop on Massively Parallel Processing (WMPP), held in conjunction with the 19th rnational Parallel and Distributed Processing System (2005)
11. Cuvillo, J., Zhu, W., Gao, G.R.: Landing openmp on cyclops-64: An efficient mapping of openmp to a many-core system-on-a-chip. In: The 3rd ACM International Conference on Computing Frontiers, Ischia, Italy (2005)
12. Gao, G.R., Likharev, K.K., Messina, P.C., Sterling, T.L.: Hybrid technology multi-threaded architecture. In: Proceedings of Frontiers 1996: The Sixth Symposium on the Frontiers of Massively Parallel Computation, pp. 98–105 (1996)
13. Amaral, J.N., Gao, G.R., Merkey, P., Sterling, T., Ruiz, Z., Ryan, S.: Performance prediction for the htmt: A programming example. In: TFP3 1999 (1999)
14. Gao, G., Amaral, J.N., Marquez, A., Theobald, K.: A refinement of the "htmt" program execution model. Technical report, CAPSL, University of Delaware (1998)
15. Wu, Y.: Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. In: PLDI 2002: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, pp. 210–221. ACM, New York (2002)
16. Zhang, Z., Torrellas, J.: Speeding up irregular applicaitons in shared-memory multiprocessors: Memory binding and group prefetching. In: 22nd International Symposium on Computer Architecture (1995)
17. Mowry, T., Gupta, A.: Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. Journal of Parallel and Distributed Computing 12, 87–106 (1991)
18. Collins, J.D., Tullsen, D.M., Wang, H., Shen, J.P.: Dynamic speculative precomputation. In: The 34th Annual International Symposium on Microarchitecture (2001)
19. Zhang, W., Tullsen, D.M.: Accelerating and adapting precomputation threads for efficient prefetching. In: 3th International Symposium on High Performance Computer Architecture (2007)
20. Erez, M., Ahn, J.H., Gummaraju, J., Rosenblum, M., Dally, W.J.: Executing irregular scientific applications on stream architectures. In: ICS 2007: Proceedings of the 21st annual international conference on Supercomputing, pp. 93–104. ACM, New York (2007)

Exploring the Optimization Space of Dense Linear Algebra Kernels

Qing Yi[1] and Apan Qasem[2]

[1] University of Texas at San Anstonio
[2] Texas State University

Abstract. Dense linear algebra kernels such as matrix multiplication have been used as benchmarks to evaluate the effectiveness of many automated compiler optimizations. However, few studies have looked at collectively applying the transformations and parameterizing them for external search. In this paper, we take a detailed look at the optimization space of three dense linear algebra kernels. We use a transformation scripting language (POET) to implement each kernel-level optimization as applied by ATLAS. We then extensively parameterize these optimizations from the perspective of a general-purpose compiler and use a standalone empirical search engine to explore the optimization space using several different search strategies. Our exploration of the search space reveals key interaction among several transformations that must be considered by compilers to approach the level of efficiency obtained through manual tuning of kernels.

1 Introduction

Compiler optimizations have often targeted the performance of linear algebra kernels such as matrix multiplication. While issues involved in optimizing these kernels have been extensively studied, difficulties remain in terms of effectively combining and parameterizing the necessary set of optimizations to consistently achieve the level of portable high performance as achieved manually by computational specialists through low-level C or assembly programming. As a result, user applications must invoke high-performance domain-specific libraries such as ATLAS [10] to achieve a level of satisfactory efficiency. ATLAS produces extremely efficient linear algebra kernels through a combination of domain/kernel specific optimization, hand-tuned assembly, and an automated empirical tuning system that uses direct timing to select the best implementations for various performance-critical kernels. Libraries such as ATLAS are necessary because native compilers can rarely provide a similar level of efficiency, either because they lack domain-specific knowledge about the input application or because they cannot fully address the massive complexity of modern architectures.

To improve the effectiveness of conventional compilers, many empirical tuning systems have been developed in recent years [1,4,8,9,11,14]. These systems have demonstrated that search-based tuning can significantly improve the efficacy of many compiler optimizations. However, most research in this area has focused

J.N. Amaral (Ed.): LCPC 2008, LNCS 5335, pp. 343–355, 2008.

on individual or a relatively small set of optimizations. Few have collectively parameterized an extensive set of optimizations and investigated the interactions among them. One impediment to parameterizing a large class of transformations is that, as yet, we have no standard representation for parameterized optimizations and their search spaces. Because of this, most tuning systems consist of highly specialized code optimizers, performance evaluators and search engines. Hence, exploring a larger search space comes with the burden of implementing additional parameterized transformations. This extra overhead has, to some degree, limited the size of the search space investigated by any one tuning system. In this paper, we describe a system in which we interface a parameterized code transformation engine with an independent search engine. We show that by leveraging the complementary strengths of these two modules we are able to explore the search space of a large collection of transformations.

While most compilers understand well the collection of code optimizations that are required to achieve high performance, it is often the details in combining and collectively applying the optimizations that determine the overall efficiency of the optimized code. In our previous work [12], we have used POET, a transformation scripting language, to implement all the kernel-level optimizations as applied by ATLAS for three dense linear algebra kernels: *gemm*, *gemv*, and *ger*. Our previous work has achieved comparable, and sometimes superior, performance to those of the best hand-written assembly within ATLAS [12]. The previous work, however, utilized kernel-specific knowledge obtained from ATLAS when applying the optimizations and when searching for the best-performing kernels. In this paper, we extensively parameterize the optimization spaces from the perspective of a general-purpose compiler. We then use an independent search engine to explore this parameter space to better understand the delicate interplay between transformations. Such interactions must be considered by a compiler when combining and orchestrating different program transformations to approach a similar level of efficiency as achieved by ATLAS. The contributions of this paper include:

1. parameterization of an extensive set of optimizations that need to be considered by a general-purpose compiler to achieve high performance;
2. integration of an independent search engine and a program transformation engine;
3. empirical exploration of the search space that reveals key interaction among optimizations.

2 Related Work

A number of successful empirical tuning systems provide efficient library implementations for important scientific domains, such as those for dense and sparse linear algebra [3,5], signal processing [6,7] and tensor contraction [2]. POET [12] targets general-purpose applications beyond those targeted by domain-specific research and complements domain-specific research by providing an efficient transformation engine that can make existing libraries more readily portable to different architectures.

Several general-purpose autotuning tools can iteratively re-configure well-known optimizations according to performance feedback of the optimized code [1,4,8,9,11,14]. Both POET and the parameterized search engine described in this paper can be easily integrated with many of these systems. POET supports existing iterative compilation frameworks by providing an output language for parameterizing code optimizations for empirical tuning.

The work by Yotov et al. [13] also studied the optimization space of the *gemm* kernel in ATLAS. However, their work used the ATLAS code generator to produce optimized code. Because the ATLAS code generator is carefully designed by computational specialists based on both architecture- and kernel-specific knowledge, the optimization space they investigated does not represent the same degrees of freedom that a general-purpose compiler must face. In contrast, we use the POET language to parameterize the different choices typically faced by general-purpose compilers and investigate the impact and interactions of the different optimization choices.

3 Orchestration of Optimizations

We used a transformation scripting language named POET to implement an extensive set of optimizations necessary to achieve the highest level of efficiency for several ATLAS kernels [12]. As shown in Fig. 1, a POET transformation engine includes three components: a language interpreter, a transformation library, and a collection of front-end definitions which specialize the transformation library for different programming languages such as C, C++, FORTRAN, or Assembly. The transformation engine takes as input an optimization script from an analyzer (in our case, a developer) and a set of parameter configurations from a search driver. An optimized code is output as the result, which is then empirically tested and measured by the search driver until a satisfactory implementation is found. For more details, see [12].

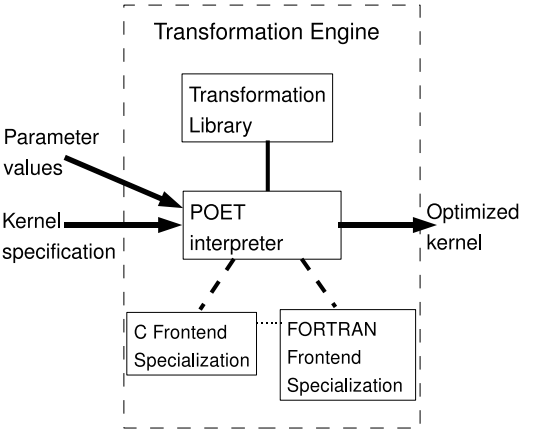


Fig. 1. POET transformation engine

The optimization scripts that we developed for the ATLAS kernels have been parameterized to reflect the degrees-of-freedom that a model-driven compiler must face when orchestrating general-purpose optimizations. The optimizations focus on the efficient management of registers and arithmetic operations and are invoked by higher-level routines in ATLAS after cache-level optimizations have already been applied by ATLAS. Fig. 2 shows the reference implementations of

```
void ATL_USERMM(int M,
  int N,int K,double alpha,
  const double *A, int lda,
  const double *B, int ldb,
  double beta, double *C,
  int ldc) {
  int i, j, l;
  for (j=0; j<N; j+=1)  {
    for (i=0; i<M; i+=1) {
      C[j*ldc+i] =
          beta*C[j*ldc+i];
      for (l=0;l<K;l+=1) {
        C[j*ldc+i]+=
        A[i*lda+l]*B[j*ldb+l]; }
  } } }
```

(a) *gemm* kernel

```
void ATL_dgemvT(int M,
  int N,double alpha,
  const double *A, int lda,
  const double *X,int incX,
  double beta,double *Y,
  int incY) {
  int i, j;
  for (i=0; i<M; i+=1) {
    Y[i] = beta * Y[i];
    for (j=0;j<N;j+=1) {
      Y[i] +=
        A[i*lda+j]*X[j];
  } }
```

(b) *gemv* kernel

```
void ATL_dger(int M,
  int N, double alpha,
  const double *X,int incX,
  const double *Y,int incY,
  double *A, int lda) {
  int i, j;
  for (j=0; j<N; j+=1) {
    for (i=0;i<M;i+=1) {
      A[j*lda+i] +=
        X[i]*Y[j*incY];
    }
  }
}
```

(c) *ger* kernel

Fig. 2. Reference implementations of ATLAS kernels

various ATLAS kernels. The corresponding higher-level routines perform identical computations but operate on larger matrices that may not fit in cache and perform cache blocking (for matrix multiplication, data copying is additionally applied to matrices A and B) before invoking the kernel implementations.

ATLAS has specialized each kernel implementation to take advantage of the known blocking strategy. We used optimization scripts to similarly specialize our kernel implementations. Our future work will use POET to integrate cache-level blocking directly with low-level kernel optimizations.

Our optimizations include two groups: register reuse optimizations (loop unroll&jam and scalar replacement) and arithmetic optimizations(SSE vectorization, strength reduction, loop unrolling, and memory prefetch). Our optimizations were carefully orchestrated to avoid introducing any inefficiency.

3.1 Register Reuse Optimizations

We applied two optimizations, unroll-and-jam and scalar replacement, to promote register reuse. Fig. 3 shows the result of applying unroll-and-jam and scalar replacement to the *gemm* kernel in Fig. 2(a). For *gemv* and *ger* (shown in Fig. 2(b) and (c)), we similarly applied unroll-and-jam to the outer loops, and scalar replacement to all the matrix/vector accesses.

We parameterized the optimizations with an unroll factor for each loop and the ordering between the two optimizations. For example, the optimizations for *gemm* (Fig 2(a)) were parameterized with three parameters: uJ (unroll factor for loop J), uI (unroll factor for loop I), and $permuteReg$ (the ordering between unroll-and-jam and scalar replacement for different matrices). In Fig. 3, the code on the left shows the result of applying unroll-and-jam followed by scalar replacement to all matrices; the code on the right shows the result of reverting the ordering of unroll-and-jam with scalar replacement for matrices A and B.

Most tuning systems treat loop unroll factors as part of the empirical search space. However, few systems (including ATLAS) have used empirical tuning to determine the ordering between optimizations. Most model-driven compilers perform unroll-and-jam before scalar-replacement, which is the right decision in a

```
void ATL_USERMM(const int M, const int N, const int K,
          const double alpha,const double *A,const int lda,
          const double *B,const int ldb,const double beta,
          double *C, const int ldc)
{
   int i, j, l;   double *pA0, *pA00, *pB0,*pB00,*pC0,*pC00;
   pB0=B; pC0=C;
   for (j = 0; j < NB; j += 2)  {
      pA0 = A; pC00=pC0;
      for (i = 0; i < MB; i += 2) {
        c_buf_0_0=*pC00; c_buf_1_0=*(pC00+ldc); c_buf_0_1=*(pC00+1); c_buf_1_1=*(pC00+ldc+1);
        c_buf_0_0 = beta * c_buf_0_0; c_buf_1_0 = beta * c_buf_1_0;
        c_buf_0_1 = beta * c_buf_0_1; c_buf_1_1 = beta * c_buf_1_1;
        pA00=pA0; pB00=pB0;
        for (l = 0; l < KB; l +=1) {             | if A,B scalarRepl is done first
          ----------------------------------    ------------------------------------------
          | a_buf_0 = *pA00;                |    | a_buf_0 = *pA00; b_buf_0=*pB00;          |
          | a_buf_1 = *(pA00+KB);           |    | c_buf_0_0 +=  a_buf_0 * b_buf_0;         |
          | b_buf_0 = *pB00;                |    | a_buf_0 = *pA00; b_buf_0=*(pB00+KB);     |
          | b_buf_1 = *(pB00+KB);           |    | c_buf_1_0 +=  a_buf_0 * b_buf_0;         |
          | c_buf_0_0 +=  a_buf_0 * b_buf_0;| ==>| a_buf_0 = *(pA00+KB); b_buf_0=*pB00;     |
          | c_buf_1_0 += a_buf_0 * b_buf_1; |    | c_buf_0_1 +=  a_buf_0 * b_buf_0;         |
          | c_buf_0_1 +=  a_buf_1 * b_buf_0;|    | a_buf_0 = *(pA00+KB); b_buf_0=*(pB00+KB);|
          | c_buf_1_1 += a_buf_1 * b_buf_1; |    | c_buf_1_1 +=  a_buf_0 * b_buf_0;         |
          ----------------------------------    ------------------------------------------
            pA00+=1; pB00+=1;
        }
        *pC00=c_buf_0_0; *(pC00+ldc)=c_buf_1_0; *(pC00+1)=c_buf_0_1; *(pC00+ldc+1)=c_buf_1_1;
        pA0 += KB; pC00 += 1;
     }
     pB0 += KB; pC00+=ldc;
   }
}
```

Fig. 3. *gemm* optimized with unroll&jam, scalar replacement, and strength reduction

majority of situations. However, as shown in Fig. 3, unroll-and-jam significantly increased the number of different memory locations accessed at each iteration of loop l. These memory references subsequently require a large number of scalar variables during scalar replacement. For example, if we set the unroll factors to be 12 for loop I in Fig. 3 then a total of 12 scalar variables will be required for matrix A. The large number of scalar variables could potentially disrupt register allocation at a later stage and cause performance break-down. In contrast, if scalar replacement for matrices A and B are applied before unroll-and-jam, scalar variables are reused across different iterations of the original loop, reducing register pressure. However, if scalar replacement is always performed first, the reuse of scalar variables could create artificial dependences that disrupt instruction scheduling at a later stage. Scalar replacement can also disable subsequent application of unroll-and-jam, and the repetitive load of the scalar variables can be a source of inefficiency. We believe that performing unroll-and-jam before scalar replacement is the correct decision in most cases. However, we parameterized their ordering for the purpose of empirically investigating their interaction.

3.2 SSE Vectorization

Some Intel and AMD processors provide specialized floating point SSE registers that allow two double-precision or four single-precision floating point values

to be operated simultaneously, potentially doubling or quadrupling the peak MFLOPS of a computer. SSE vectorization therefore could significantly boost the performance of user applications.

We implemented SSE vectorization support in POET and applied it to all three kernels where architecture allows. Before applying the optimization, all floating-point operations need to be translated into three-address code, a SSE register must be allocated to each floating-point scalar variable, and an innermost loop must be identified to be vectorized. Our POET optimization scripts applies SSE vectorization to all kernels based on kernel-specific knowledge about their dependence constraints. All required knowledge, however, can be automatically discovered by a compiler via loop dependence analysis. We parameterized SSE vectorization with two parameters: the number of available SSE registers and the size (length) of each SSE register. These parameters can be automatically determined when an application is ported to a new architecture. The optimization is automatically turned off if there are not enough SSE registers to vectorize the given code.

3.3 Strength Reduction and Loop Unrolling

To promote efficiency of arithmetic operations, two additional optimizations, strength reduction and loop unrolling, need to be applied to the result of SSE vectorization. Strength reduction significantly reduces the cost of matrix/vector references inside loops. Loop unrolling ensures that sufficient number of instructions are available to support pipelining of different functional units. Both optimizations need to be applied after SSE vectorization because both could disable the vectorization of loops. Fig. 3 shows the result of applying strength-reduction to *gemm*. For microprocessors that offer hardware support for dynamic scheduling of instructions, strength reduction combined with loop unrolling could be sufficient for satisfactory instruction-level efficiency.

We parameterized both optimizations with two parameters: the unroll factor for the innermost loop, and the ordering between the two transformations. Using Fig. 3 as example, if unrolling of loop l is performed after strength reduction, the pointer induction variables (e.g., pA00,pB00) introduced by strength reduction are incremented more frequently (every iteration of the original loop) than necessary (every iteration of the unrolled loop). In contrast, applying loop unrolling before strength reduction may cause the pointer induction variables (e.g., pA00) to be incremented only once in the unrolled loop but by a much larger offset (e.g., by 288 instead of by 1 or 2). We have parameterized the ordering of these two optimizations to study their interactions.

3.4 Memory Prefetch

The last optimization that we performed on the kernels is memory prefetch, which loads data ahead of time so that the latency of memory operations can be hidden when possible. Memory prefetch can reduce the cost of memory accesses only if the memory bandwidth is not already saturated. It is therefore necessary

to selectively prefetch only those data that will be used in the near future just enough time ahead.

The POET library allows us to insert prefetch instructions in a variety of locations. We have thus parameterized each prefetch optimization with two decisions: where to insert prefetch instructions, and what data to prefetch. For example, for *gemm*, prefetch is parameterized with three configurations: prefetch at each iteration of loop I the next cache block of matrix A; prefetch at each iteration of loop J the vector of matrix B accessed in the next iteration of loop J; prefetch at each iteration of loop J the vector of matrix C accessed in the next iteration of loop J. Different prefetch configurations can also be combined to accomplish the best effect.

4 A Parameterized Search Engine for Automatic Tuning

We designed and implemented a parameterized search engine (PSEAT) to navigate optimization search spaces with greater flexibility and efficiency. In most existing autotuning systems the search module is tightly coupled with the transformation engine. PSEAT is designed to work as an independent search engine and provides a search API that can be used by other autotuning frameworks. This section discusses design features of PSEAT and its integration with POET.

```
100                 # maximum number of program evaluations
3                   # number of dimensions in the search space
R 1 16              # range : 1 .. 16
P 4                 # permutation : sequence length 4
E 2 8 16            # enumerated : two possible value 8 and 16
```

Fig. 4. Example configuration file for PSEAT

Input. Input to PSEAT is a configuration file that describes the search space of optimization parameters. Fig. 4 shows an example configuration file. The syntax for describing a search space is fairly simple. Each line in the configuration file describes one search dimension. A dimension can be one of three types: *range* (R), *permutation* (P) or *enumerated* (E). *range* is used to specify numeric transformation parameters such as tile sizes and unroll factors. *permutation* specifies a transformation sequence and is useful when searching for the best phase-ordering. An *enumerated* type is a special case of the *range* type. It can be used to describe a dimension where only a subset of points are feasible within a given range. An example of an *enumerated* type is the prefetch distance in software prefetching. In addition, PSEAT supports inter-dimensional constraints for all three dimension types. For example, if the unroll factor of an inner loop needs to be smaller than the tile size of an outer loop then this constraint is specified using a simple inequality within the configuration file.

Information specific to a search algorithm is specified elsewhere. For example, for simulated annealing the *alpha* and *beta* factors for each dimension is specified

in a separate file. The parameters for the search algorithm have been deliberately kept separate to make the search space representation more general. Both the configuration file and the search parameter file can be written by hand or automatically generated by a transformation engine. This feature facilitates the use of PSEAT with model-based search strategies.

Program Evaluation. The procedure for program evaluation (*compile-run-measure*) depends on a number of factors and is usually different for different platforms and applications. Hence, incorporating a module for program evaluation makes the search engine less portable. To address this problem, we moved the task of program evaluation away from the search engine and into a set of scripts that are bundled together with PSEAT. These scripts are, to a great degree, self-customizing. They probe a particular machine, *a la* ATLAS, to gather necessary information for program evaluation. The scripts then interpret the output from the search module and deliver the data in the desired format to the tool responsible for applying the transformations. Once the program has been evaluated, a script gathers the performance data and feeds it to the search engine. In integrating PSEAT with POET we used the scripts to transform the search engine output to a command line configuration that POET can recognize. We used ATLAS timers to measure the performance of the kernels and PSEAT scripts to extract the relevant performance data.

Search Algorithm. PSEAT implements a number of search strategies including genetic algorithm, direct search, window search, taboo search, simulated annealing and random search. We include *random* in our framework as a benchmark search strategy. A search algorithm is considered effective only if it does better than random on a given search space.

5 Experimental Results

Our previous work has already compared the performance of POET-optimized kernels with those of ATLAS and the Intel `icc` compiler [12]. The goal of this study is to take a more extensive look at the optimization space of dense linear algebra kernels (*dgemm*, *dgemv* and *dger*). We ran experiments on two platforms. The configurations for these two machines are presented in Table 1(a). Each experiment is repeated three times and the mean from these three runs is reported.

Table 1(b) describes the optimization search space for the three kernels. Number of search dimensions is eight for *dgemm* and seven for *dgemv* and *dger* (*dgemv* and *dger* have one fewer loop to unroll than *dgemm*). PERM1 covers the ordering of four transformations: unroll-and-jam and scalar replacement of three different matrices/vectors (A,B,C for *gemm*; A,X,Y for *dgemv*/ *dger*). The range of feasible values for PERM1 is between 0 and 23 ($4! - 1$), which covers all valid permutations of the four transformations. PERM2 covers the ordering of two transformations and hence works as a binary dimension. The values for the

Table 1. Experimental design

	Core 2 Duo	Opteron
Proc.	2.2 GHz Intel Core 2 Duo	2.2 GHz AMD Dual Core
SSE#	16	16
L1	32 KB, 2-way 64 Byte/line	64 KB, 2-way 64 Byte/line
L2	4 MB, 4-way 64 Byte/line	1 MB, 2-way 64 Byte/line
Compiler	gcc 4.0.1	gcc 4.0.1

(a) Experimental platforms

Parameter	Type	Description	Values
MU	R	unroll factor for loop M	1-M/2
NU	R	unroll factor for loop N	1-N/2
KU	R	unroll factor for loop K	1-K/2
PF	R	memory prefetch	1-5
SSENO	E	number of SSE registers	(0,8,16)
SSELEN	E	length of SSE registers	(8,16)
PERM1	P	ordering of unroll&jam and scalar replacement for A,B/X,C/Y	0-23
PERM2	P	ordering of strength reduction and inner-loop unrolling	0-1

(b) Search spaces

unroll factors are between 1 and one-half of the loop iteration numbers. The software prefetching parameter (PF) has five different values that determine which and where array references are prefetched. The search spaces for all three kernels are fairly large even by empirical tuning standards. For example, for a 100x100 matrix the 8-dimensional search space for *dgemm* will have over 120 million feasible points!

5.1 Search Strategy Comparison

We explore the search spaces using three different search strategies: simulated annealing (`anneal`), direct search (`direct`) and random search (`random`). Fig. 5 shows performance of the three strategies on two platforms. Each figure shows the best performance achieved as we progressively increase the number of program evaluations. To make the comparison fair, the initial point is picked randomly and the same initial point is used for all search strategies.

From these figures, `anneal` has a clear advantage over the other two search strategies. It discovers the best value in all but one of the six cases. Because `direct` converges to a local minima much sooner than `anneal`, it often does not discover the best value, but it does discover *good* values more quickly. Given that reducing the number of program evaluations is a key consideration for effective autotuning, `direct` might well be the search strategy of choice for these kernels. The performance of all three strategies also tend to level-off after a certain number of evaluations. This indicates that large regions in the search space may have very little performance variation. Therefore, model-guided pruning can help make the searches more efficient.

5.2 Search Space Exploration

We performed exhaustive search on various cross-sections of each search space to gain insight into their characteristics. This section summarizes our key findings.

SSE and PERM1: Our experiments reveal a strong interaction between SSE vectorization and the order in which unroll-and-jam and scalar replacement are applied. Fig. 6 shows the effect on *dgemm* performance on the Core2Duo and Opteron as we vary the SSE and PERM1 parameters. Each figure displays the

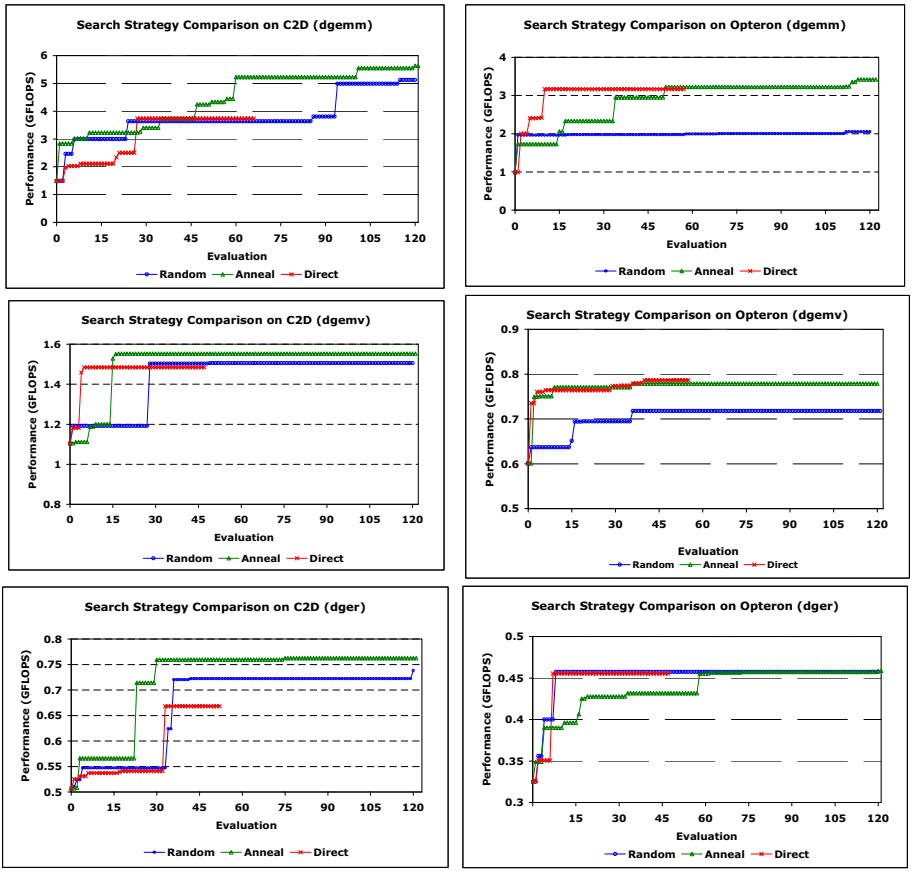


Fig. 5. Search strategy comparison

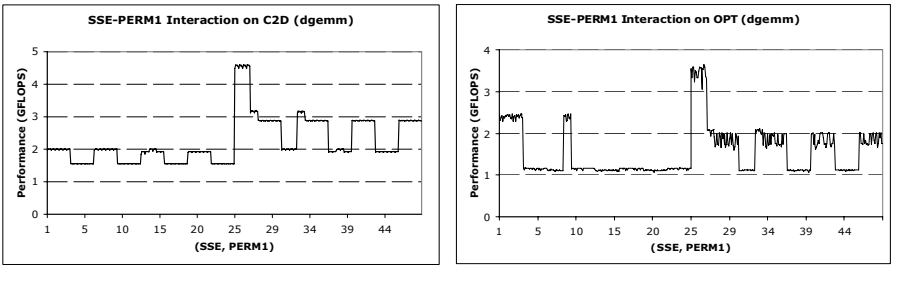


Fig. 6. SSE-PERM1 interaction for *dgemm*

linearized value of SSE and PERM1 along the x-axis (i.e., each x-value corresponds to a pair of values: one for SSE and one for PERM1). There is a clear indication that irrespective of the value of PERM1, vectorization has a positive

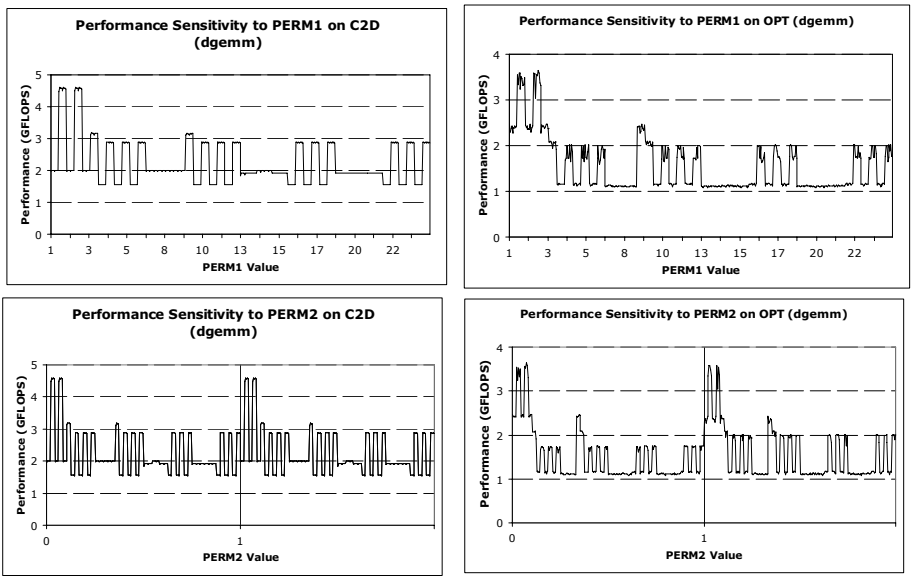


Fig. 7. *dgemm* performance sensitivity

effect on performance. The best performance on both platforms is achieved when the value along the x-axis is between 25-26, which correspond to the cases where SSE is activated and PERM1 is `[SA,UJ,SB,SC]` and `[SA,UJ,SC,SB]`, respectively. On these values, the relative ordering of unroll-and-jam and scalar replacement opens up additional opportunities for vectorization. This interplay is somewhat subtle and may not be apparent to a general-purpose compiler that uses only static heuristics to analyze the program. Picking the right values for the two parameters in question can lead to almost a factor of two performance improvement for *dgemm* on the Core2Duo.

PERM1 and PERM2: Fig. 7 shows performance sensitivity as the relative ordering of unroll-and-jam and scalar replacement for different matrices is changed. As expected, the ordering has a significant impact on performance, The best ordering occurs when unroll-and-jam is performed after scalar replacement of references in matrix A but before scalar replacement of matrix C and matrix B. This ordering is different from the one that we manually picked in our previous work [12], where we speculated that applying scalar-replacement of B after unroll-and-jam may increase the register pressure too much to create problems on some architectures. However, as it turns out, both these machines are capable of handling the excess register pressure. These results point out the limitations of static analysis and reiterate the need for empirical tuning.

Our experiments also show that the ordering of strength reduction and inner loop unrolling has very little impact on performance. On both platforms we observe very similar performance for both orderings of transformations.

6 Conclusions

This paper examines the optimization space of three linear algebra kernels by combining a program transformation engine with an empirical search tool. Our system is flexible and portable and is a small step towards better integration of existing autotuning systems. Our exploration of the search space show significant interaction among several transformations. In particular, the interaction between SSE vectorization and the ordering of unroll-and-jam and scalar replacement had not been revealed in any of the previous studies. The results of this study can be utilized by general-purpose compilers to orchestrate the set of transformations discussed in this paper to achieve improved performance.

References

1. Agakov, F., Bonilla, E., Cavazos, J., Franke, B., Fursin, G., O'Boyle, M., Thomson, J., Toussaint, M., Williams, C.: Using machine learning to focus iterative optimization. In: International Symposium on Code Generation and Optimization, 2006 (CGO 2006), New York, NY (2006)
2. Baumgartner, G., Auer, A., Bernholdt, D.E., Bibireata, A., Choppella, V., Cociorva, D., Gao, X., Harrison, R.J., Hirata, S., Krishnamoorthy, S., Krishnan, S., Lam, C.-C., Lu, Q., Nooijen, M., Pitzer, R.M., Ramanujam, J., Sadayappan, P., Sibiryakov, A.: Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. Proc. IEEE, Special Issue on Program Generation, Optimization, and Adaptation 93(2) (2005)
3. Bientinesi, P., Gunnels, J.A., Myers, M.E., Quintana-Orti, E., van de Geijn, R.: The science of deriving dense linear algebra algorithms. ACM Transactions on Mathematical Software 31(1), 1–26 (2005)
4. Chen, C., Chame, J., Hall, M.: Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In: CGO, San Jose, CA, USA (March 2005)
5. Demmel, J., Dongarra, J., Eijkhout, V., Fuentes, E., Petitet, A., Vuduc, R., Whaley, C., Yelick, K.: Self adapting linear algebra algorithms and software. Proc. IEEE, Special Issue on Program Generation, Optimization, and Adaptation 93(2) (2005)
6. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proc. IEEE, Special Issue on Program Generation, Optimization, and Adaptation 93(2) (2005)
7. Püschel, M., Moura, J.M.F., Johnson, J., Padua, D., Veloso, M., Singer, B.W., Xiong, J., Franchetti, F., Gačić, A., Voronenko, Y., Chen, K., Johnson, R.W., Rizzolo, N.: SPIRAL: Code generation for DSP transforms. Proc. IEEE, Special Issue on Program Generation, Optimization, and Adaptation 93(2) (2005)
8. Qasem, A., Kennedy, K., Mellor-Crummey, J.: Automatic tuning of whole applications using direct search and a performance-based transformation system. The Journal of Supercomputing 36(2), 183–196 (2006)
9. Stephenson, M., Amarasinghe, S.: Predicting unroll factors using supervised classification. In: CGO, San Jose, CA, USA (March 2005)
10. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated empirical optimization of software and the ATLAS project. Parallel Computing 27(1–2), 3–35 (2001)
11. Whaley, R.C., Whalley, D.B.: Tuning high performance kernels through empirical compilation. In: The 2005 International Conference on Parallel Processing (June 2005)

12. Yi, Q., Whaley, C.: Automated transformation for performance-critical kernels. In: ACM SIGPLAN Symposium on Library-Centric Software Design, Montreal, Canada (October 2007)
13. Yotov, K., Li, X., Ren, G., Cibulskis, M., DeJong, G., Garzaran, M., Padua, D., Pingali, K., Stodghill, P., Wu, P.: A comparison of empirical and model-driven optimization. In: Proceedings of the SIGPLAN 2003 Conference on Programming Language Design and Implementation, San Diego, CA (June 2003)
14. Zhao, Y., Yi, Q., Kennedy, K., Quinlan, D., Vuduc, R.: Parameterizing loop fusion for automated empirical tuning. Technical Report UCRL-TR-217808, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory (December 2005)

Author Index

Amato, Nancy M. 304
Archambault, Roch 217
Ayguadé, Eduard 31

Baghsorkhi, Sara S. 1
Bai, Tongxin 217
Becker, Aaron 279
Bianco, Mauro 304
Burger, Doug 64
Burtscher, Martin 109
Buss, Antal A. 304

Caşcaval, Călin 232
Chakravorty, Sayantan 279
Cledat, Romain 124
Coons, Katherine 64

Ding, Chen 217
Duchateau, Alexandre X. 187

Gao, Guang R. 141, 331
Gao, Yaoqing 217
Garzarán, María Jesús 187
Gonzàlez, Marc 31
Gross, Thomas R. 172
Gu, Xiaoming 217

Hermenegildo, Manuel V. 47, 94
Hironaka, Ken 249
Hwu, Wen-mei W. 1, 16

Kalé, Laxmikant 279
Kapur, Deepak 94
Kejariwal, Arun 232
Kulkarni, Milind 109
Kumar, Tushar 124

Lathara, Melvin 1
Lhoták, Ondřej 47
Li, Xiao-Feng 264
Li, Zhiyuan 292

Marron, Mark 94
Martorell, Xavier 31
Matsakis, Nicholas D. 172
McKinley, Kathryn S. 64
Méndez-Lojo, Mario 47
Meyers, Russell 292
Mycroft, Alan 156

Oancea, Cosmin E. 156
Osheim, Nissa 80

Padua, David 187
Pande, Santosh 124
Pennings, Maikel 316
Pingali, Keshav 109
Prountzos, Dimitrios 109

Qasem, Apan 343

Rajopadhye, Sanjay 80
Rauchwerger, Lawrence 304, 316
Robatmili, Behnam 64
Rostron, Dave 80
Rus, Silvius 316

Saito, Hideo 249
Sarkar, Vivek 141
Shaw, Jonathan 200
Shen, Xipeng 200
Sidelnik, Albert 187
Smith, Timmie G. 304
Sreedhar, Vugranam C. 141, 331
Sreeram, Jaswanth 124
Stefanovic, Darko 94
Stone, Sam S. 16
Stratton, John A. 16
Strout, Michelle Mills 80

Takahashi, Kei 249
Tan, Guangming 331
Tanase, Gabriel 304
Taura, Kenjiro 249
Thomas, Nathan L. 304

Ueng, Sain-Zee 1

Vujić, Nikola 31

Wang, Ligang 264
Wilmarth, Terry 279
Wu, Peng 232

Yang, Chen 264
Yi, Qing 343

Zhang, Chengliang 217
Zhang, Yuan 141
Zhu, Weirong 141