Sanjay Rajopadhye
Michelle Mills Strout (Eds.)

LNCS 7146

Languages and Compilers for Parallel Computing

24th International Workshop, LCPC 2011
Fort Collins, CO, USA, September 2011
Revised Selected Papers

Springer

Sanjay Rajopadhye Michelle Mills Strout (Eds.)

Languages and Compilers for Parallel Computing

24th International Workshop, LCPC 2011
Fort Collins, CO, USA, September 8-10, 2011
Revised Selected Papers

Springer

Volume Editors

Sanjay Rajopadhye
Michelle Mills Strout
Colorado State University
Computer Science Department
Fort Collins, CO 80523-1873, USA
E-mail: sanjay.rajopadhye@colostate.edu
E-mail: mstrout@cs.colostate.edu

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-36035-0 e-ISBN 978-3-642-36036-7
DOI 10.1007/978-3-642-36036-7
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012955139

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

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

On behalf of the organizers of LCPC 2011, it is our pleasure to present the proceedings of the 24th International Workshop on Languages and Compilers for Parallel Computing (LCPC), held September 8–10, 2011, at Colorado State University, Fort Collins Colorado, USA. Since 1988, the LCPC workshop has emerged as a major forum for sharing cutting-edge research on all aspects of parallel languages and compilers, as well as related topics including runtime systems and tools. The scope of the workshop spans the theoretical and practical aspects of parallel and high-performance computing, and targets parallel platforms including concurrent, multithreaded, multicore, accelerator, multiprocessor, and cluster systems. With the advent of multicore processors, LCPC is particularly interested in work that seeks to transition parallel programming into the computing mainstream. This year's LCPC workshop was attended by 66 registrants from across the globe. The Third Annual Concurrent Collections Workshop was also co-located with the conference, and held on Wednesday, September 7, 2011.

This year, the workshop received 52 submissions from institutions all over the world. Of these, the Program Committee (PC) selected 19 papers for presentation at the workshop, representing an acceptance rate of 36%. Each selected paper was presented in a 30-minute slot during the workshop. In addition, 19 submissions were selected for presentation as posters during a 90-minute poster session. Each submission received at least three reviews. The PC held an all-day meeting on July 22, 2011, to discuss the papers. During the discussions, PC members with a conflict of interest with the paper were asked to temporarily leave the meeting. Decisions for all PC-authored submissions were made by PC members who were not coauthors of any submissions.

We were fortunate to have two keynote speakers at this year's LCPC workshop. Vikram Adve, University of Illinois, Urbana-Champaign, gave a talk entitled "Parallel Programming Should Be – And Can Be – Deterministic-by-Default," and Vijay Saraswat of IBM Research spoke on "Constrained Types: What Are They and What Can They Do for You." Both talks were very well received and elicited very interesting questions and discussions. In addition, the technical sessions led to animated and lively discussions. Two papers were awarded the best student presentation award (tie) and one was selected for the best student poster award. The winers were Max Grossman (undergraduate student from Rice University) for "Dynamic Task Parallelism with a GPU Work-Stealing Runtime System" and Aleksandar Prokopec, PhD student at École Polytechnique Fédérale de Lausanne, Switzerland for "Lock-Free Resizeable Concurrent Tries." The best poster award winner was Shreyas Ramalingam, of the University of Utah, for "Automating Library Specialization Using Compiler-Based Autotuning."

We would like to thank the many people whose dedicated time and effort helped make LCPC 2011 a success. The hard work invested by the PC and external reviewers in reviewing the submissions helped ensure a high-quality technical program for the workshop. The Steering Committee members and the LCPC 2010 Organizing Committee provided valuable guidance and answered many questions. All participants in the workshop contributed directly to the technical vitality of the event either as presenters or as audience members. We would also like to thank workshop sponsors, the National Science Foundation, the Department of Energy, The Air Force Office of Scientific Research, and our industrial sponsors, Intel and Google, for supporting the best student paper and poster presentation awards. Finally, the workshop would not have been possible without the tireless efforts of the entire local arrangements team at Colorado State University.

Sanjay Rajopadhye
Michelle Mills Strout

Organization

Steering Committee

Rudolf Eigenmann	Purdue University, USA
Alex Nicolau	UC Irvine, USA
David Padua	University of Illinois, USA
Lawrence Rauchwerger	Texas A&M University, USA

Workshop Chairs

Sanjay Rajopadhye	Colorado State University, USA
Michelle Mills Strout	Colorado State University, USA

Workshop Program Committee

Wim Bohm	Colorado State University, USA
John Cavazos	University of Delaware, USA
Bor-Yuh Evan Chang	University of Colorado, USA
Keith Cooper	Rice University, USA
Guang Gao	University of Delaware, USA
Mary Hall	University of Utah, USA
Hironori Kasahara	Waseda University, Japan
Keiji Kimura	Waseda University, Japan
Xioaming Li	University of Delaware, USA
John Mellor-Crummey	Rice University, USA
Sriram Sankaranarayanan	University of Colorado, USA
Vivek Sarkar	Rice University, USA
Jeremy Siek	University of Colorado, USA

Sponsors

Air Force Office of Scientific Research (AFOSR)
Department of Energy (DoE)
Intel Corp
National Science Foundation (NSF)

Table of Contents

Automatic Scaling of OpenMP Beyond Shared Memory 1
Okwan Kwon, Fahed Jubair, Seung-Jai Min, Hansang Bae, Rudolf Eigenmann, and Samuel P. Midkiff

A Methodology for Fine-Grained Parallelism in JavaScript Applications 16
Jeffrey Fifield and Dirk Grunwald

Evaluation of Power Consumption at Execution of Multiple Automatically Parallelized and Power Controlled Media Applications on the RP2 Low-Power Multicore 31
Hiroki Mikami, Shumpei Kitaki, Masayoshi Mase, Akihiro Hayashi, Mamoru Shimaoka, Keiji Kimura, Masato Edahiro, and Hironori Kasahara

Double Inspection for Run-Time Loop Parallelization 46
Michael Philippsen, Nikolai Tillmann, and Daniel Brinkers

A Hybrid Approach to Proving Memory Reference Monotonicity 61
Cosmin E. Oancea and Lawrence Rauchwerger

OpenCL as a Programming Model for GPU Clusters 76
Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee

CellCilk: Extending Cilk for Heterogeneous Multicore Platforms 91
Tobias Werth, Silvia Schreier, and Michael Philippsen

OPELL and PM: A Case Study on Porting Shared Memory Programming Models to Accelerators Architectures 106
Joseph B. Manzano, Ge Gan, Juergen Ributzka, Sunil Shrestha, and Guang R. Gao

Optimizing the Concurrent Execution of Locks and Transactions 124
Justin E. Gottschlich and JaeWoong Chung

A Study of the Usefulness of Producer/Consumer Synchronization 141
Hao Lin, Hansang Bae, Samuel P. Midkiff, Rudolf Eigenmann, and Soohong P. Kim

Lock-Free Resizeable Concurrent Tries 156
Aleksandar Prokopec, Phil Bagwell, and Martin Odersky

Fine-Grained Treatment to Synchronizations in GPU-to-CPU Translation 171
Ziyu Guo and Xipeng Shen

A Mutable Hardware Abstraction to Replace Threads 185
Sean Halle and Albert Cohen

Dynamic Task Parallelism with a GPU Work-Stealing Runtime System 203
Sanjay Chatterjee, Max Grossman, Alina Sbîrlea, and Vivek Sarkar

A Code Merging Optimization Technique for GPU 218
Ryan Taylor and Xiaoming Li

Static Compilation Analysis for Host-Accelerator Communication Optimization 237
Mehdi Amini, Fabien Coelho, François Irigoin, and Ronan Keryell

Scheduling Support for Communicating Parallel Tasks 252
Jörg Dümmler, Thomas Rauber, and Gudula Rünger

Polytasks: A Compressed Task Representation for HPC Runtimes 268
Daniel Orozco, Elkin Garcia, Robert Pavel, Rishi Khan, and Guang R. Gao

Detecting False Sharing in OpenMP Applications Using the DARWIN Framework 283
Besar Wicaksono, Munara Tolubaeva, and Barbara Chapman

Author Index 299

Automatic Scaling of OpenMP Beyond Shared Memory[*]

Okwan Kwon[1], Fahed Jubair[1], Seung-Jai Min[2], Hansang Bae[1], Rudolf Eigenmann[1], and Samuel P. Midkiff[1]

[1] Purdue University, USA
[2] Lawrence Berkeley National Laboratory, USA
SJMin@lbl.gov, {kwon7,fjubair,baeh,eigenman,smidkiff}@purdue.edu

Abstract. OpenMP is an explicit parallel programming model that offers reasonable productivity. Its memory model assumes a shared address space, and hence the direct translation - as done by common OpenMP compilers - requires an underlying shared-memory architecture. Many lab machines include 10s of processors, built from commodity components and thus include distributed address spaces. Despite many efforts to provide higher productivity for these platforms, the most common programming model uses message passing, which is substantially more tedious to program than shared-address-space models. This paper presents a compiler/runtime system that translates OpenMP programs into message passing variants and executes them on clusters up to 64 processors. We build on previous work that provided a proof of concept of such translation. The present paper describes compiler algorithms and runtime techniques that provide the automatic translation of a first class of OpenMP applications: those that exhibit regular write array subscripts and repetitive communication. We evaluate the translator on representative benchmarks of this class and compare their performance against hand-written MPI variants. In all but one case, our translated versions perform close to the hand-written variants.

1 Introduction

The development of high-productivity programming environments that support the development of efficient programs on distributed-memory architectures is one of the most pressing needs in parallel computing, today. Many of today's parallel computer platforms have a distributed memory architecture, as most likely will future multi-cores.

Despite many approaches [1–4] to provide improved programming models, the state of the art for these platforms is to write explicit message-passing programs, using MPI. This process is tedious, but allows high-performance applications to

[*] This work was supported, in part, by the National Science Foundation under grants No. 0751153-CNS, 0707931-CNS, 0833115-CCF, and 0916817-CCF.

be developed. Because expert software engineers are needed, many parallel computing platforms are inaccessible to the typical programmer. In this paper we describe the first automatic system that translates OpenMP to MPI, and we introduce the key compiler and runtime techniques used in our system. OpenMP has emerged as a standard for programming shared memory applications due to its simplicity. OpenMP programmers often start from serial codes and incrementally insert OpenMP directives to obtain parallel versions. Our aim is to extend this ease of parallel programming beyond single shared-memory machines to small-scale clusters.

Like OpenMP on SDSMs, this approach supports standard OpenMP, without the need for programmers to deal with the distribution of data, as in HPF, UPC, and Co-array Fortran. Previous work [5] has shown that programs could be systematically translated by hand from OpenMP to MPI and achieve good performance. In this work, we aim to automate this translation, using a novel split between the compiler and the runtime system. At an OpenMP synchronization point where coherence across threads is required, the compiler informs the runtime system about array sections written and read before and after the synchronization point, respectively; from this information the runtime system computes the inter-thread overlap between written and read array sections and generates the necessary communication messages. The compiler identifies and utilizes several properties exhibited by many scientific parallel benchmarks for efficient generation of MPI code. Our initial goal, and the focus of this paper, is the class of applications that involve regular array subscripts and repetitive communication patterns. Our translator accepts the OpenMP version 2 used by the benchmarks.

This paper makes the following contributions. We

i. introduce an automatic approach for translating shared-address-space programs, written in OpenMP, into message-passing programs.
ii. introduce a novel compile/runtime approach to generate inter-thread communication messages.
iii. describe the implementation of the translator in the Cetus [6] compiler infrastructure.
iv. present the evaluation of five benchmarks' performance compared to their hand-written MPI counterpart.

We measured the performance on up to 64 processors. Three out of five benchmarks (JACOBI, SPMUL and EP) show comparable performance to the hand-coded MPI counterpart. One benchmark (CG) follows the performance of the hand-coded MPI version up to 16 processors and reaches 50% of the hand-coded MPI performance on 64 processors. The fifth benchmark (FT) has superior performance in the hand-coded MPI version, which we study in detail.

The remainder of the paper is organized as follows: Section 2 describes the translation system, Section 3 presents and discusses performance results and Section 4 discusses related work, followed by conclusions and future work in Section 5.

2 The OpenMP to Message-Passing Translator

The translator converts a C program using OpenMP parallel constructs into a message-passing program. This is accomplished by first changing the OpenMP program to a SPMD form; the resulting code represents a *node program* for each thread[1] that operates on partitioned data. The SPMD-style representation has the following properties: (i) the work of parallel regions is evenly divided among the processes; (ii) serial regions are redundantly executed by all participating processes; and (iii) the virtual address space of shared data is replicated on all processes. Shared data is not physically replicated - only the data actually accessed by a process is physically allocated on that process. Next, the compiler performs array data flow analyses, which collect written and read shared array references for the node program in the form of symbolic expressions. The compiler inserts function calls to pass these expressions to the runtime system. Finally, at each synchronization point, the runtime system uses the expressions from the compiler to compute the inter-thread overlap between written and read data and determines the appropriate MPI communication messages.

The translation system guarantees that shared data is coherent at OpenMP synchronization constructs only, as defined by OpenMP semantics. A data race that is not properly synchronized in the input OpenMP program might lead to unspecified results of the translated code.

The translation system is implemented using the Cetus compiler infrastructure. We also developed the runtime library. To support interprocedural analysis, we use subroutine inline expansion. Figure 1 shows an example of an input OpenMP code and the translated code.

2.1 SPMD Generation

The input OpenMP program is converted to an initial SPMD form that is suitable for execution on distributed-memory systems. First, the compiler parses OpenMP directives to identify serial and parallel regions. Second, it identifies shared variables in these regions. Third, it partitions the iteration space of each parallel loop among participating processes. The following subsections describe in detail these three steps.

2.1.1 OpenMP Directives Parsing

The compiler parses OpenMP directives in the input program and represents them internally as *Cetus annotations*. Cetus annotations identify code regions and express information such as shared and private variables as well as reduction idioms.

The compiler inserts a barrier annotation after every OpenMP work-sharing construct that is terminated by an implicit barrier, to indicate the presence of that barrier. No barriers are present at `nowait` clauses. We refer to the code executed between two consecutive barriers as a *SPMD block*. The `omp master` and `omp single` constructs in parallel regions are treated as serial regions.

[1] OpenMP thread corresponds to MPI process.

```
for (step = 0; step < nsteps; step++) {
  x = 0;

  /* loop 1 */
  #pragma omp parallel for reduction(+:x)
  for (i = 0; i < M; i++)
    x += A[i] + A[i-1] + A[i+1];

  printf("x = %f\n", x);

  /* loop 2 */
  #pragma omp parallel for
  for (i = 0; i < N; i++)
    A[i] = ...;
}
```

```
for (id = 0; id < nprocs; id++) {
  ompd_def(1, id, A, lb2[id], ub2[id]);
  ompd_use(1, id, A, lb1[id]-1, ub1[id]+1);
}

for (step = 0; step < nsteps; step++) {
  x = 0;

  /* loop 1 */
  #pragma cetus parallel for shared(A) private(i)
reduction(+:x)
  x_tmp = 0;
  for (i = lb1[proc_id]; i <= ub1[proc_id]; i++)
    x_tmp = A[i] + A[i-1] + A[i+1];
  #pragma cetus DEF()
  #pragma cetus USE(A[lb1[proc_id]-1:ub1[proc_id]+1])
  #pragma cetus barrier /* barrier 0 */
  ompd_allreduce(&x_tmp, &x, ...);

  printf("x = %f\n", x);

  /* loop 2 */
  #pragma cetus parallel for shared(A) private(i)
  for (i = lb2[proc_id]; i <= ub2[proc_id]; i++)
    A[i] = ...;
  #pragma cetus DEF(A[lb2[proc_id]:ub2[proc_id]])
  #pragma cetus USE(A[lb1[proc_id]-1:ub1[proc_id]+1])
  #pragma cetus barrier /* barrier 1 */
  ompd_ptp(1);
}
```

Fig. 1. Typical code of an OpenMP program and its translated code: The runtime library functions ompd_allreduce and ompd_ptp compute and perform the necessary communication, based on the array sections given by the ompd_def/use calls. These calls have been hoisted before the "step" loop, as they are loop invariant in this example. Cetus pragma annotations are comments only.

The compiler identifies reduction operations. Explicit `omp reduction` clauses in the source program are directly represented by a reduction annotation. A widely used practice in OpenMP programs is to code array or scalar reductions using the `omp critical` or `omp atomic` directive constructs, where each thread updates the global copy of the shared data using its own local copy. The compiler also identifies these reduction patterns and provides the corresponding critical section with a reduction annotation. The translator uses an existing algorithm in the Cetus infrastructure to recognize these patterns. The compiler then converts all these reductions to MPI reduction operations, as described in Section 2.3.1. In our benchmarks, the compiler recognized all `omp critical` sections as reduction patterns. Figure 1 shows how the OpenMP directives in the input code are represented using Cetus annotations in the translated code.

2.1.2 Shared Data Identification

In preparation for communication analysis, the compiler must identify all shared data. Variables in OpenMP parallel regions are shared by default - the OpenMP data clauses, such as `omp private` and `shared` are not sufficient to identify shared variables. The compiler uses an interprocedural algorithm described previously [7] for finding shared variables in parallel regions. The compiler updates each Cetus annotation to store the shared variables and private variables to be used in subsequent translation steps.

2.1.3 Work Partitioning

The compiler uses block distribution to divide the iteration space of a parallel loop into chunks of approximately equal size, and then associates each chunk with a process. The compiler inserts the code to perform this partitioning as early as possible in the program. Under OpenMP semantics, using block distribution, even in the presence of an OpenMP `schedule` clause, always leads to correct results[2]. Block distribution also simplifies the symbolic expressions representing the summarization of affine array subscripts. Before partitioning a parallel loop, the compiler checks if all array subscripts for shared write accesses are affine expressions. In the presence of a non-affine write array subscript (e.g., an indirect array access), the parallel loop is not partitioned, i.e., conservatively serialized. In ongoing work, we are developing techniques for handling irregular write memory accesses.

2.2 Array Data Flow Analyses

Using Data Flow Analyses, the compiler collects information about written and read shared array references in the SPMD program that may need to be communicated. For each barrier N, the compiler symbolically summarizes shared written array references in the preceding SPMD block that reach barrier N and future, shared read array references exposed to barrier N. The compiler passes this information to the runtime system, which generates communication, as explained in Section 2.3.

The compiler first constructs a *parallel control flow graph (PCFG)*. The PCFG is essentially the control flow graph (CFG) generated for a node program. Every statement is represented by a node in the PCFG. Every Cetus annotation is attached to its corresponding node, except for barriers, which are converted into special nodes. The compiler then performs the following data flow analyses: *Reaching-All Definitions* analysis to compute *DEF* sections and *Live-Any* analysis to compute *USE* sections. A DEF section summarizes shared written array references produced in the SPMD block preceding barrier node N. Because serial regions are executed redundantly, their produced (written) shared data are available to all future consumer processes; therefore DEF sections need not include those references.

The array data flow analyses are based on Cetus array section analysis [6], which summarizes the set of array elements (i.e., the sub-arrays) that are written or read by a program statement. The array section analysis makes use of symbolic range analysis, which collects, at each statement, a map from integer-typed scalar variables to their symbolic value ranges, which are represented by a symbolic lower bound (lb) and upper bound (ub). DEF and USE sections are represented

[2] See Chapter 2.5.1 of the OpenMP Specification. The other scheduling methods will be supported in our ongoing work.

using Cetus section [lb:ub] symbolic expression. Cetus array sections conservatively describe non-affine array subscripts, such as indirect accesses. Live-Any analysis overestimates by assuming all data is read when summarizing a non-affine read array subscript using the infinity section expression [-INF:+INF]. Communication is optimized for [-INF:+INF] USE data sections by using MPI collective communication operations (see Section 2.3.1). Strided affine subscripts are represented using Cetus unit-stride array section and thus overestimated as well. Overestimation for USE sections can cause extra communication but no incorrect behavior. Note that DEF sections are never overestimated; as an `omp for` containing irregular write array subscripts is serialized (see section 2.1.3).

After performing the array data flow analyses, the entry and the exit of each node N in the PCFG have the following information: (i) $shared\_write\_set(N)$ is the set of shared written arrays in the parallel region prior to node N; (ii) $shared\_read\_set(N)$ is the set of shared arrays that is upward exposed at node N; (iii) tuples of DEF sections $\langle DEF_i \rangle_V, 0 \leq i \leq (p-1)$, DEF$_i$ is the DEF section computed for process i and p is the total number of participating processes, $\forall V \in shared\_write\_set(N)$; (iv) tuples of USE sections $\langle USE_j \rangle_W, 0 \leq j \leq (p-1)$, USE$_j$ is the USE section computed for process j, $\forall W \in shared\_read\_set(N)$. The compiler represents a tuple of array sections using one symbolic expression, where this expression is a function of the processor number (proc_id). The compiler inserts Cetus annotations to show computed DEF and USE sections' tuples at barriers in the output code as shown in Figure 1.

2.3 Communication Generation

At a barrier, inter-process communication is needed if the DEF section for one process overlaps with a USE section for any other process. The compiler has identified DEF and USE sections for all barriers. The compiler inserts function calls to pass these sections to the runtime system; it also inserts a function call at each barrier notifying the runtime system to generate communication. The runtime system's role is to compute the inter-process overlapping array sections and to generate MPI point-to-point communication if the overlap is non-empty. An exception of this is when generating MPI collective communication; the compiler recognizes that using collective communication instead of using point-to-point communication is possible. The following subsections discuss in detail the process of generating communication for each case.

2.3.1 Collective Communication Generation

The compiler generates collective communication in two cases: at reduction operations and at barriers that have a [-INF:+INF] USE section overlapping with a DEF section. These cases are recognized at compile time, where the compiler inserts function calls to generate collective communication at the corresponding synchronization points. The compiler does not need to inform the runtime about DEF and USE sections in separate function calls; this information is passed in the arguments of the collective communication function call.

The `ompd_allreduce()` call is used for reductions, which invokes `MPI_Allreduce()` collective communication. In the case of a reduction clause in an `omp for` loop, the compiler creates a local copy of the reduction variable, which is updated in the body of the `omp for` loop. An `ompd_allreduce()` is inserted after the `omp for` loop to combine the local copies into the global reduction variable. Figure 1 shows an example of this case. If a reduction originates from a critical section (see Section 2.1.1), the compiler replaces the critical section code with `ompd_allreduce()`.

The `ompd_allgatherv()` is used at barriers that have a DEF section overlapping with a [-INF:+INF] USE section, which calls `MPI_Allgatherv()` collective communication. `MPI_Allgatherv()` gathers the data chunks modified in each process and distributes them to all processes. Note that the relevant information about DEF sections is passed in the arguments of the `ompd_allgatherv()`, as they are used as arguments for the `MPI_Allgatherv()`.

2.3.2 Point-to-Point Communication Generation

For each barrier, the compiler passes DEF and USE symbolic sections to the runtime system. The runtime system computes the inter-process intersections of DEF and USE sections and generates the needed MPI point-to-point communication. At each barrier N, for each shared array V $\in$ (shared_write_set(N) $\cap$ shared_read_set(N)) and $\langle USE_j \rangle_V$ is not [-INF:+INF], the compiler inserts the following runtime function calls: (i) *ompd_def()* to pass DEF$_i$ section computed for each process i; (ii) *ompd_use()* to pass USE$_j$ section computed for each process j; (iii) *ompd_ptp()* to notify the runtime system that a barrier N has been reached. Each barrier is given a unique integer identifier by the compiler.

The translator exploits the *repetitiveness* property of DEF and USE sections to make the final selection of communication at runtime more efficient. A repetitive section is a section whose symbolic expression is invariant with respect to a outer serial loop(s), i.e., the same shared array references are accessed by the same process during the execution of all iterations of outer serial loops. An algorithm developed in prior work [8] is used at compile-time to verify the repetitiveness property of array sections. The compiler inserts function calls *ompd_def()* and *ompd_use()* prior to the outermost serial loop at which array sections are repetitive. This information needs to be passed only once; the runtime system can repetitively reuse these sections. In our tested benchmarks, the compiler has recognized all array sections as repetitive sections.

When reaching a barrier B, the function call *ompd_ptp()* triggers the runtime system to determine and generate MPI point-to-point communication. Algorithm 1 depicts the underlying algorithm of this function. In the algorithm, the intersections $\langle send\_msg_j \rangle$ and $\langle receive\_msg_i \rangle$ represent array sections that need to be sent and received by a process. Because DEF and USE sections are repetitive, the intersections $\langle send\_msg_j \rangle$ and $\langle receive\_msg_i \rangle$ are also repetitive. The runtime system utilizes this property to compute these intersections once and then repeatedly uses them for subsequent iterations of outer loops. The *is_updated* flag indicates if DEF or USE sections have changed. The *is_updated* flag is set to

Algorithm 1. Determining and generating point-to-point communication messages. The total number of processors is p.

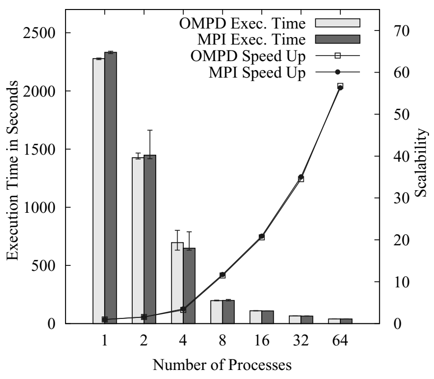
```
if $is\_updated = TRUE$ then
    for $j = 0 \to p - 1$ do
        if $j \neq proc\_id$ then
            $send\_msg_j \leftarrow DEF_{proc\_id} \cap USE_j$
            $receiver_j \leftarrow j$
        end if
    end for
    for $i = 0 \to p - 1$ do
        if $i \neq proc\_id$ then
            $receive\_msg_i \leftarrow USE_{proc\_id} \cap DEF_i$
            $sender_i \leftarrow i$
        end if
    end for
    $is\_updated \leftarrow FALSE$
end if
$count \leftarrow 0$
for all $send\_msg_j$ do
    if $send\_msg_j \neq EMPTY$ then
        $MPI\_Isend(send\_msg_j, receiver_j, ...)$
        $count \leftarrow count + 1$
    end if
end for
for all $receive\_msg_i$ do
    if $receive\_msg_i \neq EMPTY$ then
        $MPI\_Irecv(receive\_msg_i, sender_i, ...)$
        $count \leftarrow count + 1$
    end if
end for
$MPI\_Waitall(count, ...)$
```

$TRUE$ by the *ompd_def()* and *ompd_use()* functions and set to $FALSE$ when the intersections are computed. These intersections can be reused as long as *is_updated* is $FALSE$, i.e., *ompd_def()* and *ompd_use()* are not re-invoked.

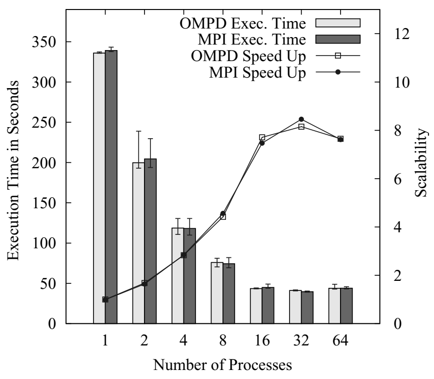
The runtime maintains $\langle send\_msg_j \rangle$ and $\langle receive\_msg_i \rangle$ sections information by storing a list of lower and upper bounds and their corresponding barrier identifier. The translator uses asynchronous point-to-point communication to overlap the processes' waiting times. Figure 1 shows the inserted function call to the runtime library at $Barrier1$. No function call is inserted at $Barrier0$ because shared_write_set($Barrier0$) ∩ shared_read_set($Barrier0$) is empty.

3 Performance Evaluation

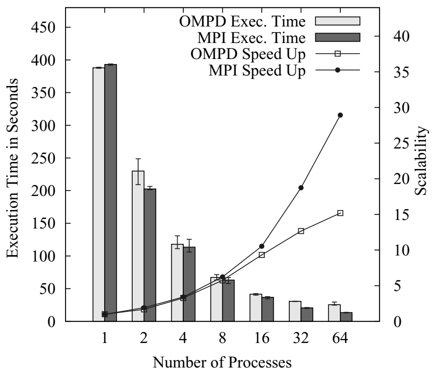
This section evaluates our translation system on five benchmarks: The JACOBI and SPMUL kernel benchmarks and CG, EP and FT from the NAS Parallel



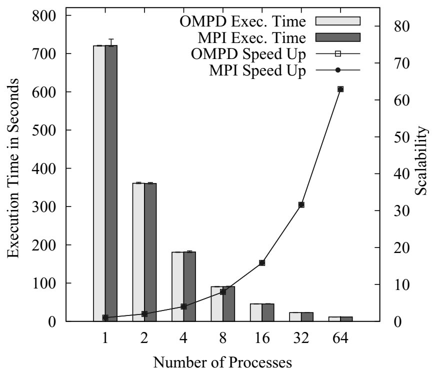
(a) JACOBI



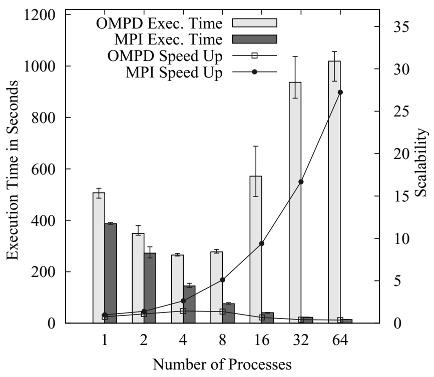
(b) SPMUL



(c) CG



(d) EP



(e) FT

Fig. 2. Performance comparison between translated and MPI programs: the translated JACOBI, SPMUL and EP perform very close to the hand-tuned MPI counterparts. The performance difference in the CG programs results from different data partitioning schemes of the translator and MPI programs. The FT benchmark exposes opportunities for future work.

Benchmarks (NPB) 2.3 [9, 10]. Each translated benchmark is compared to its hand-tuned MPI counterpart. We use publicly available OpenMP and MPI benchmarks. For JACOBI, we used an OpenMP version [11] and a fast MPI version [12] that uses asynchronous communication. For SPMUL, the main computation part of the OpenMP version is based on an implementation using compressed sparse rows (CSR) in SPARSKIT [13], and the MPI version is also available online [14].

We performed subroutine inlining and found that it did not significantly affect our performance; the difference in execution time between the original and inlined programs is less than 0.1% for all benchmarks except for FT, where it is around 4%.

3.1 Experimental Setup

We evaluated the performance on 32 nodes of a community cluster with two MPI processes per node, for a total of 64 MPI processes. The nodes are connected by an InfiniBand network, which provides 10 Gbps of bandwidth. Each node has two quad-core Intel Xeon E5410 processors running at 2.33Ghz with 6MB L2 caches per processor and 16GB of memory. The system is running a 64-bit Linux kernel, version 2.6.18, and MVAPICH2 version 1.5 MPI. We compiled all programs with gcc64 4.4.0 at optimization level 3.

We timed each benchmark ten times and recorded average, minimum and maximum execution times. Figure 2 shows the average execution time for each benchmark, with error bars indicating minimum and maximum times. We attribute the variations primarily to network traffic of other jobs running on different nodes of the system. Graphs have an axis labeled "scalability" – this is the ratio of the execution time of the serial version of the benchmark running on one processor to the execution time of the translated benchmark running on the number of processors indicated on the X axis.

3.2 JACOBI Kernel

JACOBI is a kernel benchmark that solves Laplace equations using Jacobi iteration. It uses two 2,048 $\times$ 2,048 matrices with 100,000 iterations. Figure 2(a) compares the performance of the translated program and the hand-tuned MPI variant. The runtime intersection between USE and DEF data results in the same communication volume that the MPI version generates. The MPI version is one of the fast versions that use asynchronous communication among neighbor processes. Since our runtime system also uses asynchronous communication for all messages, we are able to match the MPI variant's performance.

3.3 SPMUL Kernel

SPMUL is a kernel benchmark that performs sparse matrix-vector multiplications. We used a 100,000 $\times$ 100,000 sparse matrix as input. Figure 2(b) compares the performance of the translated program and the hand-tuned MPI program. The former program has two barriers. One causes point-to-point communication to be generated and the other causes collective communication to be generated. The communication volume at the first barrier is zero because the intersection between DEF and USE data sections is empty; no communication is performed. At the second barrier, the communication is done by `ompd_allgatherv()` because the code has irregular data reads on arrays via an indirection vector, so the USE data section is treated conservatively as [-INF:+INF]. The MPI program uses block-striped partitioning and has one communication point where a collective communication using `MPI_Allgatherv()` occurs. Even though the translated program has two barriers while the MPI program has one, the actual runtime communication patterns of the two programs are identical; the communication volume at the first barrier is zero.

3.4 CG Benchmark

The CG benchmark implements a conjugate gradient method, with a sparse matrix-vector multiplication taking most of the time. The benchmark has irregular reads on arrays via an indirect vector because of its unstructured grid computations. In addition to the irregular reads, there is an `omp for` loop that has irregular writes. Our translator treats this loop as a serial region, as described in Section 2.1.3. This serial loop execution does not significantly affect the benchmark performance because it exists outside of the outermost time-stepping serial loop.

The OpenMP version of the CG benchmark includes several reduction clauses in `omp for` work sharing constructs. The reductions are translated into `ompd_allreduce()` communication by the translator.

The translated and the MPI programs have different data partitioning schemes. The MPI version partitions the sparse input matrix using 2-D block distribution, whereas our translation scheme partitions the input matrix using a simple 1-D block distribution. This is because only the outer-most loop of the matrix-vector multiply computation is parallelized in the OpenMP version. Compared to the 1-D distribution, the 2-D block distribution of the input matrix requires a smaller number of processors for reduction communication; it comes at the expense of one additional transpose operation of the input matrix at the end of the matrix-vector multiplication. The overhead of the 2-D distribution's transpose operation is not proportional to the number of processors. Therefore, the 1-D distribution is superior for small numbers of processors. As the number of processors increases, however, the 2-D distribution shows better communication performance.

To further improve the performance of the translated version, optimization using 2-D distribution of the input matrix is needed, and it requires the recognition of matrix-vector multiplication patterns in the source program.

3.5 EP Benchmark

The EP benchmark is a highly parallel kernel, often used to explore the upper limit of floating point performance of a parallel system. The OpenMP version has one `omp critical` section that performs an array reduction and our implementation transforms it into an `ompd_allreduce()` function call. In addition, one `omp for` work sharing construct has a reduction clause on scalar variables; these reductions are translated into two `ompd_allreduce()` functions. The translated code is similar to the MPI version of the EP benchmark. The MPI version uses three `MPI_Allreduce()` function calls. The resulting performance of the translated code is nearly identical to the hand-tuned MPI variant, as shown in Figure 2(d).

3.6 FT Benchmark

The FT benchmark is a kernel that solves a 3-D partial differential equation using Fast Fourier Transforms. The code tests the communication performance of a system because it is very message intensive. As shown in Figure 2(e), the MPI version outperforms the translated variant substantially. Our version shows speedup on up to 8 processors, after which the communication overhead dominates.

The input FT code for our translator has a manual modification splitting `dcomplex` into `real` and `imaginary` variables. This is done because the current array section analysis does not support structured data types. The split affects cache behavior and generates two separate MPI communication calls. We measured the split version and the structured version of the translated FT program and found that the split version is around 20% slower than the structured version on one MPI process.

We identified two issues related to communication. First, the MPI version uses collective communication, but our implementation uses less efficient point-to-point communication. Thus, making the runtime detection of the collective communication pattern is one of the next goals of our system development. Second, the translated code uses two additional communications not present in the MPI version. Closer analysis revealed that performance could be improved through partitioning methods on different iteration spaces of nested parallel loops that consider data affinity between processes. Furthermore, more accurate analysis of indirect array accesses – including the use of runtime methods – could improve on the conservative handling of array sections.

We identified compiler transformation and runtime techniques based on the knowledge above. When we applied them manually, the translated version achieved almost identical performance of the MPI version. The automation of those techniques is beyond the scope of this paper.

4 Related Work

An approach to extending the ease of shared memory programming to distributed memory platforms, such as clusters, is the use of a Software Distributed

Shared Memory (SDSM) system. SDSM is a runtime system that provides a shared address space abstraction on distributed memory architectures. Researchers have proposed numerous optimization techniques to reduce remote memory access latency on SDSM. Many of these optimization techniques aim to perform pro-active data movement by analyzing data access patterns either at compile-time or at runtime. In compile-time methods, a compiler performs reference analysis on source programs and generates information in the form of directives or prefetch instructions that invoke pro-active data movement at runtime [15]. The challenges for compile-time data reference analysis are the lack of runtime information (such as the program input data) or complex access patterns (such as non-affine expressions). By contrast, runtime-only methods predict remote memory accesses to prefetch data based on recent memory access behavior [16]. These methods learn the communication pattern in *all* program sections and thus incur overheads even in those sections that a compiler could recognize as not being beneficial. The idea of combined compile-time/runtime solutions has been studied [8, 17–19]. However, all these page-based SDSM approaches incur inherent overheads when detecting shared data accesses using a page-fault mechanism. Another drawback of this model is substantial false-sharing due to the page granularity at which SDSMs operate. Our system does not rely on any expensive operating system support, such as a page-fault mechanism; it generates MPI messages to communicate any size of array sections, which avoids false-sharing.

Another important contribution towards a simpler programming model for distributed memory machines was the development of High Performance Fortran (HPF) [1]. There are significant differences between our OpenMP-to-MPI translation approach and that of HPF. Even though HPF, like OpenMP, provides directives to specify parallel loops, HPF's focus is on the data distribution directives and automatic parallelization guided by those directives. Data partitioning is explicitly given by these directives and computation partitioning is guided by them [20]. Data typically has a single owner, and computation is performed by the owner of written data, i.e., the *owner computes* rule. In contrast to HPF, OpenMP has no user-defined data distribution input. Thus, computation partitioning is not derived from data distribution information. Instead, computation is distributed among processors based on OpenMP directives. Therefore, unlike most HPF implementations, our execution model has no concept of the *owner computes* rule.

PGAS (Partitioned Global Address Space) languages, such as UPC [3], Co-array Fortran [4], Titanium [21], and X10 [22], are other programming paradigms that have been proposed to ease programming effort by providing a global address space that is logically partitioned between threads. The programmer needs to specify the affinity between threads and data. In our work, the programmer writes a standard OpenMP shared memory program and our system translates this program into message-passing form for distributed memory platforms.

5 Conclusions and Future Work

The vision of a programming system that offers a shared address space to the user and executes applications efficiently on a distributed architecture has been elusive so far. High-Performance Fortran and Software Distributed-Shared-Memory systems have never received wide-spread acceptance. UPC and Co-array Fortran are current systems that involve the user in decisions about data placement on the distributed architecture. By contrast, the presented approach shows that unmodified OpenMP applications can be translated to execute on a cluster platform at performance levels close to hand-coded MPI. Our system is the first automatic translator, and supporting runtime system, that proves the value of automatic translation of shared memory OpenMP programs to execute on distributed memory machines. The results indicate that a high-productivity and high-performance programming interface for modern distributed machines is possible. As with all work on compilers and tools to support high performance parallel programming, additional research will yield better results on a wider range of programs.

In ongoing research, we aim to increase the scope of applications that the system can handle. As well, immediate benefits can come from improved recognition of collective operations at both compile time and during the program execution, and from exploiting data affinity and advanced work partitioning schemes. An advanced hybrid compiler/runtime technique for improving the accuracy of array sections is also under development.

References

1. High Performance Fortran Forum: High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Houston, Tex. (1993)
2. Amza, C., Cox, A.L., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., Yu, W., Zwaenepoel, W.: TreadMarks: Shared Memory Computing on Networks of Workstations. IEEE Computer 29(2), 18–28 (1996)
3. UPC Consortium: UPC Language Specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Laboratory (2005)
4. Numrich, R.W., Reid, J.: Co-array Fortran for Parallel Programming. SIGPLAN Fortran Forum 17(2), 1–31 (1998)
5. Basumallik, A., Eigenmann, R.: Towards Automatic Translation of OpenMP to MPI. In: ICS 2005: Proceedings of the 19th Annual International Conference on Supercomputing, pp. 189–198. ACM Press, New York (2005)
6. Bae, H., Bachega, L., Dave, C., Lee, S., Lee, S., Min, S., Eigenmann, R., Midkiff, S.: Cetus: A Source-to-Source Compiler Infrastructure for Multicores. In: Proc. of the 14th International Workshop on Compilers for Parallel Computing, CPC 2009 (January 2009)
7. Min, S., Basumallik, A., Eigenmann, R.: Optimizing OpenMP programs on Software Distributed Shared Memory Systems. International Journal of Parallel Programming 31(3), 225–249 (2003)

8. Min, S., Eigenmann, R.: Combined Compile-time and Runtime-driven, Pro-active Data Movement in Software DSM Systems. In: LCR 2004: Proceedings of the 7th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems, pp. 1–6. ACM Press, New York (2004)
9. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Simon, H.D., Venkatakrishnan, V., Weeratunga, S.K.: The NAS Parallel Benchmarks (1991)
10. Satoh, S.: NAS Parallel Benchmarks 2.3 OpenMP C version (2000), http://www.hpcs.cs.tsukuba.ac.jp/omni-openmp
11. Andrews, G.R.: Foundations of Parallel and Distributed Programming. Addison-Wesley Longman Publishing Co., Inc., Boston (1999)
12. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: Point-to-Point Communication. In: MPI: The Complete Reference, vol. 1, pp. 56–65. MIT Press (1998)
13. Saad, Y.: SPARSKIT: A Basic Tool Kit for Sparse Matrix Computations. Technical report, Computer Science Department, University of Minnesota, Minneapolis, MN 55455, Version 2 (June 1994)
14. Bhardwaj, D.: Description for Implementation of MPI Programs, http://www.cse.iitd.ernet.in/~dheerajb/MPI/Document/tp.html
15. Dwarkadas, S., Cox, A.L., Zwaenepoel, W.: An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System. In: Proc. of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems, ASPLOS VII, pp. 186–197 (1996)
16. Bianchini, R., Pinto, R., Amorim, C.L.: Data prefetching for software DSMs. In: The 12th International Conference on Supercomputing, pp. 385–392 (1998)
17. Viswanathan, G., Larus, J.R.: Compiler-directed Shared-memory Communication for Iterative Parallel Applications. In: Supercomputing (November 1996)
18. Keleher, P., Tseng, C.W.: Enhancing Software DSMs for Compiler-Parallelized Applications. In: Proc. of the 11th Int'l Parallel Processing Symp., IPPS 1997 (1997)
19. Keleher, P.: Update Protocols and Iterative Scientific Applications. In: Proceedings of the First Merged Symposium IPPS/SPDP, IPDPS 1998 (1998)
20. Gupta, M., Midkiff, S., Schonberg, E., Seshadri, V., Shields, D., Wang, K.Y., Ching, W.M., Ngo, T.: An HPF compiler for the IBM SP2. In: Supercomputing 1995: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing, CDROM, p. 71. ACM, New York (1995)
21. Yelick, K., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., Aiken, A.: Titanium: A High-Performance Java Dialect, pp. 10–11. ACM (1998)
22. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: An Object-oriented Approach to Non-uniform Cluster Computing. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, pp. 519–538. ACM, New York (2005)

A Methodology for Fine-Grained Parallelism in JavaScript Applications

Jeffrey Fifield and Dirk Grunwald

Department of Computer Science
University of Colorado
Boulder, Colorado, USA
{jeffrey.fifield,dirk.grunwald}@colorado.edu

Abstract. JavaScript has long been the dominant language for client-side web development. The size and complexity of client-side JavaScript programs continues to grow and now includes applications such as games, office suites, and image editing tools traditionally developed using high performance languages. More recently, developers have been expanding the use of JavaScript with standards and implementations for server-side JavaScript. These trends are driving a need for high performance JavaScript implementations. While the performance of JavaScript implementations is improving, support for creating parallel applications that can take advantage of now ubiquitous parallel hardware remains primitive. Pipeline, data, and task parallelism are ways of breaking a program into multiple units of work that can be executed concurrently by parallel hardware. These concepts are made explicit in the stream processing model of parallelization. Using the streaming model, an algorithm is divided into a set of small independent tasks called kernels that are linked together using first-in first-out data channels. The advantage of this approach is that it allows a compiler to effectively map computations to a variety of hardware while freeing programmers from the burden of synchronizing tasks or orchestrating communication between them. In this paper we describe Sluice, a library based method for the specification of streaming constructs in JavaScript applications. While the use of such a library makes concurrency explicit, it does not easily result in parallel execution. We show, however, that by taking advantage of the streaming model, we can dynamically re-compile Sluice programs to target a high performance, multi-threaded stream processing runtime layer. The stream processing layer executes computations in a different process and the offloaded tasks communicate with the original program using fast shared memory buffers. We show that this methodology can result in significant performance improvements for compute intensive workloads.

1 Introduction

Since its introduction by Netscape in the 1990's, JavaScript – also known as ECMAScript – has been the dominant language for client-side web development. The size and complexity of client-side JavaScript programs has grown considerably since that time, and now includes applications such as games, office suites, and image editing tools traditionally developed using high performance statically compiled languages. More recently, developers have been expanding the use of JavaScript with standards and implementations for server-side JavaScript such as CommonJS [1] and node.js [2]. These trends are driving a need for high performance JavaScript implementations.

While the performance of JavaScript implementations is improving, support for creating parallel applications that can take advantage of now ubiquitous parallel hardware remains primitive. We are aware of only one proposal, Web Workers [3], which addresses the problem of writing parallel JavaScript software. Web Workers is a scheme designed for browser based JavaScript where waiting for expensive computation to complete comes at the price of decreased responsiveness in the rest of the user application. It allows a developer to spawn a JavaScript module as a new thread isolated from both the spawning code and the browser's Document Object Model (DOM). The worker communicates with the original code via message passing.

While the thread or process level parallelism available using a system like Web Workers is quite useful, there are many applications that benefit from loop or procedure level parallelism like that provided by OpenMP or Thread Building Blocks [4] [5]. In this paper we refer to such parallelism as *fine-grained* parallelism, in contrast to the *coarse-grained* parallelism provided by thread or process level mechanisms.

Pipeline, data, and task parallelism are all ways of breaking a program into fine-grained units of work that can be executed concurrently by parallel hardware. These concepts are made explicit in the stream processing model of parallelization. Using the streaming model, an algorithm is divided into a set of small independent tasks that are linked together using first-in first-out data channels. The advantage of this approach is that it allows a compiler to effectively map computations to a variety of hardware while freeing programmers from the burden of synchronizing tasks or orchestrating communication between them.

The most popular streaming language in the research community is currently StreamIt [6], a language based on the ideas of synchronous data-flow [7]. Figure 1 gives a high level view of some of the constructs available in StreamIt. Figure 1(a), shows a four stage pipeline. Each kernel receives input from the one above it, does some work, then passes its result to the next kernel. Figure 1(b) illustrates the split-join construct. In a split-join, an input stream is divided or duplicated between a number of children kernels and the result of those kernels is combined into a single output stream. Figure 1(c) makes explicit the idea that when StreamIt kernels are side effect free, multiple copies of the same kernel can execute on different parts of the input stream. In contrast to the split-join pattern, which is specified by the programmer, data parallelism is handled automatically by the StreamIt compiler.

In the next section, we describe Sluice, a library based method for the specification of streaming constructs in JavaScript applications. Sluice takes the basic constructs found in the StreamIt language and makes them available to JavaScript developers. While the use of such a library makes concurrency

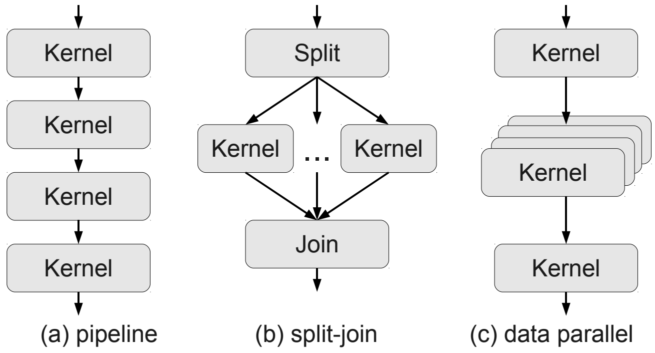


Fig. 1. Basic streaming constructs found in StreamIt and Sluice

explicit, it does not easily result in parallel execution because of the dynamic nature of JavaScript.

We show in Section 3, however, that by taking advantage of the characteristics of streaming computation, we can dynamically re-compile Sluice kernels to target a high performance stream processing runtime layer. We also show how, despite running these specialized Sluice kernels in another processes, we can maintain fast communication of streaming data and program state through the use of fast shared memory buffers. Finally, in Section 4, we evaluate our implementation, and demonstrate that our methodology can result in significant performance improvements for compute intensive workloads.

2 The Sluice Library

In this paper, we introduce a JavaScript library called *Sluice* that provides several stream processing abstractions inspired by those found in the StreamIt language. Using the Sluice library, streaming computations consist of small procedures called *kernels* communicating with one another over single input, single output data channels called *streams*. Kernels operate on streams using the `push` and `pop` operations. As in StreamIt, Sluice programs are constructed using a combination of pipeline and split-join programming patterns.

Stream program kernels in Sluice are simply JavaScript objects containing a method called `work`. This work function is called to do work on behalf of the kernel when it is scheduled to run. Once a kernel is executing as part of a streaming computation, its work function is called repeatedly by the Sluice scheduling algorithm until the kernel finishes executing. A kernel finishes executing when it becomes blocked on an empty input stream whose source has finished executing or when its work function returns `true`. This is the only requirement of a kernel work function – that it return a value of `true` if it has finished executing or return a value of `false` if it can keep processing input data if more is available. Two example Sluice kernels are shown in Figure 2.

```
function Counter(cnt) {
    this.i = 0;
    this.cnt = cnt;
    this.work = function() {
        if (this.i < this.cnt) {
            this.push(this.i++);
            return false;
        }
        return true;
    };
}
```

```
function Adder(arg) {
    this.a = arg;
    this.work = function() {
        var e = this.pop();
        e = e + this.a;
        this.push(e);
        return false;
    };
}
```

Fig. 2. Two simple Sluice kernels. The kernel on the left implements a simple counter. The kernel on the right adds the same value to each item popped the input stream and pushes the result to its output stream.

The Sluice library is responsible for implementing stream communication and for scheduling execution of the kernel work functions. Sluice schedules work functions in a single thread using a simple round robin algorithm. When an executing kernel is scheduled to run, its work function is called repeatedly until it returns `true` (indicating that it has finished), or until it can't proceed because it lacks input data. In the latter case, program control is yielded back to the scheduler, which then picks another kernel to run. If the kernel has finished, it is simply removed from the scheduler.

When a work function is called by the Sluice scheduler, the `push` and `pop` functions are available in the kernel's `this` scope. These operations have the same semantics as the JavaScript `Array.push` and `Array.pop` functions and by default Sluice streams are simply mapped onto `Arrays`. Unlike arrays, streams are conceptually infinite in size. The fact that they are necessary implemented using finite buffers is hidden by the `push` and `pop` operations. When a blocking condition is encountered during a `push` or `pop` operation, execution is yielded back to the Sluice scheduler. Once the blocking condition passes, execution can resume where it left off.

As in StreamIt, Sluice algorithms are constructed using two patterns: pipelines and split-joins. A pipeline is simply a series of kernels connected together in a producer consumer relationship. A split-join takes an input stream, distributes it to some number of kernels, then recombines the results into a single output stream. Pipelines and split-joins can be constructed from other pipelines and split-joins as well as from single kernels. The following code shows how we can create and run a three stage pipeline using the two kernels from Figure 2 and an additional kernel that simply prints everything in its input stream:

```
var src = new Counter(10);
var add = new Adder(1);
var snk = new Printer();
var p = Sluice.Pipeline(src,add,snk);
p.run();
```

The result of running this code is to print the numbers 1 through 10. We can obtain the same result in a more complicated way using a split-join:

```
var add0 = new Adder(1);
var add1 = new Adder(1);
var add2 = new Adder(1);
var sj = Sluice.SplitRR(1, add0, add1, add2).JoinRR(1);
var p = Sluice.Pipeline(new Counter(10), sj, new Printer());
p.run();
```

In this example, the split-join distributes one element at a time, in a round-robin fashion, to each of the three `Adder` kernels. The results of the `Adder` kernels are then combined in the same round-robin manner. The entire split-join is then used as a stage in a three stage pipeline.

The `run` method is used to execute the top level pipeline or split-join in a Sluice computation. This is an asynchronous method. That is, it does not wait for the Sluice computation to complete before returning. Instead, it optionally takes as an argument a callback function that will be called when the computation finishes. This style of asynchronous programming is very common in JavaScript.

As long as a Sluice kernel contains a `work` method that returns `true` or `false`, the rest of the kernel and kernel work function can contain arbitrary JavaScript code. However, as we will show in the next sections, it is advantageous for us follow a few simple coding conventions when creating Sluice kernels. First, we prefer that program state encapsulated in kernel objects be stored in the `this` scope of that kernel object. Second, we prefer that such state be initialized by the kernel object constructor. These conventions allow the implementation described in the next section to more easily obtain the program information it needs to perform optimization. A more mature implementation or one more tightly integrated into the JavaScript engine could relax these constraints.

3 Sluice Acceleration

A key ingredient for any effective program transformation or optimization is good program analysis. Software written using streaming patterns tends to have several characteristics that can greatly aid program analysis. This is also true of Sluice programs.

Using Sluice, program kernels are written in an object oriented style, encapsulating program state within program kernels. Because of this, we can more easily inspect and manipulate the state associated with a given program kernel. In addition, communication and synchronization within the streaming computation is limited to `push` and `pop` operations on streams. Coupled with well encapsulated state, this provides a great deal of flexibility in deciding where and when code should execute. Finally, once a streaming computation is started by a program, the kernels in that computation tend to execute for a long time. This mitigates the impact of any runtime compilation, analysis, and parallelization mechanisms which improve the performance of long running kernels.

In this section we discuss how we can take advantage of these characteristics to obtain higher performance and parallel execution. We describe how we compile Sluice kernels written in JavaScript to target a high performance stream processing runtime. We also describe how we provide efficient synchronization of program state and efficient communication of streaming data between the stream processing runtime and the JavaScript execution engine. The result is that we enable specialized code generation and fine-grained parallelization of a certain class of JavaScript code.

3.1 Code Generation

Specialized code generation for accelerating Sluice kernels takes place in two steps. First we perform translation from the JavaScript source code to a low level streaming abstraction, the Stream and Kernel Intermediate Representation (SKIR). SKIR is an instruction level representation built on top of the Low Level Virtual Machine (LLVM) instruction set [8]. It includes all of the LLVM instructions plus several additional instructions used to create and execute streaming computations. The second step of code generation is to compile the SKIR representation using a just-in-time compiler provided by a SKIR runtime layer. A detailed description of this process is outside the scope of this paper, but it includes transformations and code generation specialized for the streaming model. The result of the two step code generation process is optimized machine code.

Because JavaScript applications are distributed as source code, and because the source code for a particular function is available to the program itself, we can implement runtime code translation as part of the Sluice user library. For the purposes of this paper, a kernel to be accelerated is identified to the library by the programmer. This is equivalent to using program annotations or pragmas in a statically compiled language. Instances of kernels are translated after they are allocated (using the `new` operation), but before they are added to a streaming computation using the pipeline or split-join constructs. The following code example shows how a Sluice kernel can be translated to SKIR using our current implementation:

```
var k = new MyKernel(...);
sluice.toSkir(k,
              function(err, ret) {
                sluice.Pipeline(..., ret, ...).run();
              });
```

In this code, the kernel `k` is translated to SKIR using the Sluice library's `toSkir` method. The result is added to a pipeline which is immediately executed by calling `run`.

Once a kernel has been identified for optimization, its source code is parsed and an abstract syntax tree (AST) for its work function is constructed. This AST is then used to generate SKIR code corresponding to the original Sluice kernel. The main problem we face in this process is the same one faced by any code generation system for JavaScript - mapping dynamic types to static types. The issue is that the type of a particular variable cannot, in general, be statically determined by a compiler whereas the machine code generated by a JavaScript compiler or the SKIR code generated by the Sluice compiler require static types.

All high performance JavaScript implementations obtain that performance in part by making assumptions about the types contained in a piece of code. In trace-based compilers, type information is recorded during trace collection. For types that remain static during trace collection, machine code specialized to that type can be emitted [9]. In compilers employing direct translation from JavaScript to machine code, such as the V8 compiler [10], code is emitted for the first type seen, then patched if the type changes. Both approaches are based on the observation that although JavaScript supports dynamic types, in practice types remain fairly stable at runtime. Both approaches must also insert checks into the generated machine code to make sure assumptions about types are not violated.

For Sluice kernels written using the coding conventions discussed in Section 2, most types will be known before kernel execution. This is the direct result of the use of encapsulated program state and the use of the stream communication abstraction. The use of encapsulated program state means that any program state used by the kernel is already initialized when a kernel object is passed to Sluice for translation to SKIR. This in turn means that the Sluice compiler can simply access the data to determine its type. The type must be rechecked each time the kernel is executed. However, because the kernel work function is typically called many times during kernel execution, the cost of these type checks is very small when compared to the inline checks generated by a typical JavaScript compiler.

The stream operations `pop` and `push` are trivially translated to SKIR as they are supported directly by the representation. Stream communication is also the only source of data from outside of the kernel during execution. By definition we know that the streams contain only a single type. Thus we know that interaction with stream objects will not result in dynamic type behavior.

A remaining source of dynamic types is from objects created during execution of the kernel work function. For example, different types could be assigned to the same variable depending on the path taken through an if-then-else statement. The Sluice to SKIR compiler does not currently handle this behavior, however this is a limitation of our current implementation, not of our approach. Existing mechanisms such as those used in the V8 compiler could be used.

Once static type information is computed for a Sluice kernel, the system translates the AST form of the kernel work function to SKIR code. Instead of generating low level SKIR code (i.e. LLVM code) directly, Sluice generates C code which includes compiler intrinsics corresponding to SKIR operations. Because JavaScript is syntactically similar to C and because good C to LLVM compilers already exist, this design decision simplifies our implementation enormously. The C code is translated to SKIR code using a compiler that understands the SKIR intrinsics. A side benefit of this approach is that much of the compilation process automatically runs in parallel with the Sluice program.

3.2 Offloading Kernel Execution

In addition to providing a just-in-time compiler, the SKIR runtime provides a high performance multi-threaded scheduler. In this section we describe how the optimized Sluice kernel generated by the SKIR compiler can be executed using the SKIR scheduler. This scheduler provides dynamic scheduling of SKIR kernels and is built on task-based work stealing. It executes in a separate process from the original JavaScript program.

Because the SKIR scheduler executes as a separate process, our main challenge is in obtaining high performance communication between offloaded Sluice kernels and the rest of the JavaScript program. Two things must be communicated efficiently between the separate parts of the program; these are any program state used by offloaded kernels and any streaming communication between Sluice kernels and SKIR kernels.

Stream communication between processes is made efficient by mapping the streams onto lock free queues in shared memory. In SKIR, this communication is very fast as the result of specialized code generation which inlines the required instructions into the program kernel. For Sluice, this functionality is provided using stream objects implemented in C++ but made accessible through a JavaScript module. In both cases, the runtime remapping of the stream implementation is made possible because the program is written using the stream abstractions provided by Sluice or SKIR and does not assume any implementation details.

It is more difficult to efficiently communicate non-stream data used by a Sluice kernel from JavaScript to SKIR. Program state lacks the features that make implementing streaming communication fairly easy. It is not made structured or abstract by the programming model itself. Instead, it's structure depends on the algorithms used by the programmer and can vary greatly from program to program.

We can, however, take advantage of the same Sluice programming conventions that make translation feasible in the first place. Because program state is contained in the `this` scope, the system can easily look up the data. That is, it can determine the location, type and contents of a kernel's state when its work function is called. We also assume that because the Sluice or SKIR runtime owns the kernel object during its execution, another part of the JavaScript program will not attempt to alter the state of a running kernel.

Because it is intractable to figure out what parts of a kernel's state are read or written during an execution of its work function, all state must be copied from Sluice to SKIR when a kernel is called. As long as the kernel is executing, this data remains in the SKIR runtime. When the kernel finishes, the state is returned to Sluice.

Copying data between processes can incur high overheads compared to the granularity of computation the streaming model exposes. To reduce this overhead as much as possible, we again use shared memory between the processes. The first time the program state for a particular kernel is copied, the Sluice runtime requests a piece of shared memory from the SKIR runtime. The size of this memory is the same as the C language struct formed by combining all the state into a single buffer.

Each time a kernel is called to run in the SKIR layer, the kernel's state is copied from JavaScript into the shared memory buffer. The kernel executing in the SKIR runtime interprets the data as a C struct, and can directly address the data without an additional copy. When the kernel finishes execution, the shared memory can be translated back into its JavaScript version. Thus the data is copied twice – once as it is translated from JavaScript to the C struct at the start of execution, and once as it is translated back to JavaScript data at the end of execution.

4 Evaluation

In this section we evaluate the performance of our system using compute-intense JavaScript benchmarks that fit well into the streaming model. Four of the benchmarks are taken from the Pixastic library of image processing routines [11]. We also include a nbody physics simulation adapted from example code in the CUDA SDK [12]. This benchmark is similar to code that might be found in a game engine.

As written, the code found in the Pixastic library is already very close to the form required by Sluice. We must perform only a few modifications to turn the image processing routines into Sluice kernel objects. Because these routines operate at the granularity of an entire image, we identify parameters that only change between images, such as the image itself. These parameters are re-written as kernel state that is initialized when the kernel is constructed. The rest of the function body is placed within a kernel work function. Because the Pixastic functions operate on an image at a time, there is little need for stream communication in these benchmarks. Nevertheless, SKIR requires that a kernel has at least one input or output stream. We fulfill this requirement by passing the image width and height to the kernel using streams.

An example from Pixastic is shown in Figure 3. In this figure, the original Pixastic code as well as the Sluice version are shown for the `invert` routine. We run all of the Pixastic benchmarks on image data that is read from disk into memory before timing begins. All images are in RGBA format with dimensions of 2592x1944 (5 Megapixels).

The nbody physics benchmark is written as a three stage pipeline. Almost all the computation takes place in the middle stage, shown in Figure 4. Each execution of this stage's kernel work function computes the forces on single particle due to all other particles in the system. The computed forces are pushed to the output stream. Each execution of the entire three stage pipeline corresponds to a single iteration of the nbody simulation. That is, it computes the force and updates the positions and velocities (in the third stage) for all the particles in the system.

```
function invert_pixastic(params) {
  var data = Pixastic.prepareData(params);
  var invertAlpha = !!params.invertAlpha;
  var rect = params.options.rect;

  var p = rect.width * rect.height;
  var pix = p*4, pix1 = pix+1;
  var pix2 = pix+2, pix3 = pix+3;

  while (p--) {
    data[pix-=4] = 255 - data[pix];
    data[pix1-=4] = 255 - data[pix1];
    data[pix2-=4] = 255 - data[pix2];
    if (invertAlpha)
      data[pix3-=4] = 255 - data[pix3];
  }
  return true;
}
```

```
function invert_sluice(data, invert_alpha) {
  this.data = data;
  this.invertAlpha = invert_alpha;

  this.work = function () {
    var w = this.pop();
    var h = this.pop();
    var p = w * h;
    var pix = p*4, pix1 = pix + 1;
    var pix2 = pix + 2, pix3 = pix + 3;

    while (p--) {
      this.data[pix-=4] = 255-this.data[pix];
      this.data[pix1-=4] = 255-this.data[pix1];
      this.data[pix2-=4] = 255-this.data[pix2];
      if (this.invertAlpha)
        this.data[pix3-=4]=255-this.data[pix3];
    }
    this.push(w); this.push(h);
    return false;
  };
}
```

Fig. 3. Comparison of Pixastic code (left) with the same code ported to Sluice (right). The code that is shown inverts all the pixels in an image.

```
function CalculateForces(pos_rd, softeningSquared) {
    this.m_pos_rd = pos_rd;
    this.m_softeningSquared = softeningSquared;
    this.work = function () {
        var force = [0.0,0.0,0.0];
        var i = this.pop()*4;
        var N = this.pop()*4;
        for (var j=0; j<N; j+=4) {
            var r0, r1, r2;
            r0 = this.m_pos_rd[j+0] - this.m_pos_rd[i+0];
            r1 = this.m_pos_rd[j+1] - this.m_pos_rd[i+1];
            r2 = this.m_pos_rd[j+2] - this.m_pos_rd[i+2];
            var distSqr = (r0 * r0) + (r1 * r1) + (r2 * r2);
            distSqr += this.m_softeningSquared;
            var invDist = 1 / Math.sqrt(distSqr);
            var invDistCube =  invDist*invDist*invDist;
            var s = this.m_pos_rd[i+3] * invDistCube;
            force[0] += (r0 * s);
            force[1] += (r1 * s);
            force[2] += (r2 * s);
        }
        var f = i/4;
        this.push(f);
        this.push(force[0]);
        this.push(force[1]);
        this.push(force[2]);
        return false;
    };
}
```

Fig. 4. The CalculateForces kernel found in the nbody benchmark. Ported from a C++ version of the benchmark found in the NVidia CUDA Software Development Kit.

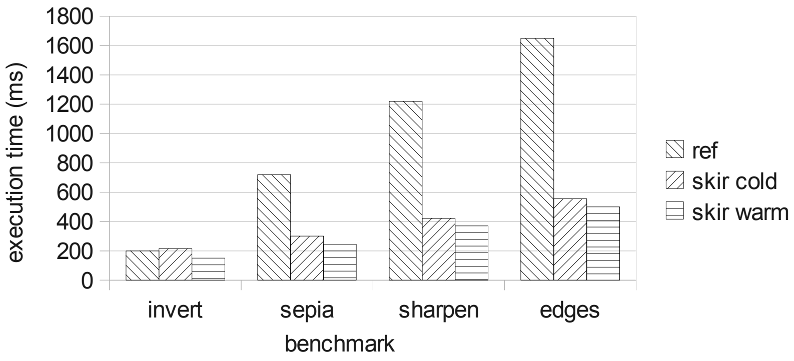


Fig. 5. Performance of Pixastic image processing routines coded as ordinary JavaScript compared to the performance of the same routines running as Sluice kernels using the SKIR runtime

We run our implementation of Sluice on node.js, a non-browser JavaScript environment build on top of the V8 JavaScript execution engine. Node is typically used in the development of network applications. The single threaded scheduling algorithm implemented by Sluice also makes use of node-fibers [13], a package providing fibers/co-routines for node.js. All experiments are run on a 4-core Intel i7-920 processor under Ubuntu 10.10.

4.1 Single Threaded Offload

We first evaluate the performance of our system when offloading a Sluice kernel to a single threaded SKIR runtime. Because the Sluice and SKIR layers run as separate processes, the offloaded kernel still runs in parallel with the rest of the Sluice program, but we do not attempt to further parallelize the offloaded kernel.

The results of this experiment for the Pixastic kernels can be seen in Figure 5. We measure three cases: "ref" shows the performance of the original image processing routine called as an ordinary JavaScript function whereas "skir cold" and "skir warm" show the performance when executing the image processing routine as a Sluice kernel using the SKIR runtime. The cold version is when the kernel has never been seen by the SKIR runtime while the warm version is when the SKIR runtime has already seen, processed, and cached the kernel. We see good performance improvements for all but the smallest kernel, `invert`. In this case, the overhead of acceleration is greater than the benefit. For the other benchmarks, we see significant performance improvement due to our specialized code generation.

The results of this experiment for the nbody benchmark can be seen in Figure 6. The Figure shows the average time per simulation iteration when using a procedural JavaScript version of the benchmark, the three stage pipeline Sluice version, and the Sluice version with the `CalculateForces` kernel offloaded to SKIR. The results are similar to those for the Pixastic benchmark, with significant performance improvements due to our specialized code generation in

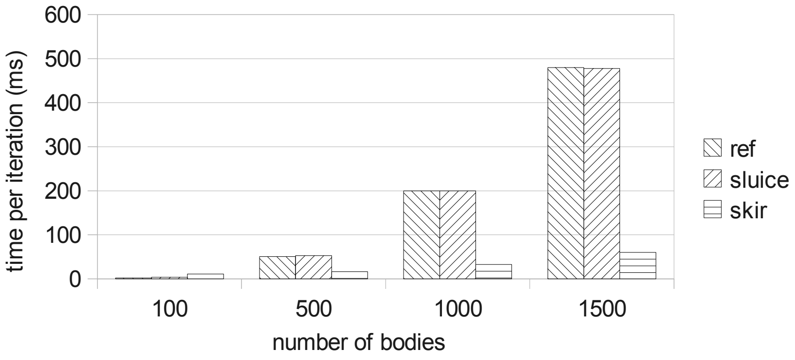


Fig. 6. Performance of the nbody benchmark implemented as procedural JavaScript, as Sluice code, and as Sluice code with CalculateForces kernel running on SKIR runtime

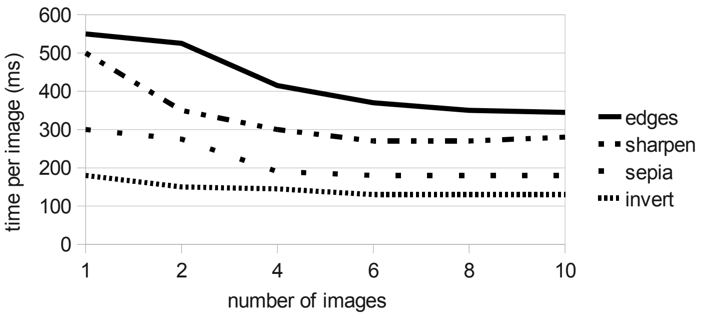


Fig. 7. Per image processing time for the Pixastic benchmarks implemented when a varying number of images are processed using Sluice task parallelism

most cases. For the smallest test, a system of only 100 particles, the overhead of acceleration overwhelms the benefit.

4.2 Parallel Kernel Execution

In this section we evaluate the potential for parallel execution created by our system.

Task Parallelism. Although we did not parallelize the Pixastic benchmarks, we can still execute multiple instances of a particular kernel in parallel. This is the simplest form of task parallelism exposed by Sluice programs. To test this, we created a Sluice program that sequentially (because JavaScript is single threaded) creates, compiles, and executes a varying number of image kernels using the SKIR runtime. We give the runtime 8 worker threads (equal to the number of hardware threads) to run the kernels, so up to 8 kernel instances

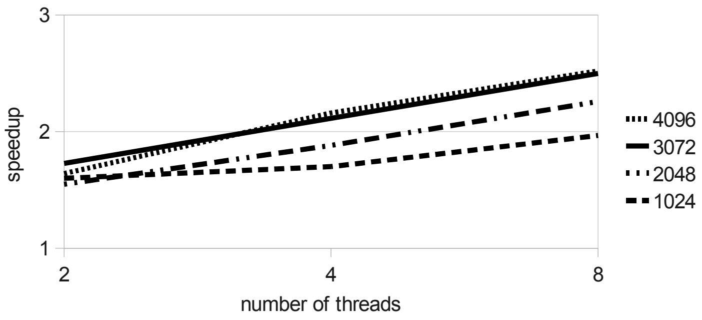


Fig. 8. Speedup of the nbody benchmark due to data parallelism when the CalculateForces kernel is run on a multi-threaded SKIR runtime compared to the same benchmark using a single-threaded SKIR runtime.

can run in parallel. This scenario is similar to what might be encountered in a compute intense server-side JavaScript application.

Figure 7 shows the results of running a varying number of Pixastic kernels on different images concurrently using Sluice task parallelism. It shows the mean time required to process a single image. We observe that as the number of executing kernel instances increases, the system starts to effectively mask much of the overhead associated with running Sluice kernels under SKIR. Eventually, however, the lines on the graph flatten, because the system cannot go faster than the sequential parts of the program (the individual kernels and the Sluice runtime). The larger `edge` detection and `sharpen` kernels show the best results as they contain the most computation to overlap with other work. Likewise, `invert` and `sepia` show the worst improvement, because their execution is dominated by sequential overhead.

Data Parallelism. Stream program kernels are often written so that they read, but do not modify, their internal state. When this is true – as it is for the `CalculateForces` kernel in the nbody benchmark – it may be profitable to run multiple copies of a single kernel instance on different portions of the input stream. This is how the stream programming model exposes data parallelism.

Figure 8 shows the results of executing the nbody benchmark with the `CalculateForces` kernel offloaded and with data parallelism enabled in the SKIR runtime. The figure shows results for 1024, 2048, 3072, and 4096 particles in the simulated system while using 2, 4, or 8 worker threads in the SKIR runtime. The benchmark runs several hundred iterations of the simulation and reports mean time per iteration. The speedup versus using a single threaded SKIR runtime is shown. All of the simulation sizes show performance improvement of 2-2.5x due to parallelism when utilizing the entire test machine. We point out that because JavaScript fully utilizes one of the cores in our four core test machine, and because this experiment measures scaling in the non-JavaScript portion of execution, we don't expect that the best case speedup is much greater than 3x.

5 Related Work

To date, there has been little work on providing programmers with tools to write parallel JavaScript code. Most efforts, such as TraceMonkey [9] and V8 [10], have gone into providing increased sequential performance.

Web Workers is an API for browser based JavaScript that allows a programmer to spawn JavaScript modules that operate in the background of a web application [3]. It provides process-like isolation as well as a basic message passing mechanism. Web Workers provide parallelism at the script or module level. In contrast, the system presented in this paper provides parallelism at a much finer granularity. It allows procedure level parallelism between program kernels as well as loop level parallelism within side effect free kernels. In addition, our system provides a mechanisms for high-performance communication and memory sharing.

There has been previous work on using the parallelism provided by modern hardware to speed JavaScript execution. Mehrara, et. al. use speculative parallelism to offload guard code to a secondary thread [14] as well as to perform speculative loop parallelization [15]. Their work has different goals from our own, as they provide increased sequential performance through automatic parallelization, while we provide programming tools for parallelism. A similar, but higher level approach is taken by Crom [16], a system that speculatively executes JavaScript in web browsers. Crom seeks to reduce latency in web applications by speculatively executing event handlers in a parallel thread that is a shadow copy of the browser context. Like our own system, it requires programmer participation.

6 Conclusion

This paper has proposed a methodology for introducing fine-grained parallelism into JavaScript applications. In particular, it has shown how to take advantage of the stream processing programming model to isolate program state and abstract inter-task communication in a way that allows parallelization within the single threaded, dynamically typed, JavaScript execution environment. This paper also introduced Sluice, a software library providing StreamIt-like constructs for JavaScript. While the approach presented is not applicable to all programming styles and patterns, we have shown it can result in significant performance improvements for well defined compute-intensive code. These performance improvements come from parallelization as well as from specialized code generation.

References

1. CommonJS, `http://www.commonjs.org`
2. node.js, `http://nodejs.org`
3. Web Workers, `http://www.w3.org/TR/workers`
4. OpenMP, `http://openmp.org`

5. Intel Thread Building Blocks (TBB), http://www.threadbuildingblocks.org
6. Gordon, M.I., Thies, W., Amarasinghe, S.: Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 151–162. ACM (2006)
7. Lee, E.A., Messerschmitt, D.G.: Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. IEEE Transactions on Computing 36(1) (1987)
8. Low Level Virtual Machine (LLVM), http://llvm.org
9. Gal, A., et al.: Trace-based Just-in-Time Type Specialization for Dynamic Languages. In: Proceedings of the 2009 Conference on Programming Language Design and Implementation, pp. 465–478 (2009)
10. Google V8 JavaScript Engine, http://code.google.com/p/v8
11. Pixastic, http://www.pixastic.com
12. Nvidia CUDA SDK, http://developer.nvidia.com
13. node-fibers, http://github.com/laverdet/node-fibers
14. Mehrara, M., Mahlke, S.: Dynamically Accelerating Client-side Web Applications through Decoupled Execution. In: Proceedings of the 2011 International Symposium on Code Generation and Optimization, CGO 2011, pp. 74–84 (2011)
15. Mehrara, M., Hsu, P.C., Samadi, M., Mahlke, S.: Dynamic Parallelization of JavaScript Applications Using an Ultra-lightweight Speculation Mechanism. In: Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture, HPCA 2011, pp. 87–98 (2011)
16. Mickens, J., Elson, J., Howell, J., Lorch, J.: Crom: Faster Web Browsing Using Speculative Execution. In: Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation, NSDI 2010 (2010)

Evaluation of Power Consumption at Execution of Multiple Automatically Parallelized and Power Controlled Media Applications on the RP2 Low-Power Multicore

Hiroki Mikami, Shumpei Kitaki, Masayoshi Mase, Akihiro Hayashi, Mamoru Shimaoka, Keiji Kimura, Masato Edahiro, and Hironori Kasahara

Dept. of Computer Science, Waseda University 3-4-1 Ohkubo, Shinjuku-ku, Tokyo, 169-8555, Japan
{hiroki,kitaki,mase,ahayashi,shimaoka,kasahara}@kasahara.cs.waseda.ac.jp, kimura@apal.cs.waseda.ac.jp, eda@etrl.jp

Abstract. This paper evaluates an automatic power reduction scheme of OSCAR automatic parallelizing compiler having power reduction control capability when multiple media applications parallelized by the OSCAR compiler are executed simultaneously on RP2, a 8-core multi-core processor developed by Renesas Electronics, Hitachi, and Waseda University. OSCAR compiler enables the hierarchical multigrain parallel processing and power reduction control using DVFS (Dynamic Voltage and Frequency Scaling), clock gating and power gating for each processor core using the OSCAR multi-platform API. The RP2 has eight SH4A processor cores, each of which has power control mechanisms such as DVFS, clock gating and power gating. First, multiple applications with relatively light computational load are executed simultaneously on the RP2. The average power consumption of power controlled eight AAC encoder programs, each of which was executed on one processor, was reduced by 47%, (to 1.01W), against one AAC encoder execution on one processor (from 1.89W) without power control. Second, when multiple intermediate computational load applications are executed, the power consumptions of an AAC encoder executed on four processors with the power reduction control was reduced by 57% (to 0.84W) against an AAC encoder execution on one processor (from 1.95W). Power consumptions of one MPEG2 decoder on four processors with power reduction control was reduced by 49% (to 1.01W) against one MPEG2 decoder execution on one processor (from 1.99W). Finally, when a combination of a high computational load application program and an intermediate computational load application program are executed simultaneously, the consumed power reduced by 21% by using twice number of cores for each application. This paper confirmed parallel processing and power reduction by OSCAR compiler are efficient for multiple application executions. In execution of multiple light computational load applications, power consumption increases only 12% for one application. Parallel processing being applied to intermediate computational load applications, power consumption of executing one application on one processor core (1.49W) is almost same power consumption of two applications on eight processor cores (1.46W).

1 Introduction

Multicore processors have been widely used in a variety of applications such as embedded systems like mobile devices, games, Digital TV, robots and automobiles, PCs, workstations, and high-performance computers. In embedded systems, various types of multicore processors, for example IBM Toshiba Sony CELL/BE[1], Hitachi Renesas Waseda RP1[2], RP2[3] and RPX[4], ARM NEC MPCore[5], Fujitsu FR1000[5], Panasonic UniPhier and so on, are used for a wide variety of applications such as image and audio processing, and real-time controls. In these embedded multicore platforms, the reduction of power consumption is a crucial problem to extend battery life. Currently, many multicore supports DVFS and/or Power Gating for each processor coare controled by OS. However, OS does not control power status inside an application program pallalelized for multiple cores. The OSCAR compiler has realized an automatic power control scheme using DVFS and Power Gating for each core on a multicore with automatic parallelization of an application program under the constraints of the minimum time execution or the satisfaction of real-time deadline, or real-time execution. However, no paper has evaluated power consumed by multiple application programs parallelized and power controlled by a compiler. This paper evaluates performance and consumed power on a 8-core homegeneous multicore RP2 integrating eight 600MHz SH4A processor cores with power control function of 100%, 50%, 25%, 0% of Frequency statuses in a one clock transition time, 1.4V, 1.2V and 1.0V three levels of voltage states and power gating for individual cores in 5 micro seconds power shut-down and 30 micro seconds power recovery when multiple media applications parallelized and power-controlled by OSCAR comoiler are executed simultaneously. Also, the parallel and power controlled C programs are generated using OSCAR multi-platform API, which are a set of about 20 directives for C and Fortran programs using 4 directives from OpenMP, such as Section, Flush, Critical and Thread-private and new directives for power control, realtime management, DMA transfer, distributed shared memory management, group barrier synchronization and so on. The generated C or Fortran parallel program using OSCAR API[6] can be compiled by ordinary OpenMP compilers. The rest of this paper is organized as follows. Section 2 provides an overview of the OSCAR compiler and its power reduction scheme. Section 3 describes RP2 low-power multicore and characteristic of evaluated applications. Section 4 shows the power consumption evaluation of OSCAR compiler power reduction scheme on RP2. Finally, Section 5 summarizes the main conclusion of this paper.

2 Multigrain Parallel Processing

The OSCAR compiler exploits multigrain parallelism, coarse grain parallelism, loop level parallelism and near fine grain parallelism from the whole source program. The OSCAR compiler consists of the Fortran77 and restricted C frontend, middle path for multigrain parallelization and several backbends for different target machines. The compiler generates coarse grain tasks called macro-tasks, analyzes parallelism among the macro-tasks by the earliest executable condition analysis,schedules macro-tasks to threads or thread groups statically, apply power reduction scheme, generates parallel code with OSCAR API.

2.1 Macro-Task Generation

In multigrain parallelization, a program is decomposed into three kinds of coarse grain tasks, or macrotasks (MTs), such as a block of pseudo assignment statements (BPA) like a basic block, a repetition block (RB) like a loop and a subroutine block (SB) like a subroutine [7–11]. Macro-tasks can be hierarchically defined inside each sequential loop which cant be parallelized, and a subroutine block.

2.2 Earliest Executable Condition

After generation of macro-tasks, data dependencies and control flow among macro-tasks are analyzed in each nested layer, and hierarchical macro-flow graphs (MFGs) as shown in Figure 1 (a) representing control flow and data dependencies among macro-tasks are generated[7, 8]. Next, to extract coarse grain task parallelism among macro-tasks, Earliest Executable Condition analysis[7, 8, 12, 13] is applied to each macro-flow graph. It analyzes control dependencies and data dependencies among macro-tasks simultaneously and determines the conditions on which macro-tasks may begin their execution earliest. By this analysis, a macro-task graph (MTG)[7, 8, 12] as shown in Figure 1 (b) is generated for each macro-flow graph. This graph represents coarse grain task parallelism among macro-tasks.

2.3 Macro-Task Scheduling

Static scheduling or dynamic scheduling is chosen for each macro-task graph. If a macro-task graph has only data dependencies and is deterministic, static scheduling at compilation time is selected. Generally, static scheduling is more effective than dynamic scheduling since it can minimize data transfer and synchronization overhead without runtime scheduling overhead. If a macro-task graph is non-deterministic by conditional branches among coarse grain tasks, dynamic scheduling at runtime is selected to handle the runtime uncertainties. Dynamic scheduling routines for non-deterministic macro-task graphs are generated by OSCAR compiler and inserted into a parallelized program code to minimize runtime scheduling overhead.

2.4 Power Reduction Scheme[14]

The power reduction scheme determines suitable voltage and frequency for each MT after Macro-task scheduling. Figure 2 (a) shows MTs 1, 2 and 5 are assigned to PE0, MTs 3 and 6 are assigned to PE1, MTs 4, 7 and 8 are assigned to

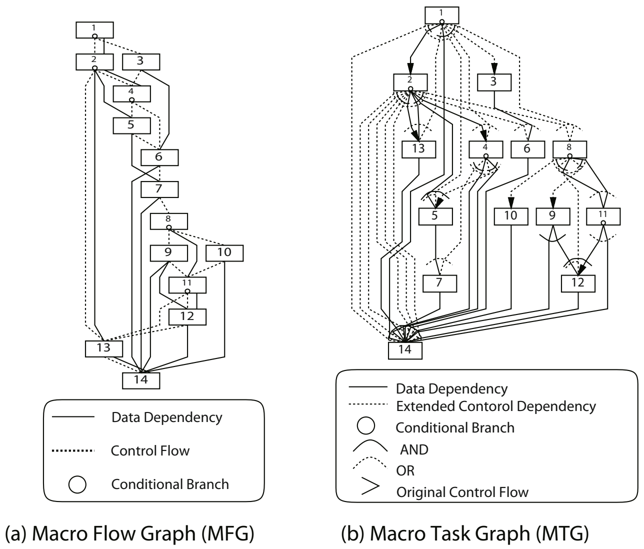


Fig. 1. Earliest Executable Condition Analysis

PE2. Edges among tasks show data dependence. OSCAR compiler estimates execution time of each MTs, then decide critical path which is longest execution time of the MTG. Defining execution time of the target MTG, parallel processing of the MTG after DVFS has to satisfy the given deadline. OSCAR compiler with the power reduction scheme decides optimal frequency and voltage of each MT to minimize the whole energy consumption. The detail of voltage and frequency scaling algorithm is described in [14]. After determining voltage and frequency of MTs, OSCAR compiler with the power reduction scheme tries to apply dynamic power shutdown, clock gating or frequency scaling to reduce unnecessary energy consumption including static power consumption by idle processors. OSCAR compiler recalculates MTGs after DVFS like Figure 2 (b) and selects power gating, clock gating, frequency scaling or no control for each idle part, considering the period of idle time and its overhead.

2.5 OSCAR API Code Generation[6]

The OSCAR API is designed on a subset of OpenMP for preserving portability over a wide range of multicore architectures. An OpenMP-based design can support both C and Fortran programs. However, in order to avoid the complexity of a backend compiler and runtime routines, only three directives are chosen from the OpenMP, such as parallel sections, flush, and critical, which enable one-time single level thread creation. Note that nested parallelism is not required for the OSCAR API. In addition to these three directives, one OpenMP directive (threadprivate) is extended, and 12 directives are newly added to

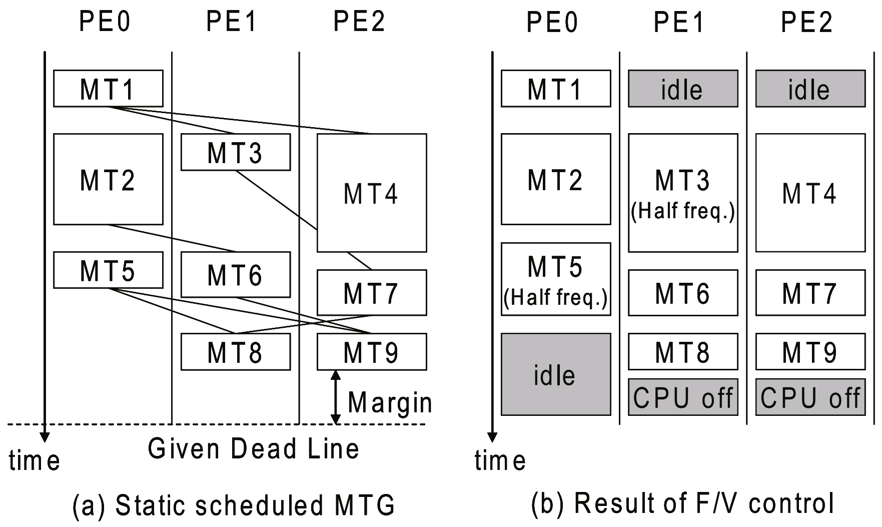


Fig. 2. OSCAR compiler's power control scheme

support the parallel optimizations carried out using the OSCAR compiler, whose specifications are simple as possible.

3 RP2 Multicore and Characteristic of Applications

This chapter describes RP2, 8-core multicore processor developed by Renesas Electronics, Hitachi and Waseda University and characteristic of evaluated applications.

3.1 RP2 Multicore

RP2[3] is 8 cores multicore processor developed by Renesas Electronics / Hitachi / Waseda University supported by NEDO Multi core processors for realtime consumer electronics project. RP2 integrate eight SH-4A cores. Figure 3 shows the architecture of RP2. Each processor core has CPU, cache memory, local memory (ILRAM, OLRAM), distributed shared memory (URAM), and DMAC (DTU). RP2 guarantee hardware cache coherence by MESI protocol until 4 cores. However, software must guarantee cache coherence 5 cores and above. Frequency of each core can be changed to 600MHz, 300MHz, 150MHz, and 75MHz independently. In addition supply voltage of entire core can be changed to 1.40V, 1.20V, and 1.00V. Figure 4 shows RP2 power status. Light Sleep stop CPU clock supply. Normal Sleep stop clock supply of processor core except URAM and DMAC. Resume Standby stop URAM clock supply and shutdown processor core except URAM. CPU off shutdown entire core.

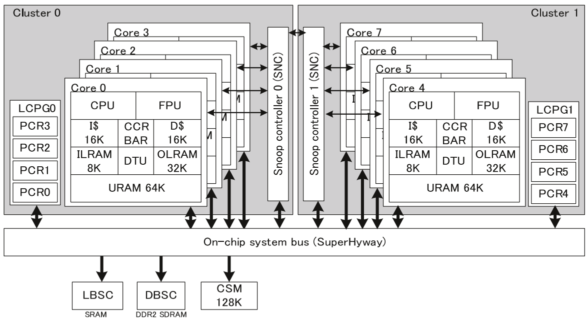


Fig. 3. Architecture of RP2 Multicore

Status	Clock Gating	Power Shutdown	Power Consumption [W]
FULL (600MHz,1.40V)	-	-	5.99
MID (300MHz, 1.20V)	-	-	2.61
LOW (150MHz,1.00V)	-	-	1.27
VERYLOW (75MHz, 1.00V)	-	-	1.00
Normal Sleep	CPU, cache, ILRAM, OLRAM	-	0.725
Resume Standby	URAM	CPU, cache, ILRAM, OLRAM, DTU	0.563
CPU off	-	CPU, cache, ILRAM, OLRAM, DTU, URAM	0.554

Fig. 4. Power Status of RP2 Multicore

3.2 Evaluated Applications

This section describes specifications of evaluated applications.

AAC Encoder. This program read audio data and process Filter bank, MS stereo, Quantization, and Haffman coding for each frame. Each frame can be processed parallel. Encoding process are unrolled by number of processor. Deadline of power control is 23ms per one frame. This AAC encoder is implemented by Parallelizable C, which is referred to AAC-LC encoding program of Renesas Electronics and Hitachi.

MPEG2 Decoder. MPEG2 decoder stages are Variable Length Decoding (VLD), Motion Compensation, Inverse Quantization and Inverse DCT. MPEG2 decoder has slice parallelism and macroblock parallelism. Each parallelism has sequential execution on VLD. In this paper, the code of MPEG2 decoder is implemented with reference of MediaBench[15] with the description explained in Section5.1. Furthermore, VLD for a slice is divided into searching a slice header, called Prescanning[16], and decoding a slice. Decoding slices is executed in parallel, so that parallel execution part is increased. The OSCAR compiler extracts slice level parallelism. Deadline of power control is 33ms per one frame.

Characteristics of Application. Figure 5 shows characteristics of each application. As a light computational load application, AAC encoder is selected. AAC encoder can fulfill deadline by VERYLOW (75MHz) power status. 19 seconds audio data is inputted. AAC encoder process 19 seconds audio data by 2.7 seconds. AAC encoder has enough waiting time for deadline. As a middle computational load application, AAC encoder (deadline 3 seconds) and MPEG2 decoder (resolution 352x128) is selected. AAC encoder need 2.7 seconds by one core, so this AAC encoder must run at FULL power status when using only one core. MPEG2 decoder process 352 pixel x 128 pixel video by 8.3 seconds by using one core. In addition continuous I/O has been issued by bit processing of Prescanning. As a high computational load application, MPEG2 decoder (resolution 352x240) is selected. MPEG2 decoder process 352 pixel x 240 pixel video by 17.6 seconds by using one core and 11.1 seconds by using two cores. MPEG2 decoder must use two cores or above to fulfill deadline (15 seconds). This paper execute these applications multiple and mesure power consumption of entire chip.

Load	Application	Characteristic
Low	AAC encoder (deadline 19 seconds)	・enough waiting time to deadline ・computational load is relatively light
Intermediate	AAC encoder (deadline 3 seconds)	・no waiting time to deadline ・computational load is relatively light
Intermediate	MPEG2 decoder (resolution 352x128)	・no waiting time to deadline ・computational load is relatively high ・frequent I/O access
High	MPEG2 decoder (resolution 352x240)	・parallel processing is need to meat deadline ・computational load is relatively light ・frequent I/O access

Fig. 5. Characteristic of Application

4 Performance of Simultaneous Execution of Multiple Application Programs Parallelized and Power-Controlled by OSCAR Compielr

This section evaluates execution performance and consumed power on the RP2 eight core homogeneous multicore with DVFS and power gating capabilities when multiple media application programs, which are automatically parallelized and power-controlled by OSCAR compiler, are executed simultaneously sharing eight cores.

4.1 Performance of Simultaneous Execution of Multiple Low-Processing-Load Applications

This sub-section describes consumed power on RP2 when low-processing-load applications shown in Figure 5 namely each application program is a real-time AAC encoder to encode a 16 seconds music file in 16 seconds, are executed in parallel. Figure 6 shows consumed power when one to eight light-load AAC encoders are executed in parallel using different numbers of processor cores on the RP2. The vertical axis shows the consumed power and the horizontal axis shows number of processor cores, or PEs, to execute the two AAC encoders. In each number of PEs, AAC encoder programs less than number of PEs are executed. For example, on one PE, just one AAC encoder is executed sequentially with a non power controlled mode shown in left bar and a power controlled mode shown in right bar. Also, on eight PEs, the left bar shows consumed power when one AAC encoder is executed in parallel, each of which uses one PE, with non power controlled mode. The second left bar shows the consumed power when one AAC encoder is executed on 8 PEs in parallel with the power control. The third left bar shows the consumed power when two AAC encoders are executed in parallel each of which uses 4 PEs respectively. The forth bar shows the power when 4 AAC encoders are executed in parallel, each of which uses 2 PEs respectively. The fifth left bar shows the power when 8 AAC encoders are executed in parallel, each of which uses 1 PE. In other words, if "N" application programs are executed on "M" PEs, "M/N" PEs are assigned to each application programs in each number of PEs. In this light-load AAC encoder, a PE can easily execute the AAC encoder keeping real-time deadline. On the1 PE, though one AAC encoder without power control, or an ordinary single core execution with 100% frequency (600 MHz) and the highest voltage (1.4V), consumes 1.89 W, the power controlled AAC encoder just requires 0.59 W since OSCAR compiler chose 1/8 frequency (75 MHz) and the lowest voltage (1.0V) for waiting the dead line. Namely, OSCAR compiler gives us 69% of power reduction when one AAC encoder is executed on one PE. On 2 PEs, one AAC encoder parallelized on 2 PEs without power control consumes 2.19 W. One AAC encoder parallelized to 2 PEs with power control consumes 0.59 W that is the same as on the 1 PE since OSCAR compiler applies appropriate DVFS control in nano-second order during execution and power gating to the second PE and lowest frequency and voltage power states to the first PE during waiting for the deadline. Also, two AAC encoders on 2 PEs, in which each AAC encoder is executed on 1 PE with power control, consumed just 0.66 W. In the two AAC encoder real-time execution, OSCAR compiler reduces power by 70% from 2.19W to 0.66W. On 4 PEs, one AAC encoder parallelized for 4 PEs without power control consumes 2.75 W. One AAC encoder parallelized to 4 PEs with power control consumes 0.59 W, or the same as on 1 PE. The two AAC encoders on 4 PEs, in which each AAC encoder is executed on 2 PEs with power control, consumed just 0.68 W. Also, four AAC encoders on 4 PEs, in which each AAC encoder is executed on 1 PE with power control, consumed just 0.78 W. In other words, OSCAR compiler reduces the power by 72% from 2.19W to 0.66W when four AAC encoders are executed on 4PEs. On 8 PEs, one

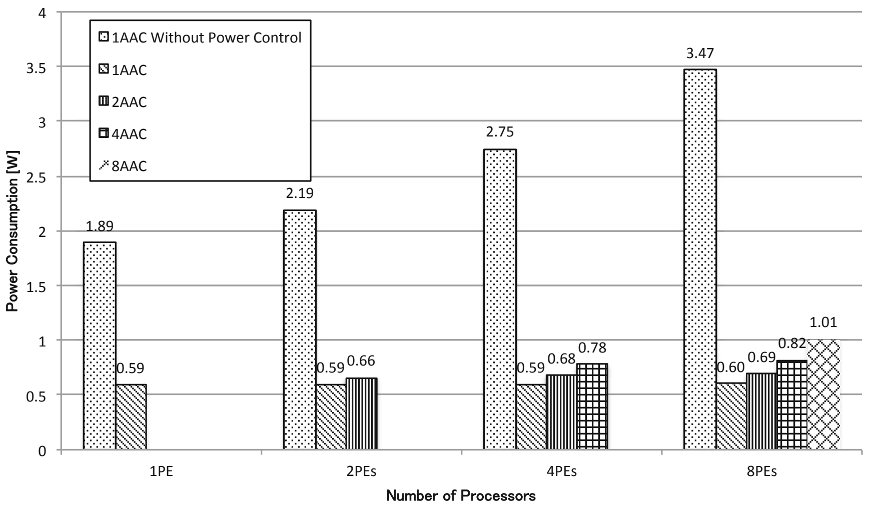


Fig. 6. Power Consumption of Low processing-load applications (AAC encoder)

AAC encoder parallelized for 8 PEs without power control consumes 3.47 W. One AAC encoder parallelized to 8 PEs with power control consumes 0.60 W, or the almost same as on 1 PE. The two AAC encoders on 8 PEs, in which each AAC encoder is executed on 4 PEs with power control, consumed just 0.69 W that is similar to 0.66 W on 2PEs and 0.68 W on 4 PEs. The four AAC encoders on 8 PEs, in which each AAC encoder is executed on 2 PEs with power control, consumed 0.82 W that is just 0.04 W larger than on 4 PEs. Also, eight AAC encoders on 8 PEs, in which each AAC encoder is executed on 1 PE with power control, consumed just 1.01 W. In other words, OSCAR compiler reduces the power by 71% from 3.47 W to 0.66 W when eight AAC encoders are executed on 8 PEs. Here, we should pay attentions that the eight AAC encoder execution on 8 PEs just requires the just 0.13 W for one AAC encoder though the one AAC execution on 1 PE without power control consumes 1.89 W, namely the eight core execution gives us 93% power reduction, and the one AAC execution on 1 PE with power control consumes 0.59 W, namely the eight core execution gives us 78% power reduction. These results show the simultaneous execution of multiple low-load application programs with OSCAR compiler's power control gives us huge reduction in a single application average power consumption.

4.2 Performance of Simultaneous Execution of Multiple Intermediate Processing-Load Applications (AAC encoder)

This sub-section describes consumed power on RP2 when intermediate- processing-load applications shown in Figure 5, namely each application program is a super-real-time AAC encoder to encode a 16.0 seconds music file in 3.0 seconds, are executed in parallel. In this application, a single PE needs 2.7 seconds to process and manages to satisfy the real-time deadline with the almost highest clock frequency. Figure 7 shows consumed power when one to four intermediate-load AAC encoders are executed in parallel using different numbers of processor cores on the RP2. Here, the case of eight AAC encoders is not shown because eight AAC encoders on 8 PEs exceeds RP2's memory capacity. The vertical axis shows the consumed power and the horizontal axis shows number of PEs as Figure 7. On the1 PE, one AAC encoder without power control consumes 1.95 W and the power controlled AAC encoder requires 1.78W, namely 9% power reduction, since in the intermediate-load there is only a 0.3 s to wait for dead line in which OSCAR compiler can choose 1/8 frequency (75 MHz) and the lowest voltage (1.0V). On 2 PEs, one AAC encoder parallelized for 2 PEs without power control consumes 2.27 W. One AAC encoder parallelized to 2 PEs with power control consumes 1.18 W that is 40% power reduction from 1.95 W on 1 PE without power control and 34% power reduction from 1.78 W on 1PE with power control since OSCAR compiler applies appropriate DVFS and power gating automatically. Also, two AAC encoders on 2 PEs, in which each AAC encoder is executed on 1 PE with power control, consumed 2.20 W. In the two AAC encoder real-time execution, OSCAR compiler reduces power by 3% from 2.27W to 2.20W. On 4 PEs, one AAC encoder parallelized for 4 PEs without power control consumes 2.85 W. One AAC encoder parallelized to 4 PEs with power control consumes 0.84 W, namely 57% power reduction from 1.95 W of one AAC on 1 PE without power control and 53% power reduction from 2.78W of one AAC on 1 PE with power control. On 8 PEs, one AAC encoders without power control consumes 3.59 W. One AAC encoder parallelized to 8 PEs with power control consumes 0.87 W that is the same as one AAC on 4 PEs. The two AAC encoders on 8 PEs, in which each AAC encoder is executed on 4 PEs with power control, consumed just 1.13 W that is 78% reduction from 2.20 W of two AACs on 2 PEs which is the minimum number of PEs to satisfy the deadline and 26% reduction from 1.53 W of two AACs on 4PEs. The four AAC encoders on 8 PEs, in which each AAC encoder is executed on 2 PEs with power control, consumed 2.21 W that is the same as 2.20W of two AACs on 2PEs. These results show the simultaneous execution of multiple intermediate-load application programs with OSCAR compiler's power control gives us large reduction by parallel execution of each application since the parallel processing make chances of DVFS and power gating during execution.

4.3 Performance of Simultaneous Execution of Multiple Intermediate-Processing-Load Applications (MPEG2 decoder)

Figure 8 shows power consumption of MPEG2 decoder multiple executions. Horizontal axis indicates sum of processor PEs used by all applications. Vertical axis indicates power consumption of entire chip. Each bar indicates number of MPEG2 decoders executed multiple. 1MPEG2 means power consumption of one MPEG2 decoder execution. 2MPEG2 means power consumption of two MPEG2 decoders, which is using half number of PEs indicated by X-axis. Power

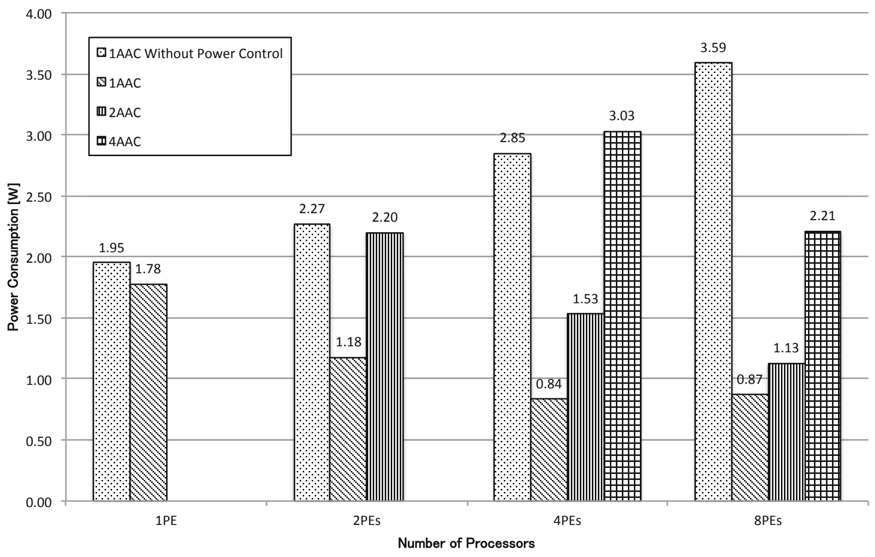


Fig. 7. Power Consumption of Intermediate processing-load applications (AAC encoder)

consumption reduce from 1.99W (one PE) to 1.0W (four PEs) by applying parallel processing. Power consumption of 2 MPEG2 decoders (four PEs for each) is 1.46W. This is almost same power consumption of 1 MPEG2 decoder (one PE, 1.49W). Middle computational load applications can be reduce power consumption by applying parallel processing and executing multiple applications. Power consumption of each MPEG2 decoder is 0.73W, which reduce power consumption at 51% against 1 MPEG2 decoder (one PE). MPEG2 decoder has high bus pressure by Prescanning. MPEG2 decoder should be executed to shift I/O timing. This is because power consumption reduction ratio of MPEG2 decoder is relatively low against AAC encoder. Processors voltage control domain should be divided to control power effectively when middle computational load applications are executed.

4.4 Performance of Simultaneous Execution of Multiple High-Processing-Load Applications (MPEG2 decoder)

Figure 9 shows power consumption of MPEG2 decoder multiple executions. horizontal axis indicates sum of processor PEs used by 352x128 resolution MPEG2 decoder and 352x240 resolution MPEG2 decoder. Vertical axis indicates power consumption of entire chip. When different resolution MPEG2 decoders are executed multiple, timing of FV control are differ. This is because execution time of each decode stage are different. Thus, supply voltage (controlled by entire chip) is hard to down. In addition, power status of each application is different.

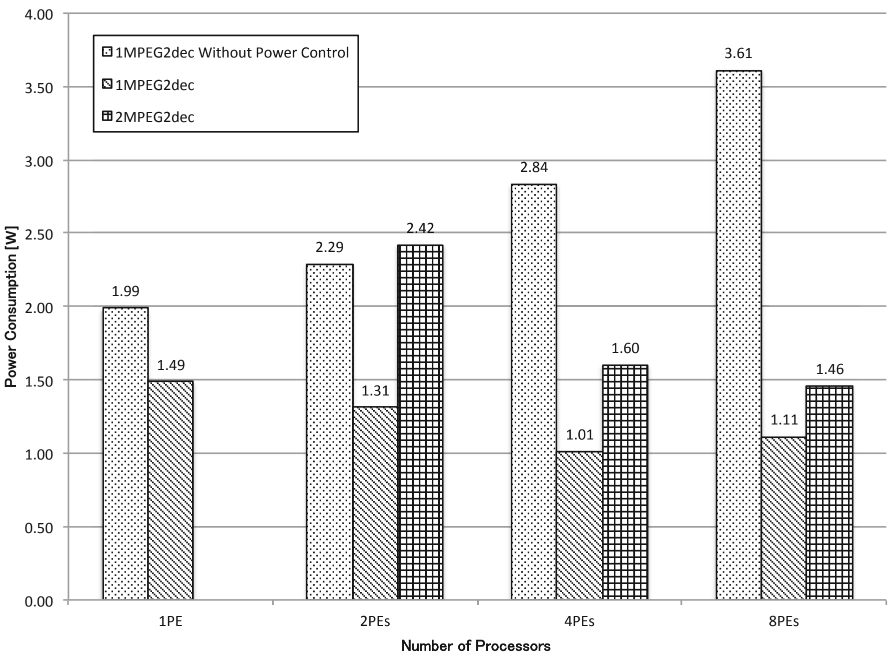


Fig. 8. Power Consumption of Intermediate processing-load applications (MPEG2 decoder)

For example, power status is FULL+MID by 2PE+2PE execution. At this time supply voltage is 1.40V (FULL), so power consumption is 2.28W. This reduces only 3% against 2PE+1PE execution. However, power status is MID+LOW by 4PE+2PE execution. At this time supply voltage is 1.20V and power consumption is 1.85W. This reduces 21% against 2PE+1PE execution (2.35W). When middle computational load application and high computational load application are executed multiple, hardware co-operation to divide supply voltage domain is effective to reduce power consumption.

5 Conclusions

This paper has evaluated the power reduction scheme of OSCAR automatic parallelizing compiler when multiple media application programs parallelized and po-wer-controlled by OSCAR compiler are executed simultaneously on the Renesas / Hitachi / Waseda RP2 eight core homogeneous multicore processor. Execution performances are almost no differences from a single program execution when multiple parallelized programs are executed simultaneously since OSCAR compiler's cache memory optimization function minimizes main memory, or off-chip shared memory, accesses and prevents main memory contentions. Power

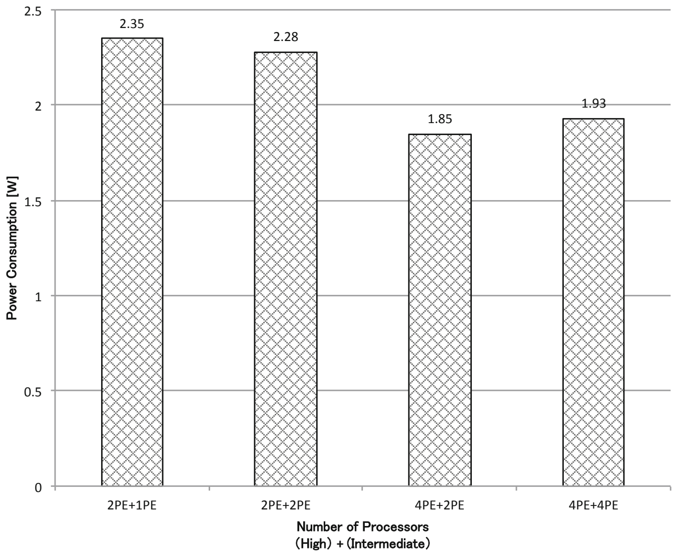


Fig. 9. Power Consumption of High processing-load application and Intermediate processing-load application

consumption of multiple applications with relatively light computational load was reduces by 68% against non-power controlled applications. When multiple applications are executed simultaneously, total power consumption was 1.01W with 8 AAC encoders, each of which was executed on one core. At this time, an average power consumption of each AAC encoder was 0.13W and this value was 22% of power compared with one application executed on 8 cores. This result shows that when we execute 8 AAC encoder (1 core for each), power consumption reduce at 78% against 1 AAC encoder (8 cores). Light computational load applications should be executed as much as possible to reduce power consumption of one application. Average power consumption of multiple applications with middle computational load reduces at 30% by parallel executions. In addition, when 4 AAC encoders (2 cores for each) are executed, power consumption is 0.55W by 1 AAC encoder. This power consumption reduces at 37% against 1 AAC encoder (8 cores), and reduces at 69% against 1 AAC encoder (1 cores). When multiple meddle computational load applications are executed, parallel processing can reduce power consumption. Average power consumption of 2 MPEG2 decoders (4 cores for each) is lower than average power consumption of 1 MPEG2 decoder (1 core). At this time, power consumption of each MPEG2 decoder (0.73W) reduces at 51% against 1 MPEG2 decoder (1.49W). However, if voltage control domain is divided by hardware, there is more room to reduce power consumption. This tendency is more notable at executing both high computational load application and middle computational load application multiple. This paper confirmed automatic parallelization and automatic power control scheme of OSCAR compiler is effective to reduce power consumption of multiple applications.

Acknowledgements. I would like to thank Junji Sakai (NEC Corporation) whose comments and suggestions were of inestimable value for our study. I am also indebt to Kazuhisa Ishizaka (NEC Corporation) whose comments made enormous contribution to our work.

A part of this research has been supported by NEDO ”Advanced Heterogeneous Multiprocessor”, NEDO ”Multi core processors for realtime consumer electronics” and STARC ”Automatic Parallelizing Compiler Cooperative Single Chip Multiprocessor”.

References

1. Pham, D., et al.: The design and implementation of a first-generation cell processor. In: Proceeding of the IEEE International Solid-State Circuits Conference (2005)
2. Hayase, K., Shibahara, S., Nishii, O., Hattori, T., Hasegawa, A., Takada, M., Irie, N., Uchiyama, K., Odaka, T., Takada, K., Kimura, K., Kasahara, H., Yoshida, Y., Kamei, T.: A 4320mips four-processor core smp/amp with individually managed clock frequency for low power consumption. In: 2007 IEEE International Solid-State Circuits Conference, ISSCC 2007 (February 2007)
3. Yoshida, Y., Hayase, K., Hayashi, T., Nishii, O., Yasu, Y., Hasegawa, A., Takada, M., Ito, M., Mizuno, H., Uchiyama, K., Odaka, T., Shirako, J., Mase, M., Kimura, K., Kasahara, H., Ito, M., Hattori, T.: An 8640 mips soc with independent power-off control of 8 cpu and 8 rams by an automatic parallelizing compiler. In: Proc. of IEEE International Solid State Circuits Conference, ISSCC 2008 (February 2008)
4. Kiyoshige, Y., Nitta, Y., Matsui, S., Nishii, O., Hasegawa, A., Ishikawa, M., Yamada, T., Miyakoshi, J., Terada, K., Nojiri, T., Satoh, M., Mizuno, H., Uchiyama, K., Wada, Y., Kimura, K., Kasahara, H., Maejima, H., Yuyama, Y., Ito, M.: A 45nm 37.3gops/w heterogeneous multi-core soc. In: IEEE International Solid-State Circuits Conference, ISSCC 2010 (February 2010)
5. Cornish, J.: Balanced energy optimization. In: International Symposium on Low Power Electronics and Design (2004)
6. Kimura, K., Mase, M., Mikami, H., Miyamoto, T., Shirako, J., Kasahara, H.: OSCAR API for Real-Time Low-Power Multicores and Its Performance on Multicores and SMP Servers. In: Gao, G.R., Pollock, L.L., Cavazos, J., Li, X. (eds.) LCPC 2009. LNCS, vol. 5898, pp. 188–202. Springer, Heidelberg (2010)
7. Honda, H., Iwata, M., Kasahara, H.: Coarse grain parallelism detection scheme of a fortran program. Trans. of IEICE J73-D-1(12), 951–960 (1990)
8. Kasahara, H., et al.: A multi-grain parallelizing compilation scheme on oscar. In: Proc. 4th Workshop on Language and Compilers for Parallel Computing (1991)
9. Kasahara, H.: Advanced automatic parallelizing compiler technology. IPSJ Maganie (April 2003)
10. Ishizaka, K., Miyamoto, T., Shirako, J., Obata, M., Kimura, K., Kasahara, H.: Performance of OSCAR Multigrain Parallelizing Compiler on SMP Servers. In: Eigenmann, R., Li, Z., Midkiff, S.P. (eds.) LCPC 2004. LNCS, vol. 3602, pp. 319–331. Springer, Heidelberg (2005)

11. Obata, M., Shirako, J., Kaminaga, H., Ishizaka, K., Kasahara, H.: Hierarchical Parallelism Control for Multigrain Parallel Processing. In: Pugh, B., Tseng, C.-W. (eds.) LCPC 2002. LNCS, vol. 2481, pp. 31–44. Springer, Heidelberg (2005)
12. Kasahara, H., Honda, H., Iwata, M., Hirota, M.: A compilation scheme for macro-dataflow computation on hierarchical multiprocessor system. In: Proc. Int Conf. on Parallel Processing (1990)
13. Kasahara, H., Honda, H., Narita, S.: Parallel processing of near fine grain tasks using static scheduling on oscar. In: Proceedings of Supercomputing 1990 (November 1990)
14. Shirako, J., Oshiyama, N., Wada, Y., Shikano, H., Kimura, K., Kasahara, H.: Compiler Control Power Saving Scheme for Multi Core Processors. In: Ayguadé, E., Baumgartner, G., Ramanujam, J., Sadayappan, P. (eds.) LCPC 2005. LNCS, vol. 4339, pp. 362–376. Springer, Heidelberg (2006)
15. Lee, C., et al.: Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In: 30th International Symposium on Microarchitecture, MICRO30 (November 1997)
16. Iwata, E., et al.: Exploiting coarse-grain parallelism in the mpeg-2 algorithm. Technical Report CSL-TR-98-771 (September 1998)

Double Inspection for Run-Time Loop Parallelization

Michael Philippsen[1,⋆], Nikolai Tillmann[2], and Daniel Brinkers[1]

[1] University of Erlangen-Nuremberg, Computer Science Dept., Programming Systems Group, Erlangen, Germany
{philippsen,daniel.brinkers}@cs.fau.de
[2] Microsoft Research, One Microsoft Way, Redmond, WA, USA
nikolait@microsoft.com

Abstract. The Inspector/Executor is well-known for parallelizing loops with irregular access patterns that cannot be analyzed statically. The downsides of existing inspectors are that it is hard to amortize their high run-time overheads by actually executing the loop in parallel, that they can only be applied to loops with dependencies that do not change during their execution and that they are often specifically designed for array codes and are in general not applicable in object oriented just-in-time compilation.

In this paper we present an inspector that inspects a loop *twice* to detect if it is fully parallelizable. It works for arbitrary memory access patterns, is conservative as it notices if changing data dependencies would cause errors in a potential parallel execution, and most importantly, as it is designed for current multicore architectures it is fast – despite of its double inspection effort: it pays off at its first use.

On benchmarks we can amortize the inspection overhead and outperform the sequential version from 2 or 3 cores onward.

1 Introduction

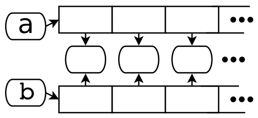
Just-in-time compiled object-oriented script languages like JavaScript [3] are getting important and heavily used in practice[1] and their use is no longer restricted to small, short-running or interactive applications. But they are not well-suited for the multicore future as, in general, they do not offer any means to express parallelism. Sequential JavaScript code runs in every web browser and will therefore face the fate of every sequential code in the foreseeable multicore future. But due to the dynamic typing and due to the fact that such codes in general are not the typical regular array-based codes known from scientific programs, well-known results from automatic parallelization of high-performance codes are hardly applicable. Hence, unless/until there is some way to express parallelism

⋆ For this work the author has spent time at Microsoft Research during his sabbatical.

[1] As of March 2011, www.tiobe.com reports that about 35% of today's code is written in dynamically typed languages and that in the last five years about half of the most used programming languages have been scripting languages.

in those languages, run-time parallelization is the only viable way to go – especially since the dynamic typing renders even fewer loops amenable to static parallelization than usual. Consider the following JavaScript example:

```
var a = [{v:1}, {v:1}, {v:1} ...];
var b = [a[0],  a[1],  a[2] ...];
for(i = 0; i < 10.000; i++){
   a[i].f = 2 * b[i].v;
}
```



Two complications in this (pseudo-)code cause traditional parallelizers to fail. The first problem is that a[] and b[] are arrays of references to objects that reside on the heap instead of arrays of primitive types. As a[i] and b[j] can refer to the same object parallelizers can no longer work on array indices but they must consider general memory addresses and data dependences that forbid parallel execution which are hard to detect by means of alias analysis. The second problem is that the code writes to the field f of the a-objects. Although there might be some objects that already have this field; for others the field is created on demand at the moment of the assignment.

In addition to work on privatization and on detection of reductions [12], most work on run-time parallelization is based on the inspector/executor idea of Saltz et al. [13]. Their idea is that in a first loop, an inspector performs a dry-run of the loop and looks at all array indices that the loop will touch – without actually executing the operations of the given loop. If certain addresses are touched by more than one iteration and not just for reading, then there is a data dependence that requires that these iterations are performed in their original order, i.e., not in parallel to each other. If there are no such dependencies, then the loop is fully parallelizable; otherwise the loop is partially parallelizable in so-called wave fronts. All those iterations of the original loop are scheduled to a single wave front that can be executed in parallel to each other as they do not have any cross-iteration dependencies to each other. Dependencies only exist to iterations that are scheduled to other wave fronts and that are hence separated by at least one synchronization barrier.

The general problem is that the dry-run takes time as the inspector must evaluate all addresses and must keep track of each of these addresses in a book-keeping data structure to detect potential dependencies. The longer this inspection takes, the more parallelism must be found so that the total time needed for both the inspector plus the subsequent executor (that executes all the loop's iterations in wave fronts) is still smaller than a pure sequential execution of the original loop. The runtime of the inspector is thus the most important aspect and the crucial stumbling block of the whole idea.

There are four responses to this challenge. The first is optimistic/speculative, i.e., to execute the loop in parallel and check whether data dependencies occurred. In that case some form of roll-back purges the wrong effects and restarts with the original sequential loop. (Check the literature for thread level speculation [14] or transactional memories [5] for solutions.) Note that the optimistic approach cannot avoid the book-keeping cost for detecting dependencies.

Book-keeping must still be fast. The second response is an amortization argument: the inspector/executor approach is only applied for loops with compute intensive bodies that do much work with few array elements. (In some early papers [6] the authors added a few (and at that times slow) trigonometric functions for the mere purpose of making their benchmarks look nice.) The third response is a second-degree amortization argument. If the data dependencies are not changed by the loop itself, then the inspector can be run once and its result can later be used for many parallel executions of the loop at hand. This works well, if for example the parallelized loop is buried in an outer loop. Unfortunately, such a so-called schedule reuse [10] often does not work in dynamically typed script languages or on real-world problems [7], since in general no static analysis can assure that the data dependencies will not change. Finally, the fourth countermeasure is to perform the inspection itself in parallel to save time. However, when two parallel inspectors try to register in a book-keeping data structure that a single address is being touched, this will – at least conceptually – require slow synchronization or critical sections to guard the tracking data structure. Thus, it is essential to avoid such synchronization wherever possible. This is even more difficult on modern multicores as their memory systems are typically not sequentially consistent and cores might see changes at different times. And even with the synchronization demands solved, the individual effort for registering every single address must be kept tiny and without branches to keep the processor pipelines busy.

The following section presents such an inspector that detects if a loop can be fully executed in parallel. The main idea is to inspect the loop twice, but with only a single synchronization barrier and memory fence in-between. Our inspector's book-keeping effort for registering every single address is tiny – just a few machine instructions with only one conditional branching instruction. It amortizes easily and its complexity is in the order of the complexity of the loop to be parallelized (instead of being in the order of the size of the main memory). We show-case the performance of double inspection in Sec. 3 before we have a quick look at the related work in Sec. 4 and conclude.

2 Double Inspection

2.1 Basic Idea – Overly Conservative

The main idea of the double inspection is to inspect the loop in two phases, each of which is done in parallel by a set of inspector threads. To ease understanding we discuss a simplified and overly conservative version of the double inspection approach first. This version cannot parallelize loops with loop-independent dependencies.[2] The two subsections that follow will remove this restriction and they will refine and optimize the idea.

[2] A loop-independent dependence is a data-flow dependence that already exists in the loop body even if the loop control structure is taken away.

We assume that the inspector is deterministic so that each phase will observe the same sequence of memory accesses. For a formal view on our inspector, we start with a set of definitions. Let $U(I)$ be the set of all memory addresses from which iteration I reads. (Note that we talk about addresses and not array indices.) $D(I)$ is the set of all memory addresses to which iteration I writes. $W(A)$ is the set of iterations that write to the memory address A.

Foreach iteration I of the loop do (maybe in parallel):

- **(1):** If $\exists\ u \in U(I)$ with $W(u) \neq \emptyset$ then there is a flow or anti dependence. (This is overly conservative as we will discuss in Sec. 2.2.)
- **(2):** If $\exists\ d \in D(I)$ with $|W(d)| \neq 1$ then there is an output dependence.

Note that it would be possible to find all dependencies of one iteration I by iterating through the corresponding W set for every address in $U(I)$ and $D(I)$.

We now reduce $W(A)$ to an arbitrarily selected $W'(A)$. Every $W'(A)$ holds only one arbitrary element of $W(A)$. We can use a similar algorithm with these sets. Step (1) stays unmodified, as we only check whether the set is empty and do not care about the elements or their number. Step (2) has to change slightly. As the size of the set $W'(A)$ is known to be one, we now check whether the element in $W'(d)$ is I:

Step (2'): If $\exists\ d \in D(I)$ with $W'(d) \neq I$ then there is an output dependence.

With the modified sets W' we are no longer able to find all dependencies of an iteration I because the information about some dependency may be discarded. But we are still able to detect, *if* there are any dependencies. If $W(d)$ has more than one element and $W'(d)$ consists of the element I, Step (2') will not detect the dependence for iteration I. But it will of course detect the dependence for all other elements in $W(d)$.

We represent all sets W' by means of an array `used`. Each `used[A]` in this array holds one element of the set $W(A)$. An empty set is represented by the special value null. Parallel inspector threads fill the array `used` in the first loop of the double inspection, the **pre-registration loop** and register all memory addresses to which the inspected loop writes. To do so, there is a copy of the original loop in which each of the writes is replaced by a macro that performs the book-keeping. After the writes are gone, dead code elimination purges almost everything expect for the macros (and for the relavent control structures).

In the running example, with thread-specific values for `lwb` and `upb`, the pre-registration loop of the inspector would therefore be:

```
for(i = lwb; i < upb; i++){
   preregister(hash(&a[i].f), i);  //only for writes
}
```

Note that instead of the real addresses, we use hashed versions to make sure that the book-keeping array stay small. The smaller it is, the more likely are false positives. In garbage-collected languages that may move around objects the "hashed" value needs to survive the collector's activities. Our current prototype ignores that and uses the hash function `(ptr<<3)&0xffff`.

The pseudo-code of the inspector's pre-register macro is:

```
int used[]; //shared array, used by all inspector threads
void preregister(int addr_hsh, int iteration){
   used[addr_hsh] = iteration;
}
```

Note, that at one point of time several inspector threads might write to the same slot of the book-keeping array `used`. This is fine and one of the main design principles of double inspection, as our only requirement is that the machine architecture makes sure that one of the threads will win, i.e., when all inspector threads have finished, one uncorrupted iteration number will be in the books. It is irrelevant that write operations will be buffered in the store buffer or in the caches of the individual cores. We only have to make sure that there is a single synchronization barrier with a memory fence after this first inspection pass.

After this first pass, the inspector threads perform their second pass over all the iterations. This time, in a **checking loop**, we check both for the reading and writing memory accesses whether they cause dependencies and hence render the original loop as not fully parallelizable.

Here is the checking loop for the running example (again, everything is removed except for the relevant control structures):

```
for(i = lwb; i < upb; i++){
   read(hash(&b[i].v), i)
   write(hash(&a[i].f), i);
}
```

The macros for checking are shown below.

```
void write(int addr_hsh, int iteration){
   if(used[addr_hsh] != iteration) alert(); //alert, if someone else wrote
}
void read(int addr_hsh, int iteration){
  if(used[addr_hsh]) alert(); //alert, if ever written
}
```

Let's look at the `write` first. If an inspector thread is the only one that has pre-registered that an iteration has written to a certain address, it will find in the second pass that the iteration number is still in the books, and everything is fine. If however, there are more than one iterations that write to an a address, at least one of the inspectors will detect a difference either because the thread itself has registered two iteration numbers or because its iteration number has been overwritten by another inspector thread. The asynchronous pre-registrations can happen without synchronization and in any order, since *at least one* of the inspector threads will find a different value in the second pass. Hence, output dependencies (= two writes to the same address by different iterations) will be detected – the `alert` will flag the loop as not fully parallelizable.

In the same way, the inspector that checks an address for reading will flag a dependence if that address has been pre-registered (i.e., it is written to) at all.

This basic double inspection is overly conservative, since even loops that only have loop-independent dependencies will be flagged as not fully parallelizable. For example, even if a loop iteration just reads a value and updates it without any interference from other iterations, this basic double inspector signals a dependence as the updated address has already been pre-registered when the checking loop tests the read access and finds some non-null entry in the books.

In inspector/executor systems it is crucial that the inspector and the executor follow the same control paths. Being conservative helps, because data value dependent execution paths will be recognized.[3]

Let's make the inspector more aggressive now. Sec. 2.3 will avoid to re-initialize the book-keeping array for every inspection.

2.2 First Improvement: Tolerate Flow Dependences

The naive approach to extend the basic version to ignore loop-independent dependencies is to replace the null-check in the `read`-macro. This "improved" version of the macro would only flag a dependence if some *other* iteration had written to an address that is read in the current iteration.

```
void read(int addr_hsh, int iteration){
   if(used[addr_hsh] && (used[addr_hsh] != iteration)) alert();
}
```

Unfortunately, this is too simplistic. The problem are dependencies that are a result of the computations in the loop itself. Consider the following example:

```
int A[4] = {0,1,2,3}
for(int i=0; i<3; ++i){
   A[i] = i+1;
   A[A[i]] = i+1;
}
```

In this example, just registering and checking the addresses of `A[i]` and `A[A[i]]` will not detect any conflicts as `A[A[i]]` stays equal to `A[i]` because the inspector is a dry-run and does not execute the computations/assignments. The data dependences that prevent full parallelization in this example are however a result of these modifications. A flow dependence that reads a value and later contributes to an overwriting of it, is ok as the assignment is in a way the last thing that happens to that memory address. In contrast, an anti-dependence that writes a value first and later (re-)reads it (or something it depends on), might change dependencies. Thus, inspectors that are based on a dry-run can ignore loop-independent flow dependences but they have to be conservative and signal a potential threat to parallelizability upon a loop-independent anti-dependence.

The insight for an efficient implementation is that if there is a loop-carried dependence for a certain memory address, then the checking-loop will find a mismatch of iteration numbers at some point. If there are only loop-independent dependencies for that address, then *only one* inspector thread reads and modifies the corresponding slot of the book-keeping array. As no synchronization and memory consistency measures are needed to guard against interfering inspector threads in the checking loop, we can use a bit of this slot to detect the nature of a loop-independent dependence. Upon a write, this so-called flow bit is set. When

[3] We apply a pre-test to stay away from loops with conditional branches that depend on loop-external side effects.

a read finds the flow bit already set, there is an anti-dependence and hence a threat to full parallelizability. If the read precedes the write, the read finds the flow bit still un-set and the check remains quiet.

It is crucial to implement this efficiently. To do so, we (for now) use the least significant bit of the `used` integer values. The pre-registration registers `2*iteration` which shifts all bits to the left and lets the least significant flow bit un-set. In the checking loop, `write` ignores the flow bit in the comparison, but then sets it by assigning `2*iteration+1`. The comparison in `read` is sensitive to the flow bit. If the loop body has written to an address before, the `read`-macro will flag a parallelizability problem. Writes after reads are ok.

We expect the `iteration` numbers to be non-negative, fitting into 31 bits. A guard just before the inspector loop checks the lower and upper bounds to ensure that no overflows occur in the actual inspector.

```
void preregister(int addr_hsh, int iteration){
   used[addr_hsh] = 2 * iteration;
}
void write(int addr_hsh, int iteration){
   if((used[addr_hsh] | 1) != (2  * iteration + 1)) alert();
   used[addr_hsh] = iteration * 2 + 1;
}
void read(int addr_hsh, int iteration){
  if(used[addr_hsh] && (used[addr_hsh] != iteration * 2)) alert();
}
```

On our benchmarks it has improved the speed of book-keeping by about 2% on average that `write` does not compare `used[addr_hsh]&(~1)` to `2*iteration`. The reason is that the compiler can keep `2*iteration+1` in one iteration variable when unrolling the checking loop, whereas the bitmasking code would need two iteration variables.

2.3 Second Improvement: Avoid Re-initialization

Up to now the inspectors have to re-initialize the book-keeping array `used` for every loop that has to be inspected. To avoid re-initialization, we turn `used` into an array of 64 bit values. The upper 32 bits are used for a generation number plus the extra bit needed to tolerate flow dependencies. The generation counter `base` is incremented once for every loop to be inspected. Only after `MAX_INT/2` inspections, that is almost never, the `used`-array needs to be re-initialized.

```
long used[]; //shared array, used by all inspector threads
int base = 0;
void preregister(int addr_hsh, int iteration){
    used[addr_hsh] = (base<<33) + iteration;
}
void write(int addr_hsh, int iteration){
   if(((int32*)&used[addr_hsh])[0] != iteration) alert();
   ((int32*)&used[addr_hsh])[1] = 2 * base + 1;
}
void read(int addr_hsh, int iteration){
   if(((used[addr_hsh] - (base<<33)) ^ iteration) > 0) alert();
}
```

An optimization insight is that `write` does *not* have to check whether the value is from the correct generation, since *it has to be* since each value a `write` checks has been written by a corresponding `preregister`. Hence `write` only compares the lower 32 bit to `iteration` which automatically also ignores the flow bit. After the comparison the flow bit is set in the upper `int`.

Our x86-64 assembler code is fast and only needs five instructions, namely `lea` and `mov` to compute the hash value and to access the booked value, a `cmp`, a (short) conditional jump `jne`, and another `mov` for the assignment. A register holds `2*base+1` as it is fixed for the whole inspection.

A straightforward implementation of `read` would first check whether the value in the books is from the current generation and only then look at the iteration numbers. That would result in two conditional jumps per `read`. On current pipeline architectures with branch prediction this turns out to be too slow. It is crucial to avoid the jump on a generation mismatch. To do so, there is the `xor` in the `read`-operation. We first subtract the current `base`-value from the value in the book in the upper `int` (`base<<33` is a pre-computed fixed value that is kept in a register). If the value in the books originated from an earlier version, then the result is negative, which is ok, as there is no dependence. If the value is from the same generation but from a different iteration, then the result of the `xor` is positive. The value is also positive, if after subtracting `base` and `xor`-ing away the iteration number, only the flow bit remains switched on. Hence, with a simple subtraction and an `xor`, we merge the generation test, the iteration test, and the flow test into a single conditional jump. Moreover, in the case of x86-64, the result of the comparison to 0 is implicitly remembered in status bits after the `xor` operation, so that no explicit machine instruction is needed to perform the comparison. Thus, the `read`-macro also needs just five machine instructions (`lea`, `mov`, `sub`, `xor`, and `jg`).

As an alternative to generation numbers one could use two sets of bookkeeping arrays. While one set is being used for inspection the other is cleaned. However, since current multicore processors do not yet have an abundance of cores, we need all of them so that inspection plus execution performs faster than the sequential loop.

3 Benchmarks

For the measurements we have used a single Intel Core i5 760 chip (4 cores) running at 2.80 GHz with 8 MB shared cache and 8 GB of RAM (DDR-1333). The machine runs Windows 7 x64. We have used the Visual Studio 2010 compiler.[4]

We choose four fully parallelizable benchmarks to test our inspector's performance. The loops under consideration are tiny, i.e., 3–5 lines of code long. We extracted them from the SunSpider JavaScript benchmarks, where the loops

[4] We have also performed measurements on an Intel Xeon X7560 "Nehalem-Ex" chip (8 cores + hyperthreading) running at 2.27 GHz with 24 MB shared cache and 512 GB of RAM (DDR3-1333). This machine runs Linux 2.6.32. We have used the GNU g++ 4.4.3 compiler. We found the same overall behavior and only show i5 times.

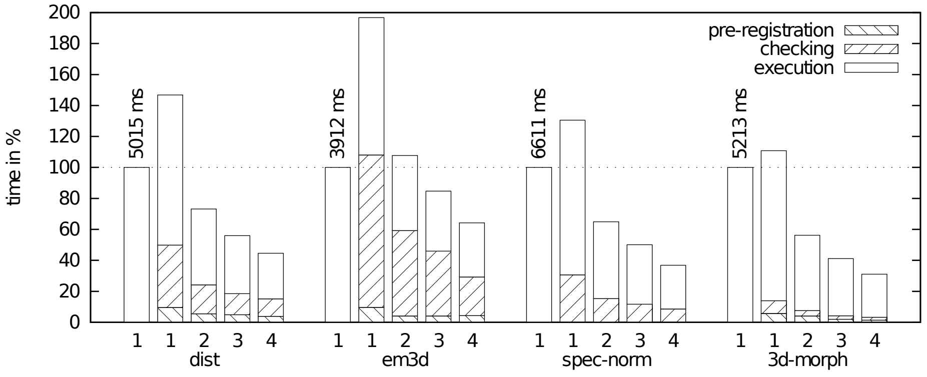


Fig. 1. Sequential performance and parallel performance with double inspection. Numbers show the runtimes of the parallelizable loop without framework setup on 1-4 cores.

have significant runtime which we determined with the SPUR tracing JIT [1]. We have implemented them in C/C++ since this made it easier to interact with our C-based inspection framework. As can be seen in Fig. 1, the pre-registration loop, the checking-loop, and the executor scale nicely and from two cores on outperform the sequential execution on a single core. (On em3d, we need 3 cores to see a speedup.) So even if the benchmark loops are only executed just once, double inspection and parallel execution are faster than executing the loops sequentially. The cost of the inspection is amortized immediately, i.e., without schedule and schedule reuse or speculation.

Dist calculates the distance between pairs of 2D points. What rules out static parallelization are the indirections: another array holds pointers to the 2D points that need to be used in the distance computation. This is similar to traversing a linked list of objects and applying a side-effect free method to each of the objects. Although this is parallelizable, there is the underlying container implementation whose links and pointers are only known at runtime. In Dist there are four read accesses and one write access per iteration. This explains the relation of the time spent in the pre-registration versus the checking loop.

Em3d propagates electro magnetic waves that are stored in a bipartite graph with many indirections. In our benchmarks we have a used an input graph that can be processed in parallel. Computing the indirections, i.e., the addresses of the data that is needed at run time is costly. While the sequential execution only needs to do this computation once, we (currently) re-compute the information in both inspector loops and in the executor. Hence, two cores are not sufficient to compensate for the extra cost – we only see speedups from three cores onward. The checking loop is so much slower here, since there are 20 reads for each write.

Spec-norm calculates the spectrum norm of a parameterized matrix with a matrix-vector product as the hot loop. We have included this benchmark to demonstrate that our technique is even applicable and achieves good performance on typical numeric problems. The loop in this benchmarks has 4865 reads for one write. Due to the normalization to the sequential execution time, the cost of pre-registration cannot be seen in the graph. Execution takes longer than inspection, because many computations are irrelevant for the dry-run.

3d-morph is a 3d graphics algorithm that does (slow) trigonometric computations in its hot loop. This is another worst-case benchmark as a static analysis could have detected the parallelizability. In contrast to Spec-norm, there are only write operations. Hence pre-registration and checking take about the same time.

4 Related Work

A number of parallel inspectors have been developed in the past that use a critical section to guard the book-keeping data structures, for example [9,16]. To check whether the availability of a highly optimized atomic machine instruction on today's multicore machines (like xchg, cmpxchg, ...) helps those types of inspectors, we have designed and optimized such a basic inspector that also only uses a tiny number of machine instructions for the book-keeping. Whenever this basic inspector sees a read or a write of an address, it upgrades the state of a corresponding slot in a `used`-array by conceptually calling the inlined `read`- or `write`-macro given below in pseudo-code.[5] Again, instead of the real addresses, we use hashed versions to make sure that the book-keeping array remains small.

```
const int NO_ACCESS = 0;
const int READ      = 1;
const int WRITE     = 2;
int used[]; //shared array, used by all inspector threads
void read(int addr_hsh) {
   if(xchg(&used[addr_hsh], READ) == WRITE)) alert();
}
void write(int addr_hsh) {
   if(xchg(&used[addr_hsh], WRITE) != NO_ACCESS)) alert();
}
```

This inspector is also very conservative as it even signals a dependence if an address is touched more than once from the same iteration and not just for reading. An extended version of this inspector that funnels the iteration number into the slots of the `used`-array to avoid an alert if an iteration reads and later writes to the same address was significantly slower.

Again, we added a generation counter to avoid re-initialization cost and found a way to do the `read`- and `write`-macros with just a single conditional jump. Both macros are very light-weight, except for the atomic `xchg` instruction, which dominates execution time due to the necessary synchronization. We feel that it would be hard to construct a faster inspector that relies on a synchronous access to a book-keeping data structure. Hence, this `xchg`-algorithm is the most competitive representative of this line of related work.

[5] An `xchg` machine instruction (InterlockedExchange on Windows) atomically writes the second argument into the address given by the first argument. It also returns the value that has been at that address before the assignment.

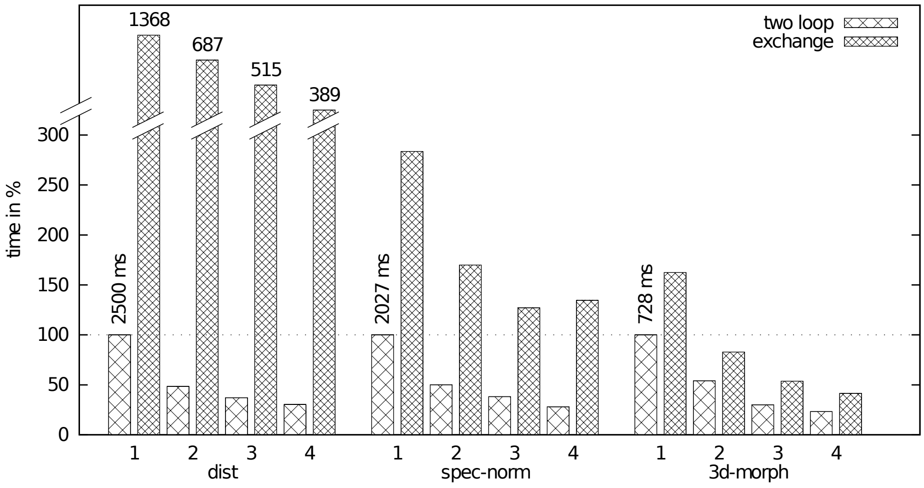


Fig. 2. Comparison to `xchg`-times. Times of the inspector only, taken on 1-4 cores. The runtime of the executor is not shown as it would be the same regardless of the type of inspector used.

```
int base = 0; //shared
void read(int addr_hsh) {
   if(xchg(&used[addr_hsh], READ) == base + WRITE)) alert();
}
void write(int addr_hsh) {
   if(xchg(&used[addr_hsh], WRITE) > base + NO_ACCESS)) alert();
}
```

These results shown in Fig. 2 demonstrate that even on today's multicore architectures such types of inspectors are (still?) not fast enough and that parallel inspectors need to be lock free, as our double inspection is. The benchmark em3d is missing in Fig. 2, since the `xchg`-algorithm is overly conservative, detects a loop independent flow dependence, and hence signals that the loop cannot be parallelized. On spec-norm the `xchg`-algorithm does not scale well (even on the 8 core chip). The bottleneck is the bus traffic needed to implement the locking.

Other inspectors try to actually compute wave fronts and deal with partially parallelizable loops, e.g. [11]. Leung and Zahorjan [8] have introduced the ideas of sectioning and bootstrapping and have demonstrated that these ideas help in speeding up parallel inspectors that compute wave fronts. But our double inspection deliberately stays away from computing wave fronts and just decides whether a loop is fully parallelizable or not. The reason is, that all the known wave front computing algorithms spend too much time in their book-keeping.

For a comparison, we have used sectioning to parallelize the wave front algorithm by Yang et al. [15]. This algorithm firstly is straightforward to parallelize, secondly requires fewer book-keeping data than other wave front algorithms, and finally can (in contrast to others) also handle all types of dependencies. We have

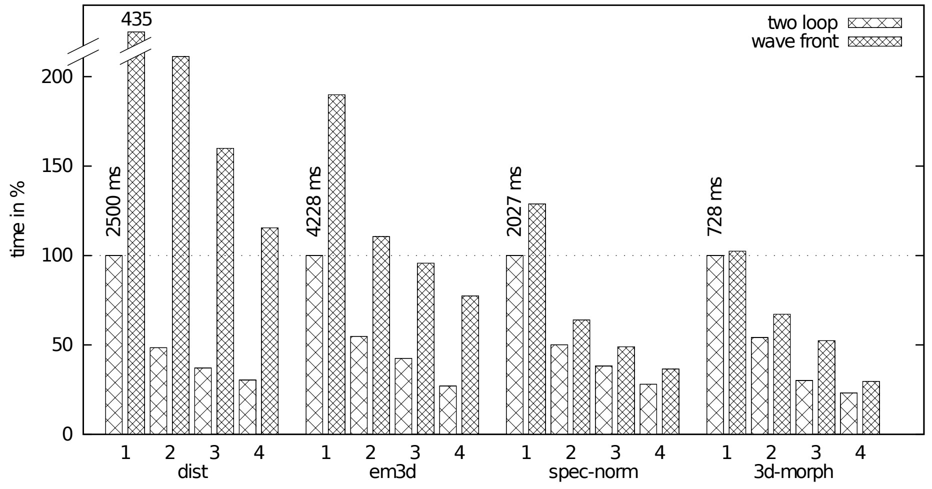


Fig. 3. Comparison to wave front algorithm. Times from taken on 1-4 cores.

optimized their algorithm by removing time consuming array dereferencing and indirection. This also allowed to avoid re-initialization of the book-keeping data structures. In addition, we have fixed a bug. The appendix gives the resulting pseudo-code of the fastest wave front algorithm we know.

Fig. 3 also shows that even with our optimizations, computing wave fronts is slow on those of our benchmarks that have indirect and more read accesses.[6] In addition to a slower inspection, execution suffers from extra synchronization barriers as the resulting wave fronts of all inspectors are concatenated. Even on fully parallelizable loops there is at least one synchronization barrier per thread. The result is that the time needed to compute the wave fronts cannot (easily) be amortized by parallel execution without schedule reuse or large loop bodies. Hence, our double inspection approach only checks for full parallelizability which can be done much faster and is universally applicable, even in dynamically typed codes that are hard to analyze for applicability of schedule reuse.

Other inspectors have used two phases before, e.g., [2,11]. In their first phase, the threads usually work on a private data structure. The second phase then merges the results from the first phase. Unfortunately, the merging phase is usually bounded by the sizes of the arrays, instead of being bounded by the complexity of the loop to be parallelized. For non-array codes that use heap objects, the merging phase would have a complexity that scales with the size of the main memory/the total address space. Moreover, the number p of threads often

[6] We are showing the best case for the wave front algorithm here. The code in the appendix can fail on certain loop-independent dependencies and is not as generally applicable as our double loop inspector. Moreover, for spec-norm and 3d-morph the access pattern is the best case for the wave front algorithm, as in contrast to the first two benchmarks, it causes almost no cache misses.

comes in as an additional factor (sometimes just as log p). Our double inspection is different because both its loops scale with the complexity of the original loop. The number of threads only affects the cost of the single synchronization barrier and does not come in as a multiplicative factor. Moreover, our book-keeping data structure does not need to be re-initialized between inspector runs. And it does not need to be scanned in its entirety in a merging phase.

Although our inspector is presented in this paper for use with a subsequent executor, it can also be used in speculative parallelism, as suggested in [4,12]. Since the book-keeping effort of our double inspection is tiny, it can easily be added to the speculative parallel execution without affecting its performance too much. To implement it, the speculative execution can either be piggy-backed to the pre-registration loop or to the checking loop of our inspector.

5 Conclusion

In this paper we have presented a fast parallel inspector that detects loops that can be executed in parallel. The novel idea of this inspector is that parallel threads inspect segments of the loop asynchronously. And instead of a costly merging phase, we have those threads inspect the loops again after a single synchronization barrier. The highly optimized implementation for the double inspection loops on current multicore architectures makes it possible to amortize the cost of inspection immediately – there is no longer any need for schedule reuse. On benchmarks we can amortize the inspection overhead and outperform the sequential version from 2 or 3 cores onward.

As a by-product, we also suggested optimizations for other types of inspectors that are known from the related work.

Future work should study the runtime overheads in a just-in-time engine. While one core could start executing a loop sequentially, the double inspection could be applied to the tail/the majority of the loop iterations. And if the tail turns out to be parallelizable, the JIT would switch. This could hide the inspection overhead. It will also be interesting to see if the technique can be extended to perform task-level parallelization of function calls or other code regions.

Acknowledgements. We thank Wolfram Schulte and Sebastian Burckhardt for many discussions and insights on contemporary multicore architectures, and especially Erez Petrank for an earlier investigation of the problem space.

Appendix: A Parallel Wave Front Inspector without any Shared Data Structures

With sectioning, we split the iteration space of a given loop into sections, each of which is handled by a single inspector thread that uses the algorithm of [15] with our enhancement and the bug fix. The resulting wave fronts of all threads are concatenated for execution. Fully parallelizable loops that are inspected and executed with n threads need n synchronization barriers.

The arrays `def` and `use` store in which wave front a value has been written/read before.[7] The wave front of an iteration is 1 plus the maximum of all the wave fronts of all the addresses that the iteration touches. This maximum is computed by the `read` and `write` macros and is stored in `current_wf` after all addresses are touched. Before all the addresses that the iteration touches are checked again, `wf` is updated to keep the wave front of the current iteration.

A second pass through all the addresses updates `use` and `def` to reflect that the current iteration reads/writes them. The original algorithm sets the corresponding slot of the `use` array for every read to the current iteration number. This is a bug and can lead to wrong results. This update must happen, *iff* the wave front number corresponding to the `use` array entry is smaller than the wave front number of the current iteration, because when referring to the latest iteration that reads this address, the new execution order must be considered.

Per read and per write, this algorithm needs about twice as many machine instructions than our double inspection. And since each of the inspector threads only works on its private data structures there is no need to synchronize. The downside is that itis far from optimal to concatenate all the thread-local wave fronts. The executer pays for this.

```
int use[];         //private per inspector thread
int def[];         //private
int wf[];          //private
int base_wf;       //of inspection
int current_wf;    //of iteration
int max_wf;        //of inspection
void begin_iteration() {
    current_wf = base_wf;
}
void read(int addr_hsh) {
    current_wf = max(current_wf, def[addr_hsh]);
}
void write(int addr_hsh) {
    current_wf = max(current_wf, def[addr_hsh], use[addr_hsh]);
}
void between(int iteration){
    wf[iteration] = ++current_wf;
    max_wf = max(max_wf, current_wf);
}
void read_update(int addr_hsh){
     use[addr_hsh] = max(use[addr_hsh], current_wf);
}
void write_update(int addr_hsh){
    def[addr_hsh] = current_wf;
}
```

Due to lack of space, we cannot show an extension of this algorithm that reuses its book-keeping data structures instead of re-initializing it for each inspection.

[7] The original algorithm stored the iterations numbers in those arrays and used the `wf` array to look up the wave fronts for the iterations. We avoid this indirection.

Again, a generation number can be funneled into the values that are written to the arrays.

References

1. Bebenita, M., Brandner, F., Fahndrich, M., Logozzo, F., Schulte, W., Tillmann, N., Venter, H.: SPUR: a trace-based JIT compiler for CIL. In: Proc. OOPSLA 2010, ACM Intl. Conf. Object-Oriented Programming, Systems, Languages, and Applications, Reno, NV, pp. 708–725 (October 2010)
2. Chen, D.K., Torellas, J., Yew, P.C.: An efficient algorithm for the run-time parallelization of DOACROSS loops. In: Proc. ACM/IEEE Conf. Supercomp., Washington, DC, pp. 518–527 (November 1994)
3. Eich, B.: JavaScript at ten years. In: ACM SIGPLAN Intl. Conf. Functional Programming, keynote. Tallinn, Estonia (September 2005), http://www.mozilla.org/js/language/ICFP-Keynote.ppt
4. Gupta, M., Nim, R.: Techniques for speculative run-time parallelization of loops. In: Proc. ACM/IEEE Conf. Supercomp., Melbourne, Australia, pp. 1–12 (July 1998)
5. Harris, T., Fraser, K.: Language support for lightweight transactions. In: Proc. OOPSLA 2003, ACM Intl. Conf. Object-Oriented Programming, Systems, Languages, and Applications, Anaheim, CA, pp. 388–402 (October 2003)
6. Kao, S.H., Yang, C.T., Tseng, S.S.: Run-time parallelization for loops. In: Proc. HICSS 1996, Hawaii Intl. Conf. System Sciences, Wailea, HI, vol. 1, pp. 233–242 (January 1996)
7. Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., Chew, L.P.: Optimistic parallelism requires abstractions. Comm. ACM 52(9), 89–97 (2009)
8. Leung, S.T., Zahorjan, J.: Improving the performance of runtime parallelization. In: Prof. PPoPP 1993, ACM SIGPLAN Symp. Principles and Practice of Parallel Programming, San Diego, CA, pp. 83–91 (May 1993)
9. Midkiff, S.P., Padua, D.A.: Compiler algorithms for synchronization. IEEE Trans. Comput. 36(12), 1485–1495 (1987)
10. Ponnusamy, R., Saltz, J., Choudhary, A.: Runtime compilation techniques for data partitioning and communication schedule reuse. In: Proc. ACM/IEEE Conf. Supercomp., Portland, OR, pp. 361–370 (November 1993)
11. Rauchwerger, L., Amato, N.M., Padua, D.A.: A scalable method for run-time loop parallelization. Intl. J. Parallel Programming 26(6), 537–576 (1995)
12. Rauchwerger, L., Padua, D.A.: The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. IEEE Trans. Parallel and Distrib. Systems 10(2), 160–180 (1999)
13. Saltz, J.H., Mirchandaney, R., Crowley, K.: Run-time parallelization and scheduling of loops. IEEE Trans. Comput. 40(5), 603–612 (1991)
14. Steffan, J.G., Colohan, C.B., Zhai, A., Mowry, T.C.: A scalable approach to thread-level speculation. In: Proc. Intl. Symp. Computer Architecture, Vancouver, Canada, pp. 1–12 (June 2000)
15. Yang, C.T., Tseng, S.S., Kao, S.H., Hsieh, M.H., Jiang, M.F.: Run-time parallelization for partially parallel loops. In: Proc. Intl. Conf. Parallel and Distrib. Systems, Seoul, South Korea, pp. 308–313 (December 1997)
16. Zhu, C.Q., Yew, P.C.: A scheme to enforce data dependence on large multiprocessor systems. IEEE Trans. Softw. Eng. 13(6), 726–739 (1987)

A Hybrid Approach to Proving Memory Reference Monotonicity

Cosmin E. Oancea and Lawrence Rauchwerger

PARASOL Lab., Texas A & M University, College Station, 77843, USA
{coancea,rwerger}@cse.tamu.edu

Abstract. Array references indexed by non-linear expressions or subscript arrays represent a major obstacle to compiler analysis and to automatic parallelization. Most previous proposed solutions either enhance the static analysis repertoire to recognize more patterns, to infer array-value properties, and to refine the mathematical support, or apply expensive run time analysis of memory reference traces to disambiguate these accesses. This paper presents an automated solution based on static construction of access summaries, in which the reference non-linearity problem can be solved for a large number of reference patterns by extracting arbitrarily-shaped predicates that can (in)validate the *reference monotonicity* property and thus (dis)prove loop independence. Experiments on six benchmarks show that our general technique for dynamic validation of the monotonicity property can cover a large class of codes, incurs minimal run-time overhead and obtains good speedups.

Keywords: monotonicity, access summary, autoparallelization.

1 Introduction

The emergence of multi-core systems as mainstream technology has brought automatic program parallelization back to the forefront. Classical dependence analysis, based on distance/direction vectors [1,2], or on algorithms to solve exactly a system of integer (in)equations [11,20], has achieved somewhat limited success for the class of small loop nests with linear access patterns.

For larger codes, several studies [4,16,19] have outlined the necessity of interprocedural data-flow analysis that: (i) summarizes array accesses to overcome array reshaping at call sites and to reduce the number of dependency tests, (ii) exploits control flow to either improve summary precision or to predicate optimistic results of statically undecidable access summaries, and that (iii) is capable of disambiguating non-linear array accesses, which fall outside Presburger arithmetic [10]. These issues were first investigated in Fortran codes, however the solutions that have been developed also apply to other languages, such as C++ and Java. In this paper we will present in some detail the issue of *non-linear accesses*, such as those using nonlinear indexing or subscript (index) arrays. These typically appear in programs with sparse data structures using index arrays and/or are an artifact of compiler transformations, such as induction variable substitution and/or multi-dimensional array reshaping at call sites, where merging summaries requires flattening the original array. Historically there have been two main approaches to the analysis of such non-linear, irregular memory reference patterns:

(a) *Static analysis* methods which attempt to prove loop independence at compile time. Solutions either (i) extend the library of recognizable access patterns and apply interprocedural inference of index-array value properties [14], or (ii) enhance the mathematical support with more refined symbolic ranges [6,9], or more encompassing algebras, such as chains of recurrence [8], or extensions of Presburger arithmetic [21]. If the analysis fails, then the user may be asked to examine parallelism based on the irreducible result of the corresponding algebra [16,21]. Static techniques are efficient (no overhead) but conservative and often ineffective in detecting parallelism.
(b) *Run-time Analysis* techniques which analyze the code memory references during program execution and decide if an optimization (e.g., parallelization) can be applied. Notable examples are the TLS (thread-level speculation) [22] and inspector/executor [23] techniques, which analyze dynamically memory reference traces to detect data dependencies. Run-time techniques are effective because they can extract most available parallelism, but exhibit significant overhead.

In this paper we present an automated solution rooted at a mid-point between the two classical directions: *Static* interprocedural analysis builds memory access summaries and extracts arbitrarily-shaped predicates, which can qualify loops as independent. When these predicates are too complex for the current symbolic analysis available to us then their evaluation is deferred until program *execution-time* where results are exact but overhead may be costly. Our technique represents a unified framework to optimize the tradeoff between static and dynamic analysis. We always prefer compile time analysis but we will complement it with minimal run-time analysis in order to successfully parallelize programs.

We have developed several techniques that can specialize the parallelization predicates in the most efficient manner. Instead of trying to extract exact conditions (necessary and sufficient) conditions for loop independence we generate predicates that represent only sufficient conditions for parallelization. The main contribution of this paper is a technique to extract predicates that can validate the assertion that memory references take strictly monotonic values (and are therefore independent). The motivation behind it is that in practice this specialized condition is easier to formulate than the general independence assertion and that in practice monotonic references are quite frequent.

For example, with the loop nest:

```
DO i = 1, N, 1
  DO j = 1, i, 1
    DO l = 1, 1 + i - j, 1
       XIJRSO(j + l - 1 + ($i^2$ - i)*M) = ...
 ENDDO ENDDO ENDDO
```

$$\textit{Output Independence}: \cup_{i=1}^{M}(S_i \;\cap\; \cup_{k=1}^{i-1} S_k) \;=\; \emptyset$$
$$\textit{where } S_i = [M*(i^2-i)+1,\; M*(i^2-i)+i]$$

an overestimate of the set of written locations of array XIJRSO in iteration i is interval $S_i = [M*(i^2-i)+1,\; M*(i^2-i)+i]$. Loop output independence requires that the writes of iteration i do not overlap with the writes of any iteration preceding i, which is formalized via equation $\cup_{i=1}^{N}(S_i \cap \cup_{k=1}^{i-1} S_k) = \emptyset$, where $\cup$ and $\cap$ denote set union and intersection, respectively. One can observe that the above (data-flow) equation is satisfied if S_i is strictly monotonic – i.e. the upper bound of S_i is less than the lower bound of the iteration $i+1$ summary S_{i+1}, for any i. This leads to the output independence predicate: $2 * M \geq 1$, which is relatively easier to formulate and verify at run-time.

The extracted predicates, including those testing monotonicity, are then (i) optimized via common mathematical support, (ii) factored into a sequence of sufficient loop-independence conditions ordered by their estimated run-time complexity, and (iii) evaluated at run time until one (possibly) succeeds. In practice we were able to often extract predicates of complexity $O(1)$ and $O(N)$ for polynomial and array indexing, respectively, which compares quite favorably (relative to overhead) to previous run-time techniques.

An evaluation on six PERFECT Club benchmarks [3] validates our statements by showing negligible run-time overhead for the predicate evaluation, as well as speed-ups between 1.54x and 3.72x, with an average of 2.24x on a commodity four-core system.

2 Preliminary Concepts

Our framework for analyzing non-linear accesses has two main stages: First, interprocedural data-flow analysis constructs an *exact* summary, named unified set reference (USR), of the read-only (RO), read-write (RW) and write-first (WF) array accesses. Loop independence is reduced to proving that an USR is empty. This stage has been named *hybrid analysis* [24], since run-time USR evaluation can be seen as a continuation of static analysis.

The second stage applies a top-down $USR = \emptyset$ factoring algorithm to extract sufficient predicates for parallelization, which are tested at run-time in the order of their complexity. If they all fail, an exact answer can be obtained via thread-level speculation – the LRPD test [7,22] – or via USR run-time evaluation. This part has been named *sensitivity analysis* [25] because it models how parallelism depends on statically unavailable input parameters. Sections 2.1 and 2.2 give the gist of the two stages and the information needed to understand this paper.

2.1 Hybrid Analysis

Hybrid Analysis builds on *linear-memory access descriptors* (LMAD) [19], which are denoted $[\delta_1, .., \delta_P] \vee [\sigma_1, .., \sigma_P] + \tau$ and represent the set of points: $\{\tau + i_1 * \delta_1 + .. + i_P * \delta_P \mid 0 \leq i_k * \delta_k \leq \sigma_k, \forall k \in 1..P\}$. Being uni-dimensional, LMADs allow reshaping at call sites where the dimensions of the formal and actual parameter differ, but strides δ_k and spans σ_k also model "virtual" multi-dimensional accesses. Summaries are constructed by traversing the call and control dependency (CDG) graphs in reverse topological order, while within a CDG region nodes are traversed in program order. During this bottom-up parse, data-flow equations dictate how summaries are initialized at statement level, merged across branches, translated

```
SUMMARIZE($REG_1$; $REG_2$)
  $(WF_1, RO_1, RW_1) \leftarrow REG_1$
  $(WF_2, RO_2, RW_2) \leftarrow REG_2$
  $WF = WF_1 \cup (WF_2 - (RO_1 \cup RW_1))$
  $RO = (RO_1 - (WF_2 \cup RW_2)) \cup$
        $(RO_2 - (WF_1 \cup RW_1))$
  $RW = RW_1 \cup (RW_2 - WF_1) \cup$
        $(RO_1 \cap WF_2)$
  RETURN $(WF, RO, RW)$
```

(a) Consecutive Region Composition

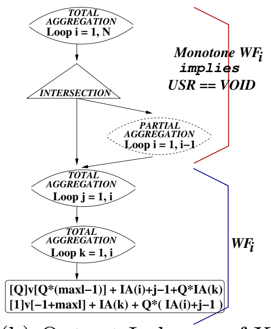
```
SUMMARIZE($REG_i$, $i = 1, N$)
  $(WF_i, RO_i, RW_i) \leftarrow REG_i$
  $WF = \bigcup_{i=1}^{N}(WF_i - \bigcup_{k=1}^{i-1}(RO_k \cup RW_k))$
  $RO = \bigcup_{i=1}^{N} RO_i - \bigcup_{i=1}^{N}(WF_i \cup RW_i)$
  $RW = \bigcup_{i=1}^{N}(RO_i \cup RW_i) - (WF \cup RO)$
  RETURN $(WF, RO, RW)$
```

(b) Loop Summary Aggregation

Fig. 1. Building WF, RO and RW summaries across consecutive regions and loops

```
REAL X(Q, *)
DO i = 1, N, 1
   DO j = 1, i, 1
     a = 1.0D+00/(i+j)
     DO k = 1, i, 1
       maxl = k
       IF (k+(-i).EQ.0) maxl = j
       DO l = 1, maxl, 1
         X(IA(i)+j, IA(k)+l) = a + 1.0/(k+l)
         X(IA(k)+l, IA(i)+j) = a + 1.0/(k+l)
       ENDDO
     ENDDO ENDDO ENDDO
```

(a) Simplified INTGRL_do140 Code


TOTAL AGGREGATION Loop i = 1, N
Monotone WF_i
implies
USR == VOID
INTERSECTION
PARTIAL AGGREGATION Loop i = 1, i–1
TOTAL AGGREGATION Loop j = 1, i
TOTAL AGGREGATION Loop k = 1, i
WF_i
[Q]v[Q*(maxl–1)] + IA(i)+j–1+Q*IA(k)
[1]v[–1+maxl] + IA(k) + Q*(IA(i)+j–1)

(b) Output-Ind. USR of X

Fig. 2. Code and Output-Independence USR for `TRFD`'s Loop `INTGRL_do140`

across call sites, composed between consecutive regions, and aggregated across loops. The latter two cases are illustrated in Figures 1(a) and 1(b). For example, the composition of a read-only region S_1 with a write-first region S_2 gives $RO = S_1 - S_2$, $RW = S1 \cap S2$ and $WF = S_2 - S_1$.

We employ an *exact* (DAG) representation, named USR, in which leafs are sets of LMADs and internal nodes are operations that cannot be precisely expressed in the LMAD domain. Internal nodes can represent unsimplifiable *set operations* ($\cup$, $\cap$, $-$), or control flow: *gates* predicating LMAD's existence, UN-translatable *call sites*, or *loops* that fail exact aggregation. As such, USRs represent memory references at a program level, in a scoped and closed-under-composition language. A loop of iteration summary (WF_i, RO_i, RW_i) has no flow/anti dependencies *iff*:

$$\{(\cup_{i=1}^{N} WF_i) \cap (\cup_{i=1}^{N} RO_i)\} \;\cup\; \{(\cup_{i=1}^{N} WF_i) \cap (\cup_{i=1}^{N} RW_i)\} \;\cup$$
$$\{(\cup_{i=1}^{N} RO_i) \cap (\cup_{i=1}^{N} RW_i)\} \;\cup\; \{\cup_{i=1}^{N}(RW_i \cap (\cup_{k=1}^{i-1} RW_k))\} = \emptyset.$$

Similarly, output independence can be modeled via equation:

$$\{\cup_{i=1}^{N}(WF_i \cap (\cup_{k=1}^{i-1} WF_k))\} = \emptyset.$$

The privatization and reduction parallelization transformations are also supported but are outside the scope of this paper.

Figure 2(a) and 2(b) show loop `INTGRL_do140` of the `trfd` benchmark and the associated output independence USR for array `X`. The WF_i USR is shown in the bottom part of the figure – loop nodes are introduced because exact LMAD

```
SG **FACTOR**($D$ : USR)
  // ***Output:*** $P$ s.t. $P \Rightarrow (D = \emptyset)$
  Case $D$ of:
    $q\#A$:      $P = \overline{q} \vee$ **FACTOR**($A$)
    $A \cup B$:  $P =$ **FACTOR**($A$) $\wedge$ **FACTOR**($B$)
    $A - B$:     $P =$ **FACTOR**($A$) $\vee$ **INCLUDED**($A$,$B$)
    $A \cap B$:  $P =$ **FACTOR**($A$) $\vee$ **FACTOR**($B$)
                         $\vee$ **DISJOINT**($A$,$B$)
    $\bigcup_{i=1}^{N}(A_i)$: $P = \bigwedge_{i=1}^{N}$ **FACTOR**($A_i$)
    $A \bowtie CallSite$: $P =$**FACTOR**($A$)$\bowtie CallSite$

SG **DISJOINT_APPROX**($C$ : USR, $D$ : USR)
  $(P_C, \lceil C \rceil) \leftarrow$ conditional LMAD overestimate of$A$
  $(P_D, \lceil D \rceil) \leftarrow$ conditional LMAD overestimate of$D$
  $P = P_C \vee P_D \vee$ **DISJOINT_LMAD**($\lceil C \rceil$, $\lceil D \rceil$)
```

(a) Extracting Predicates from $USR = \emptyset$

```
SG **DISJOINT**($C$ : USR, $D$ : USR)
  // ***Output:*** $P$ s.t. $P \Rightarrow (C \cap D = \emptyset)$
  $P =$ **DISJOINT_HALF**($C$, $D$) $\vee$
       **DISJOINT_HALF**($D$, $C$) $\vee$
       **DISJOINT_APPROX**($C$, $D$)

SG **DISJOINT_HALF**($C$:USR,$D$:USR)
  Case $C$ of:
    $q\#A$:      $P = \overline{q} \vee$ **DISJOINT**($A$, $D$)
    $A \cup B$:  $P =$ **DISJOINT**($A$, $D$) $\wedge$
                      **DISJOINT**($B$, $D$)
    $A - B$:     $P =$ **DISJOINT**($A$, $D$) $\vee$
                      **INCLUDED**($D$,$B$)
    $A \cap B$:  $P =$ **DISJOINT**($A$, $D$) $\vee$
                      **DISJOINT**($B$, $D$)
```

(b) Functions used by FACTOR

Fig. 3. Construction of an Arbitrarily-Shaped Independence Predicate

aggregation fails over loop `k` due to the non-linear use of `IA(k)`. The output independence pattern is explicit in the upper part of the figure, where partial aggregation corresponds to the $\cup_{k=1}^{i-1} WF_k$ term. Even if LMAD aggregation would have succeeded for loop `k`, it would still fail partial aggregation due to `IA(i)`.

2.2 Sensitivity Analysis

Figure 3(a) and 3(b) give the gist of the top-down `FACTOR` algorithm. Inference on set-algebra properties guides a recursive construction of an arbitrary complex predicate. Its representation, named *sensitivity graph* (SG), similar to USR, is a DAG in which leafs contain predicate expression, while nodes represent either (i) logical operators – *or* ($\vee$) , *and* ($\wedge$) – or (ii) control flow – call sites, loops – across which predicates cannot be summarized.

For example, from $q\#A = \emptyset$, where q denotes the branch condition under which A is defined, we extract the predicate $\overline{q} \vee$ `FACTOR`(A). Sufficient conditions for $A \cap B = \emptyset$ are $A = \emptyset$ or $B = \emptyset$. Additional predicates for this equation are inferred in `DISJOINT_HALF` by examining the shape of A and B.

Whenever we reach LMAD leafs or when we encounter call site or loop nodes (that end the `DISJOINT`-based logical inference), `DISJOINT_APPROX` conservatively flattens the problem to the LMAD domain. We have found most useful to represent an overestimate of USR D as a pair $(P_D, \lceil D \rceil)$, where P_D is a predicate under which D is empty, while $\lceil D \rceil$ is an LMAD-overestimate of D. Building on the LMAD intersection and subtraction algorithms [19], `DISJOINT_LMAD` extracts the conditions for (i) valid projection on number-of-strides dimensions and for (ii) the corresponding pairs of resulting 1D-LMADs to be disjoint.

The result predicate is brought to disjunctive normal form in which terms are sorted in increasing order of their estimated complexity. Code is automatically generated to cascade these tests, and to implement conditional loop parallelization. Redundancy is optimized by hoisting calls to these tests inter-procedurally at the highest dominator point where all the input values are available.

3 Extracting LMAD-Monotonicity Predicates

Section 3.1 answers the question "where and when is monotonicity tested?" Since our solution to non-linear accesses is intrinsically related to LMAD summarization, Section 3.2 establishes a uniform notation and discusses how LMADs are aggregated across loops. Our strategy applies an incremental effort to proving monotonicity: Section 3.3 presents a simple test to handle one quasi-linear LMAD, and Sections 3.4 and 3.5 treat the uni and multi-dimensional cases of non-linear LMADs with polynomial and array indexing, respectively. Finally, Section 3.6 discusses the overall design of the monotonicity test, and possible extensions.

3.1 When to Apply Monotonicity Tests?

We try to extract monotonicity predicates *wherever* the `FACTOR` algorithm encounters the equation $\cup_{i=1}^{N}(A_i \cap (\cup_{k=1}^{i-1} A_k)) = \emptyset$, for an arbitrary USR A_i. One can observe that if an overestimate of A_i, named $\lceil A_i \rceil$, is strictly monotonous, under some definition of set order ">", then the above equation holds, i.e. if $\lceil A_i \rceil > \lceil A_{i-1} \rceil$ for any $2 \leq i \leq N$, then by induction $\lceil A_i \rceil > \lceil A_j \rceil$, for any $1 \leq j \leq i-1$, and hence the intersection $A_i \cap (\cup_{k=1}^{i-1} A_k)$ is empty for any i. We also check the monotonicity of $RO_i \cup WF_i \cup RW_i$ since this is a sufficient condition for the independence of the loop of index `i`. The answer to the *where* question is thus pattern matching at USR level; this is very different from solutions based on code pattern recognition in that our matching is less a consequence of the problem and more an artifact of the proof, i.e. irreducible data-flow equations.

We try to extract monotonicity predicates *whenever* an LMAD overestimate of A_i, named L_i, can be computed, but the aggregation of L_i over the loop of index i is either inexact or fails in the LMAD domain. The latter is the sign of an irregular access, either non-linear: L_i exhibits subscripted or polynomial or exponential indexing, or quasi-linear: the same index appears in two LMAD dimensions, or division operations occlude linearity. Either way, `FACTOR` is likely to fail. For example, assuming an LMAD overestimate L_i exists for WF_i in Figure 2(b), L_i would fail partial aggregation over the outermost loop of index `i` due to the indirect access `IA(i)`. `FACTOR` can only try to prove that WF_i is empty but since this is not the case the result will be the predicate `false`.

3.2 Problem Statement, Notation and LMAD Loop Aggregation

The problem addressed is extracting predicates that are sufficient conditions for the satisfiability of the USR equation $X = \cup_{i=1}^{N}(D_i \cap (\cup_{k=1}^{i-1} D_k)) = \emptyset$, where $i = 1, N$ is the range of normalized loop L. We denote with $A_i = \{A_i^1, A_i^2, ..., A_i^M\}$ a list of LMADs that overestimates D_i (as a set of points). We recall that:

$E = [\delta_1, \delta_2, .., \delta_P] \vee [\sigma_1, \sigma_2, .., \sigma_P] + \tau$ denotes a P-dimensional LMAD representing the set of points: $\{i_1 * \delta_1 + i_2 * \delta_2 + ... + i_P * \delta_P + \tau \mid 0 \leq i_k * \delta_k \leq \sigma_k, 1 \leq k \leq P\}$, where δ_k and σ_k are called the stride and span of dimension k, respectively.

If for any k, $\delta_k > 0$, then interval $[\tau, \tau + \sum_{j=1}^{P}(\sigma_j)]$ overestimates E. A_{i+1}^k, τ_{i+1} are obtained by replacing i with $i+1$ in A_i^k and τ_i.

To compute an LMAD overestimate $\lceil X \rceil$ of an USR X we apply a recursively defined operator on the USR domain. In the *top down* parse the operator disregards node B in terms such as $C - B$, $C \cap B$, while in the return (*bottom-up*) parse it translates, aggregates and adds (union) the encountered LMAD leafs over call site, loop and $\cup$ nodes, respectively. At the LMAD level, aggregation over loop L succeeds as long as the resulting strides/span/offset are loop invariant (since the USR language is scoped). The remainder of this section demonstrates at an intuitive level how LMAD's *overestimate* and *exact* aggregation works.

```
      DO i = 1, N                    DO i = 1, N
        DO j = 1, N                    DO j = 1, i
S1 :      Y(j+i*N) = ...                 Y(j+i*N) = ...
      ENDDO ENDDO                    ENDDO ENDDO
```

With both loop nests above, the LMAD describing the write access of Y at statement S_1 is the LMAD point $(j-1)+N*(i-1)$. Aggregation over the *left-hand side loop* of index j creates a new dimension of stride $\delta_{new} = \tau_{j+1} - \tau_j = 1$ and span $\sigma_{new} = \tau_{j \leftarrow N} - \tau_{j \leftarrow 1} = N-1$, since $j \in [1, N]$, giving LMAD $[1] \vee [N-1] + N*(i-1)$. Aggregation over loop i similarly creates a new dimension of stride N and span $N^2 - N$, giving $[1, N] \vee [N-1, N^2 - N] + 0$. The left-hand side loop nest aggregation has been exact. Considering the *right-hand side loop* nest, aggregation over loop j gives $[1] \vee [i-1] + N*(i-1)$, since $j \in [1, i]$, and over loop i gives $[1, N] \vee [i-1, N^2 - N] + 0$. Since the USR language is scoped, exact aggregation over loop i fails, because loop-variant symbol i still appears in the LMAD. However, an overestimate can be computed by replacing i with its upper bound N, which gives LMAD $[1, N] \vee [N-1, N^2 - N] + 0$.

Intuitively, the strategy for testing monotonicity is to overestimate each LMAD dimension via intervals, and to formulate several kinds of monotonicity which all reduce to studying interval monotonicity; for example increasing monotonicity would correspond to the upper bound of the interval corresponding to iteration i being less than the lower bound of the interval corresponding to iteration $i+1$.

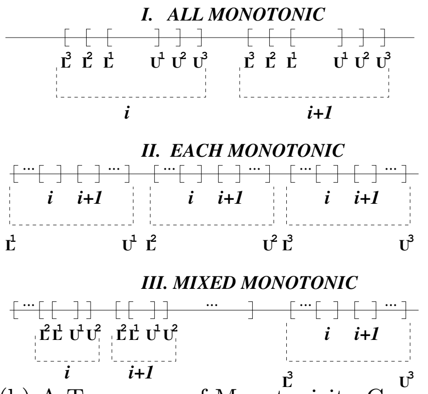
3.3 Quasi-Linear Case

We consider the case when the overestimate of USR D_i is *one* LMAD A_i. Observe that if overestimate L-aggregation of A_i succeeds and creates a new dimension, then necessarily the new stride is L invariant, and hence the access is strictly monotonic in the new dimension. Denoting with A the aggregated LMAD, a sufficient condition for the data-flow equation $X = \emptyset$ to hold is that A has non-zero strides and its "virtual" dimensions do not overlap. Checking non-overlap requires normalizing A – all strides/spans are made positive and dimensions are sorted according to strides – and checking that $\sum_{j=1}^{k-1} \sigma_j < \delta_k$, for any $2 \leq k \leq P$. When the values of δ_k, σ_k are not all known statically, an non-overlap predicate of complexity $O(1)$ is extracted and evaluated at run time.

While this test primarily handles quasi-linear access in which i appears in two dimensions, it also covers some non-linear cases. For example access $Y(j+N^2*i)$ in a two-level loops nest of indexes $i = 1, N$ and $j = 1, i^2$ gives after j-aggregation

```
SE REDUCE_GT_0( expr )
  //Input:  an int-type expression
  //Output: P s.t. P ⇒ (expr > 0)
    (a, b, i, L, U, err) =
          FIND_SYMBOL(expr);
    // expr = a*i+b, L ≤ i ≤ U, i ∉ b
// P = (a ≥ 0 ∧ a*L+b>0) ∨ (a < 0 ∧ a*U+b>0)
    IF ( err ) THEN RETURN (expr > 0);
    ELSE RETURN
              [ REDUCE_GT_0(a+1) ∧
                REDUCE_GT_0(a*L+b) ] ∨
              [ REDUCE_GT_0(−a)  ∧
                REDUCE_GT_0(a*U+b) ];
```

(a) Symbolic Fourier-Motzkin Elimination



(b) A Taxonomy of Monotonicity Cases

Fig. 4.

$[1] \vee [i^2 - 1] + N^2 * (i - 1)$. Note that σ_1 is quadratic in i, but this does not impede i-aggregation because $\tau = (i-1) * N^2$ is linear in i and an upper bound for i^2 is known (N^2) leading to $[1, N^2] \vee [N^2 - 1, N^3 - N^2]$ which never overlaps.

Loop `FTRVMT_do109` of the `ocean` benchmark was solved via such an $O(1)$ non-overlap test, where the L-aggregated LMAD has four symbolic dimensions.

3.4 Nonlinear, Unidimensional (1D) LMAD Case

We consider the case when the per-iteration summary is overestimated via a list of 1D LMADs $A_i = \{A_i^1 = [\delta_i^1] \vee [\sigma_i^1] + \tau_i^1, .., A_i^M = [\delta_i^M] \vee [\sigma_i^M] + \tau_i^M\}$, where all M strides and spans are positive. The intuitive idea is to overestimate each A_i^k to an interval $F_i^k = [\tau_i^k, \tau_i^k + \sigma_i^k]$ and to study monotonicity at interval level, where we denote the lower and upper bounds of F_i^k by $L_i^k = \tau_i^k$ and $U_i^k = \tau_i^k + \sigma_i^k$.

Figure 4(b) depicts three main cases: In the *all monotonic case*, overestimating the whole per-iteration summary A_i to an interval still forms a strictly monotonic sequence of intervals in i. This guarantees that the accesses of iteration i do not overlap with those of iteration $i+1$, for any i, and by induction, with any of an iteration other than i. More formally, predicate $P_{all\_mon} =$

$$[\wedge_{i=1}^{N-1}(\texttt{MAX}_{k=1}^{M}(U_i^k) < \texttt{MIN}_{k=1}^{M}(L_{i+1}^k))] \quad \vee \quad [\wedge_{i=1}^{N-1}(\texttt{MAX}_{k=1}^{M}(U_{i+1}^k) < \texttt{MIN}_{k=1}^{M}(L_i^k))]$$

satisfies our data-flow equation $X = \emptyset$, where the first and second terms are the conditions for strictly increasing and decreasing monotonicity, respectively.

In the *each monotonic case*, each LMAD in list A_i has the property that its accesses form a strictly monotonic, hence non-overlapping, sequence in the iteration space: i.e. the interval sequence F_i^k is strictly monotonic in i. Additionally, maximizing the M intervals over the range of i gives M pairwise disjoint intervals F^k. This is a sufficient condition that satisfies equation $X = \emptyset$. Formally, the predicate for the first condition is $PRED_{each\_mon} =$

$$\begin{aligned}&[\wedge_{i=1}^{N-1}(U_i^1 < L_{i+1}^1) \ \vee \ \wedge_{i=1}^{N-1}(U_{i+1}^1 < L_i^1)] \wedge ... \wedge \\ &[\wedge_{i=1}^{N-1}(U_i^M < L_{i+1}^M) \ \vee \ \wedge_{i=1}^{N-1}(U_{i+1}^M < L_i^M)].\end{aligned}$$

The "additional" part requires computing the lower and upper bounds for each interval F^k. We use the assumed monotonicity of F_i^k to exactly express the lower and upper bounds of F^k: $L^k = \mathrm{MIN}(L_1^k, L_N^k)$ and $U^k = \mathrm{MAX}(U_1^k, U_N^k)$. We create a predicate $PRED_{sc}$ as a sensitivity graph node, named *sorted check*, whose run-time semantic is: sort the M intervals $[L^k, U^k]$ in increasing order of L^k and return $\wedge_{k=1}^{M-1}(U^k < L^{k+1})$. Thus $PRED_{each\_mon} \wedge PRED_{sc}$ is another sufficient condition that satisfies the loop-independence data-flow equation $X = \emptyset$.

Finally, in the *mixed monotonic* case, any combination of the previously discussed cases would satisfy $X = \emptyset$. This leads to too many possible solutions and hence we restrict our implementation to the first two "extreme" patterns.

While the predicates extracted so far will prove monotonicity at run-time, they can be significantly optimized via symbolic Fourier-Motzkin elimination of loop-variant symbols of known bounds, as depicted in Figure 4(a). In the *each monotonic* case, we apply it to eliminate L-variant symbols such as i from the $U_i^k < L_{i+1}^k$ and $U_{i+1}^k < L_i^k$ inequations. In the *all monotonic* case, we try to solve $O(M^2)$ inequations, the resulting predicate being:

$$P_{all\_mon\_opt} = \wedge_{1 \le k,j \le M}(U_i^j < L_{i+1}^k) \ \vee \wedge_{1 \le k,j \le M}(U_{i+1}^j < L_i^k).$$

Successful elimination gives a loop-invariant predicate of $O(1)$ run-time complexity for both cases. Otherwise, in the *all monotonic* case, we use the non-optimized predicate, which has $O(N * M)$, rather than $O(N * M^2)$ complexity.

Array `XIJRS0` from loop `OLDA_do300` of `trfd` benchmark exhibits a quadratic per-iteration summary formed by three LMADs ($M = 3$), that gives an $O(1)$ independence predicate equivalent to[1]: $\mathtt{morb}+\mathtt{morb}^2+(-1)*\mathtt{num}+(-1)*\mathtt{num}^2 \le 0$.

3.5 Nonlinear Multi-dimensional LMAD Case with Index Arrays

When A_i contains P-dimensional LMADs of similar shape – i.e. all strides match across $A_i^k, k \in [1, M]$, then we study monotonicity dimension wise. This requires to split τ across dimensions, and to extract a well-formedness predicate P_{wf}.

For example, LMAD $[1, Q] \vee [i-1, Q*(i-1)] + i + Q*(i+1)$ can be interpreted as a 2D array access that covers intervals $[i, 2*i-1]$ and $[i+1, 2*i]$ on the first and second direction, respectively. The decomposition is valid as long as dimensions do not overlap, hence P_{wf} is $2*i-1 < Q$, which, since $i \in [1, N]$, becomes $2*N-1 < Q$ (by Fourier Motzkin). τ splitting is implemented via a heuristic that we do not discuss here, but which may fail when there is not enough confidence that the guess is correct. When splitting fails we approximate an LMAD with the interval $[\tau, \tau+\sum_{j=1}^{P}\sigma_j]$ and apply the technique of Section 3.4.

Intuitively, with the *each monotonic case*, each LMAD monotonicity is established by showing monotonicity in *one* dimension. The *all monotonic case* is more complex because the generation of a sufficient condition for increasing monotonicity, requires that, for any LMAD, there exist at least one dimension in which its lower bound for iteration $i+1$ is greater than the iteration-i upper bounds of *all* LMADs. Formally, denoting by $[L_i^{t,j}, U_i^{t,j}]$ the interval overestimate of dimension t of A_i^j, and

[1] Constant propagation could determine $morb = num$ and the decision can be static.

$$\begin{aligned} P_{each\_mon} &= \wedge_{j=1}^{M}\{\ \vee_{t=1}^{P}[\ \wedge_{i=1}^{N-1}(U_i^{t,j} < L_{i+1}^{t,j})\ \vee\ \wedge_{i=1}^{N-1}(U_{i+1}^{t,j} < L_i^{t,j})\ ]\ \} \\ P_{all\_mon} &= \wedge_{j=1}^{M}\{\ \vee_{t=1}^{P}[\ \wedge_{i=1}^{N-1}(U_i^{t,1} < L_{i+1}^{t,j}\ \wedge ... \wedge\ U_i^{t,M} < L_{i+1}^{t,j})\ \vee \\ &\qquad\qquad\qquad \wedge_{i=1}^{N-1}(U_{i+1}^{t,1} < L_i^{t,j}\ \wedge ... \wedge\ U_{i+1}^{t,M} < L_i^{t,j})\qquad]\ \}, \end{aligned}$$

the *all monotonic* predicate is $P_{all\_mon} \wedge P_{wf}$ and the *each monotonic* one is $P_{each\_mon} \wedge P_{sc} \wedge P_{wf}$, where we optimize by Fourier-Motzkin all inequations.

To handle complex index-array cases, we refine implementation to allow conditional LMAD aggregation based on assumed monotonicity of the index array that would otherwise hinder aggregation.

We demonstrate the approach on loop `INTGRL_do140` of `trfd` benchmark, where, for clarity, we show only the monotonically *increasing* case. The code and $\texttt{WF}_i$ USR are shown in Figures 2(a) and 2(b). To compute A_i we need to aggregate: $B_k^j = \{\ [\texttt{Q}] \vee [\texttt{Q} * (\texttt{maxl} - 1)] + (\texttt{IA}(i) + j - 1) + \texttt{Q} * \texttt{IA}(k),$

$$[1] \vee [\texttt{maxl} - 1] \quad + \texttt{Q} * (\texttt{IA}(i) + j - 1) + \texttt{IA}(k) \qquad \}$$

over loops of indexes $k \in [1, i]$ and $j \in [1, i]$. We assume that `IA(k)` is monotonic in k, i.e. $P_{IAmon}^{i,j} = \wedge_{k=1}^{i-1}\texttt{IA}(k) < \texttt{IA}(k+1)$, and extend this predicate over the range of i, giving $P_{IAmon} = \wedge_{i=1}^{N-1}\texttt{IA}(i) < \texttt{IA}(i+1)$. Using IA's assumed monotonicity, the range of `IA(k)` in loop k becomes $[\texttt{IA}(1), \texttt{IA}(i)]$, and the range of `maxl` is $[1, i]$. Thus, overestimate aggregation over loops k and j succeeds:

$$\begin{aligned} A_i = \{\ & [1, \texttt{Q}] \vee [i-1, \texttt{Q} * (i - 1 + \texttt{IA}(i) - \texttt{IA}(1))] + \texttt{IA}(i) + \texttt{Q} * \texttt{IA}(1), \\ & [1, \texttt{Q}] \vee [i - 1 + \texttt{IA}(i) - \texttt{IA}(1)\ , \texttt{Q} * (i-1)] + \texttt{Q} * \texttt{IA}(i) + \texttt{IA}(1)\}. \end{aligned}$$

Projection on the first dimension gives intervals: $[\texttt{IA}(i), \texttt{IA}(i)+i-1]$, and $[\texttt{IA}(1), \texttt{IA}(i)+i-1]$; the second dimension is similar, but reversed. After simplification, the *all monotonicity* formula gives: $P_{all\_mon} = \wedge_{i=1}^{N}[\ \texttt{IA}(i+1) \geq \texttt{IA}(i) + i]$.

Finally, the well-formedness predicate that guarantees dimensions do not overlap is $P_{wf} = \texttt{IA}(N) + N - 1 < Q$, and implication-based reductions, such as $\texttt{IA}(i+1) \geq \texttt{IA}(i) + i \Rightarrow \texttt{IA}(i+1) > \texttt{IA}(i)$, optimizes away term P_{IAmon}. Overall, we obtain the output-independence predicate $P_{all\_mon} \wedge P_{wf}$, whose $O(N)$ runtime overhead is negligible against the $O(N^4)$ complexity of loop `INTGRL_do140`.

3.6 Overall Design and Possible Extensions

We try first the non-overlap test presented in Section 3.3 since it is the cheapest in terms of both compile and run time complexity – $O(1)$. If a predicate cannot be extracted this way, we study monotonicity in the more comprehensive form of Sections 3.4 and 3.5: If A_i contains LMADs of similar shape and dimension-wise projection succeeds, we apply the multi-dimensional test, otherwise LMADs are flattened and predicates are extracted via the $1D$ test. In practice we encountered only instances of the *all monotonic* cases.

Our solution evolves naturally: *First*, non-linearity is discovered as a consequence of exact aggregation failing on LMADs. *Second*, predicate extraction models a general form of monotonicity at interval level, and as such is transparent to the non-linearity shape, i.e., indirect access or polynomial indexing. *Finally*, the extracted predicates are aggressively optimized: (i) symbolic Fourier-Motzkin elimination is applied on inequations, (ii) the assumed monotonicity refines ranges and (iii) implication-based invariants effectively filter the sensitivity graph SG, e.g., if $A \Rightarrow B$ then $A \wedge B \equiv A$. Denoting by N the number of iterations of the outermost loop, we typically extract predicates of run-time complexity $O(1)$ for polynomial and $O(N)$ for array indexing, where the $O(N)$ predicate is evaluated in parallel, giving *scalable* performance.

4 Related Work

Hoeflinger's ART test [13,19] summarizes accesses via non-linear LMADs that allow strides to contain loop-variant symbols of the aggregated loop, such as the loop index. The LMAD corresponding to the independence test is aggregated across the to-be-analyzed loop and independence is tested by checking that LMAD dimensions do not overlap. We found that such aggressive aggregation is often too conservative. We diverge early in our design by keeping LMADs quasi linear, i.e., loop-invariant strides, and using USRs to allow exact summarization, and `FACTOR` to extract independence predicates. The test of Section 3.3 is similar with Hoeflinger's non-overlap test, the difference being that our LMAD algebra requires checking that aggregation creates a new dimension. While manual application of ART reported success on codes with exponential indexing, such as `tfft2`, that we do not cover, ART probably cannot handle the codes exhibiting index arrays and even quadratic indexing, such as `INTGRL_do140` and `OLDA_do300`. This would probably require an analysis similar to ours to prove loop-variant strides positive.

The test we described in Section 3.4 intuitively resembles Blume and Eigenmann's Range Test [5], which uses symbolic-range propagation and the monotonicity of a read/write pair of accesses to disprove non-independent direction vectors. The Range Test covers polynomial and some exponential indexing, but cannot handle index arrays (e.g., `INTGRL_do109`). Loop index appearing in different dimensions may also be a hindrance in some cases.

Lin and Padua give a code *pattern-matching* extension of the Range Test, named Offset-Length Test [14], that handles index-array accesses such as the ones in `INTGRL_do140`. The solution uses interprocedural analysis to verify invariants on the values stored in the index array and as such, to derive independence statically. While a static result is always desired, array properties could be impossible to establish statically, e.g., array read from a file. In contrast we extract a $O(N)$ predicate, whose run-time overhead is negligible given that the original loop has $O(N^4)$ complexity, and furthermore our predicate stems naturally from LMAD monotonicity rather than relying on code pattern matching.

Pugh and Wonnacott extend Presburger arithmetic with support for uninterpreted functions [21] to analyze dependencies of a pair of non-linear accesses (some control flow included as well). Wherever Presburger-like algebra cannot prove independence, the irreducible formula is presented to the user for validation. Several refinements, such as inductive simplification, extract simpler formulas that are sufficient independence conditions. The more complex index-array loop `INTGRL_do140` is not reported, and the quadratic indexed loops

Table 1. Loop and Benchmark Level Properties. The last three columns describe (i) the type of non-linearity: polynomial indexing (i^2), indirect array ($\texttt{IA}_{mon}$), or index spanning multiple dimensions (Non-Overlap), (ii) the complexity of the predicate that proves flow/output independence (F/O-IND), (iii) the loop contribution to sequential coverage (*weight*) and (iv) the predicate overhead as percentage of the parallel loop run time.

Loops with Non-Linear USRs from Several Perfect Benchmarks					
Benchmark	Bench Properties	Non-Linear Loop	Successful Pred Type	Weight	Pred OV
TRFD	SC = 98.5% RT-IW, PRIV, SLV, DLV	OLDA_do100 OLDA_do300 INTGRL_do140	O-IND, $O(1)$, i^2 F/O-IND, $O(1)$, i^2 O-IND, $O(N)$, IA_{mon}	68.2% 26.8% 3.4%	0.003% 0.015% 0.510%
DYFESM	SC = 96.5% RT-IW, PRIV, RED, RT-RED	MXMULT_do10 SOLXDD_do10 SOLVH_do20 FORMR_do20	F/O-IND, $O(N)$, $\texttt{IA}_{mon}$ O-IND, $O(N)$, $\texttt{IA}_{mon}$ F/O-IND, $O(N)$, $\texttt{IA}_{mon}$ F/O-IND, $O(N)$, $\texttt{IA}_{mon}$	62.6% 12.6% 12.1% 8.1%	0.076% 0.001% 0.016% 0.692%
ARC2D	SC = 96.7% PRIV, SLV	YPENT2_do11 *et al.*	F-IND, $O(1)$ Non-Overlap	10.2%	0.084%
MDG	SC = 99.3% PRIV, RED,	INTERF_do1000 POTENG_do2000	STATIC, NONE STATIC, NONE	91.7% 7.4%	0% 0%
SPEC77	SC = 91.3% PRIV,RED,SLV	SICDKD_do1000	F-IND, $O(1)$ Non-Overlap	4.1%	0.007%
OCEAN	SC = 88.8% PRIV,RED,SLV	FTRVMT_do109	F-IND, $O(1)$ Non-Overlap	63.2%	0.075%

`olda_do100/300` are (only) observed to form monotonic indexing before induction variable substitution. Our test extracts parallelism from all reported loops.

Hall *et al.* use interprocedural summarization [12,16] to give a comprehensive study of parallelism for the PERFECT Club and SPEC2000 benchmarks. Branch conditions are exploited to enhance summary precision and to predicate optimistic data-flow results. If independence cannot be decided statically, user tools test an independence predicate. While SUIF does not seem to target non-linear accesses, it handles loops `olda_100/300` in a way similar to Pugh's.

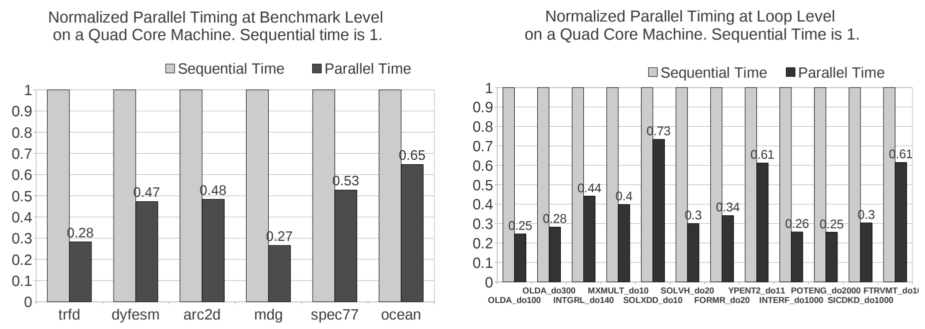
Another body of important work includes formulation of powerful symbolic algebra systems such as the one of Fahringer [9] that among others computes upper and lower bounds of nonlinear expressions, and the one of Engelen that uses a more general form of induction variables via recurrence chains [8].

Other work presents pattern matching solutions for code with conditional induction variables such as Lin's and Rus' stack [15] and pushback [26] arrays. While we handle cases as difficult as `TRACK`'s `extend_do400`, we do not discuss them here because there we rely on improving LMAD aggregation, while this paper handles non-linearity that fails (any) aggregation.

Finally, the other main direction of solving irregular accesses has been to analyze memory references at run-time [22,17,18]. These techniques have overhead proportional to the number of the original-loop accesses, and hence we use them only as a last resort, once all the lighter independence predicates have failed.

5 Experimental Evaluation

We evaluate our technique on a number of benchmarks known to be rich in non-linear accesses. Table 1 characterizes several such irregular loops, named in *column* 3, and their corresponding benchmark, named in *column* 1.



(a) Benchmark Timing on a Quad Core. (b) Loop Timing on a Quad-Core.

Fig. 5. Parallel Normalized Timing on a Quad-Core Machine

Column 2 shows the percentage of the sequential run-time coverage (SC) that has been parallelized, and the enabling transformations: privatization (PRIV), static/dynamic last value (SLV/DLV), reduction (RED). RT-IW and RT-RED correspond to predicates that may eliminate the need for privatization and reduction.

The *fourth column* classifies non-linear accesses: indirect arrays ($\texttt{IA}_{mon}$), polynomial indexing (i^2), and loop index spanning multiple dimensions, which corresponds to the quasi-linear test that checks that dimensions do *not overlap*. Predicates typically prove flow/output independence (F/O-IND) in $O(N)$, for $\texttt{IA}_{mon}$, and $O(1)$ run-time complexity for the other two. The *fifth column* shows the *weight* of the loop, as percentage of the sequential run time of the entire program. The *sixth column* demonstrates that the overhead of the extracted predicates, seen as a percentage of the concurrent loop run time, is negligible.

Figures 5(a) and 5(b) show the normalized timing for entire benchmarks and for selected loops (sequential time is 1). The tests and loops are shown in the order they appear in Table 1. We used IFORT version 11.1 to compile the source file generated by our compiler on a commodity INTEL quad-core Q9550@2.83GHz machine. Since the data-sets of these tests are very small, we used the `O0` level. Compilation under `O2` gives comparable speed-up for `trfd`, `mdg` and `spec77` benchmarks, some speed-up for `arc2D` and significant slowdown for `dyfesm` and `ocean`. The reason for the slowdown is that all important loops in the latter two benchmarks, while executed many times, have granularity in the range of tens of microseconds, which is too small to amortize the thread spawning overhead.

We have already discussed `TRFD`'s irregularities. All important loops from `DYFESM`: `MXMULT_do10`, `FORMR_do20`, `SOLXDD_do4/10/30/50`, `HOP_do20`, are proved flow/output independent via $O(N)$ monotonicity predicates that involve index arrays. We have found loop `SOLVH_do20` to be particularly interesting in that it requires predicates extracted from control flow *and* from both linear and non-linear LMADs. For example, flow independence of arrays `XE` and `HE` is verified with the $5 * \texttt{NNPED} \geq \texttt{NDFE} \ \wedge\ \texttt{NSYM} \neq 0$ and $8 * \texttt{NNPES} \geq \texttt{NSFE}$ predicates, respectively, while output independence of `HE` is verified via the $O(N)$ predicate:

$$\wedge_{\texttt{iss}=1}^{\texttt{NSS}-1}[-33 + \texttt{NSFE} + 32 * (\texttt{IDBEGS(iss)} + \texttt{NEPSS(iss)}) \ < \ 32 * \texttt{IDEBGS(iss}+1)].$$

Benchmarks `ARC2D`, `SPEC77` and `OCEAN` exhibit instances of the non-overlap, $O(1)$ test. `ARC2D`'s loops `YPENTA_do11`, `XPENT2_do11/15`, `YPENT2_do11/13` together have weight 10.2%, while `OCEAN`'s `FTRVMT_do109` shows a robust weight of 63.2%. Finally, while monotonicity tests are used on some unimportant loops of `MDG`, we included results here mainly because it has been reported non-linear [21]. Both important loops, `INTERF_do1000`, and `POTENG_do2000` are proved statically in our framework: the control-flow-based implication necessary to privatize `rl` is found *true* very early at USR aggregation level (otherwise a branch predicate would be extracted). We also apply run-time reduction and privatization.

In summary, on a four-core commodity system, we get an average speed-up on entire benchmarks of 2.24x and 2.52x, with maximal values of 3.70x and 3.97x, and minimal values of 1.54x and 1.40x, even when small data sets with (too) small granularity are used. The overhead of the monotonicity predicate is on average a negligible 0.15% of the loop concurrent run time.

6 Conclusions

In this paper we have presented a static analysis technique that can extract arbitrary predicates that can (in)validate the monotonicity property of memory reference summaries. These predicates are then simplified using symbolic elimination algorithms and implications of the assumed monotonicity. Finally, the predicates are evaluated at run time, where they can (in)validate loop independence. We have implemented the technique in our compiler and applied it on six benchmarks with difficult to analyze non-linear array references. The experimental results show that the dynamic evaluation of predicates represents a negligible overhead, especially when compared to previous run-time parallelization techniques. Our technology enabled the parallelization of the benchmarks with full coverage and produced scalable speedups.

Acknowledgment. This research was supported in part by NSF awards CRI-0551685, CCF-0833199, CCF-0830753, IIS-096053, IIS-0917266, NSF/DNDO award 2008-DN-077-ARI018-02, by the DOE NNSA under the Predictive Science Academic Alliances Program by grant DE-FC52-08NA28616, by by Chevron, IBM, Intel, Oracle/Sun and by Award KUS-C1-016-04, made by King Abdullah University of Science and Technology (KAUST).

References

1. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures. Morgan Kaufmann (2002)
2. Banerjee, U.: Speedup of Ordinary Programs. Ph.D. Thesis, Dept. of Comp. Sci. Univ. of Illinois at Urbana-Champaign Report No. 79-989 (1988)
3. Berry, M., et al.: The PERFECT Club Benchmarks: Effective Performance Evaluation of Supercomputers. Int. J. of Supercomputer Applications 3, 5–40 (1988)
4. Blume, W., Eigenmann, R.: Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks Programs. IEEE Trans. Par. Distr. Sys. 3, 643–656 (1992)
5. Blume, W., Eigenmann, R.: The Range Test: A Dependence Test for Symbolic, Non-Linear Expressions. In: Procs. Int. Conf. on Supercomp., pp. 528–537 (1994)

6. Blume, W., Eigenmann, R.: Demand-Driven, Symbolic Range Propagation. In: Huang, C.-H., Sadayappan, P., Banerjee, U., Gelernter, D., Nicolau, A., Padua, D.A. (eds.) LCPC 1995. LNCS, vol. 1033, pp. 141–160. Springer, Heidelberg (1996)
7. Dang, F., Yu, H., Rauchwerger, L.: The R-LRPD Test: Speculative Parallelization of Partially Parallel Loops. In: Procs. of Int. Parallel and Distributed Processing Symp., pp. 20–29 (2002)
8. Engelen, R.A.V.: A unified framework for nonlinear dependence testing and symbolic analysis. In: Procs. Int. Conf. on Supercomputing, pp. 106–115 (2004)
9. Fahringer, T.: Efficient symbolic analysis for parallelizing compilers and performance estimators. J. of Supercomputing 12, 227–252 (1997)
10. Feautrier, P.: Parametric Integer Programming. Operations Research 22(3), 243–268 (1988)
11. Feautrier, P.: Dataflow Analysis of Array and Scalar References. Int. J. of Parallel Programming 20(1), 23–54 (1991)
12. Hall, M.W., et al.: Interprocedural parallelization analysis in SUIF. ACM Trans. on Programming Languages and Systems 27(4), 662–731 (2005)
13. Hoeflinger, J., Paek, Y., Yi, K.: Unified Interprocedural Parallelism Detection. Int. J. of Parallel Programming 29(2), 185–215 (2001)
14. Lin, Y., Padua, D.: Demand-Driven Interprocedural Array Property Analysis. In: Carter, L., Ferrante, J. (eds.) LCPC 1999. LNCS, vol. 1863, pp. 303–317. Springer, Heidelberg (2000)
15. Lin, Y., Padua, D.: Analysis of Irregular Single-Indexed Array Accesses and Its Applications in Compiler Optimizations. In: Watt, D.A. (ed.) CC 2000. LNCS, vol. 1781, pp. 202–218. Springer, Heidelberg (2000)
16. Moon, S., Hall, M.W.: Evaluation of predicated array data-flow analysis for automatic parallelization. In: Procs. Int. Princ. Pract. of Par. Prog., pp. 84–95 (1999)
17. Oancea, C.E., Mycroft, A.: Set-Congruence Dynamic Analysis for Thread-Level Speculation (TLS). In: Amaral, J.N. (ed.) LCPC 2008. LNCS, vol. 5335, pp. 156–171. Springer, Heidelberg (2008)
18. Oancea, C.E., Mycroft, A., Harris, T.: A Lightweight, In-Place Model for Software Thread-Level Speculation. In: Procs. Symp. Paral. Alg. Arch., pp. 223–232 (2009)
19. Paek, Y., Hoeflinger, J., Padua, D.: Efficient and Precise Array Access Analysis. ACM Trans. on Programming Languages and Systems 24(1), 65–109 (2002)
20. Pugh, W.: The Omega Test: a Fast and Practical Integer Programming Algorithm for Dependence Analysis. Communications of the ACM 8, 4–13 (1992)
21. Pugh, W., Wonnacott, D.: Nonlinear Array Dependence Analysis. In: Proc. of Workshop on Lang. Comp. and Run-Time Support for Scallable Systems (1995)
22. Rauchwerger, L., Padua, D.: The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. IEEE Trans. Par. Distr. Sys. 10(2), 160–199 (1999)
23. Rauchwerger, L., Amato, N.M., Padua, D.A.: A scalable method for run-time loop parallelization. Int. J. of Parallel Programming 26, 26–6 (1995)
24. Rus, S., Hoeflinger, J., Rauchwerger, L.: Hybrid analysis: Static & dynamic memory reference analysis. Int. J. of Parallel Programming 31(3), 251–283 (2003)
25. Rus, S., Pennings, M., Rauchwerger, L.: Sensitivity Analysis for Automatic Parallelization on Multi-Cores. In: Procs. Int. Conf. on Supercomp., pp. 263–273 (2007)
26. Rus, S., Zhang, D., Rauchwerger, L.: The Value Evolution Graph and its Use in Memory Reference Analysis. In: Procs. of Int. Conf. on Parallel Architectures and Compilation Techniques, pp. 243–254 (2004)
27. Yu, H., Rauchwerger, L.: Techniques for Reducing the Overhead of Run-Time Parallelization. In: Watt, D.A. (ed.) CC 2000. LNCS, vol. 1781, pp. 232–248. Springer, Heidelberg (2000)

OpenCL as a Programming Model for GPU Clusters

Jungwon Kim, Sangmin Seo, Jun Lee,
Jeongho Nah, Gangwon Jo, and Jaejin Lee

Center for Manycore Programming
School of Computer Science and Engineering
Seoul National University, Seoul 151-744, Korea
{jungwon,sangmin,jun,jeongho,gangwon}@aces.snu.ac.kr,
jlee@cse.snu.ac.kr
http://aces.snu.ac.kr

Abstract. In this paper, we propose an OpenCL framework for GPU clusters. The target cluster architecture consists of a single host node and multiple compute nodes. They are connected by an interconnection network, such as Gigabit and InfiniBand switches. Each compute node consists of multiple GPUs. Each GPU becomes an OpenCL compute device. The host node executes the host program in an OpenCL application. Our OpenCL framework provides an illusion of a single system for the user. It allows the application to utilize GPUs in a compute node as if they were in the host node. No communication API, such as the MPI library, is required in the application source. We show that the original OpenCL semantics naturally fits to the GPU cluster environment, and the framework achieves both high performance and ease of programming. We implement the OpenCL framework and evaluate its performance on a GPU cluster that consists of one host and eight compute nodes using six OpenCL benchmark applications.

1 Introduction

Open Computing Language (OpenCL)[11] is a unified programming model for different types of computational units in a single heterogeneous computing system. OpenCL provides a common hardware abstraction layer across different computational units. Programmers can write OpenCL applications once and run them on any OpenCL-compliant hardware. This portability is one of OpenCL's chief advantages. It allows programmers to focus their efforts on the functionality of their application rather than the lower-level details of the underlying architecture. Some industry-leading hardware vendors such as AMD[2], IBM[9], Intel[10], NVIDIA[17], and Samsung[19] have provided OpenCL implementations for their hardware. This makes OpenCL a standard for general-purpose, heterogeneous parallel programming model.

However, one of the limitations of current OpenCL is that it is restricted to a single node in a cluster system. It does not work for a cluster of multiple nodes unless the developer explicitly uses communication libraries, such as MPI. The same thing is true for CUDA[16]. A GPU cluster contains multiple GPUs across its nodes to solve bigger problems within an acceptable time frame[4,7,8,18]. As such clusters widen their user base, application developers for the clusters are being forced to turn to an incompatible mix of programming models, such as MPI-OpenCL and MPI-CUDA. Thus, the application becomes a mixture of a communication API and OpenCL (or CUDA). This makes the application more complex, less portable, and hard to maintain.

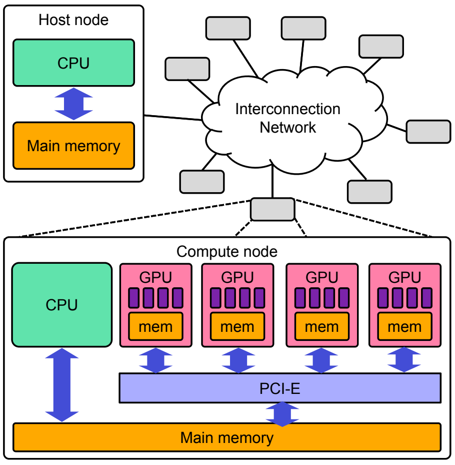


Fig. 1. The target cluster architecture

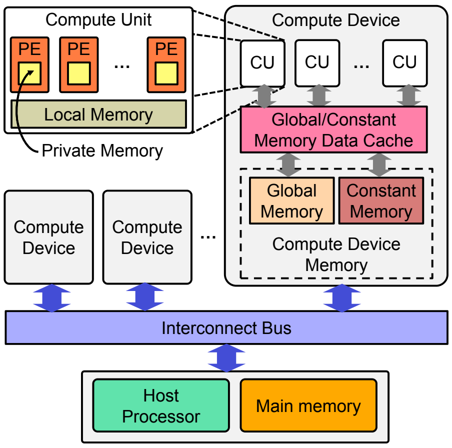


Fig. 2. The OpenCL platform model

In this paper, we propose an OpenCL framework and we show that our OpenCL framework can be a unified programming model for GPU clusters. The target cluster architecture is shown in Figure 1. It consists of a single host node and multiple compute nodes. The nodes are connected by an interconnection network, such as Gigabit and InfiniBand switches. The host node executes the host program in an OpenCL application. Each compute node consists of multiple GPUs. A single GPU becomes an OpenCL compute device. A GPU has its own device memory, up to several gigabytes. Within a compute node, data is transferred between the GPU device memory and the main memory through a PCI-E bus.

Our OpenCL framework provides an illusion of a single system for the user. It allows the application to utilize GPUs in a compute node as if they were in the host node. Thus, it enables OpenCL applications written for a single system (i.e., single node) to run on the cluster without any modification. It makes the application more portable. To achieve an illusion of a single system for the cluster, our OpenCL runtime hides communication between nodes from the user. The user can launch a kernel to a GPU or manipulate memory object using only OpenCL API functions.

The major contributions of this paper are the following:

– We show that the original OpenCL semantics naturally fits to the GPU cluster environment.
– We describe the design and implementation of our OpenCL framework (the OpenCL runtime and source-to-source translators) for GPU clusters.
– We show the effectiveness of our OpenCL framework by implementing the OpenCL runtime and the source-to-source translators. We evaluate its performance with a GPU cluster that consists of one host node and eight compute nodes using six OpenCL benchmark applications from various sources.

The rest of the paper is organized as follows. Section 3 describes the design and implementation of the OpenCL framework. Section 4 discusses and analyzes the evaluation results of our OpenCL framework. Section 5 surveys related work. Finally, Section 6 concludes the paper.

2 Background

In this section, we briefly introduce OpenCL.

2.1 The OpenCL Platform Model

The OpenCL platform model (Figure 2) consists of a host processor connected to one or more OpenCL compute devices. A compute device is divided into one or more compute units (CUs), each of which contains one or more processing elements (PEs).

An OpenCL application consists of two parts: kernels and a host program. A kernel is a function and written in OpenCL C. It executes on a compute device. The host program runs on the host processor and enqueues a command to a command-queue that is attached to a compute device. A kernel command executes a kernel on the PEs within the compute device. A memory command controls a buffer object, and a synchronization command enforces an ordering between commands. The OpenCL runtime schedules the enqueued kernel command on the associated compute device and executes the enqueued memory or synchronization command directly.

When a kernel command is enqueued, an abstract index space has been defined. The index space called NDRange is an N-dimensional space, where N is equal to 1, 2, or 3. An NDRange is defined by an N-tuple of integers and specifies the extent of the index space (the dimension and the size). An instance of the kernel executes for each point in this index space. This kernel instance is called a *work-item*, and is uniquely identified by its global ID (N-tuples) defined by its point in the index space. Each work-item executes the same code but the specific pathway and accessed data can vary.

A *work-group* contains one or more work-items. Each work-group has a unique ID that is also an N-tuple. An integer array of length N (i.e., the dimension of the index space) species the number of work-groups in each dimension of the index space. A work-item in a work-group is assigned a unique local ID within the work-group, treating the entire work-group as the local index space. The global ID of a work-item can be computed with its local ID, work-group ID, and work-group size. Work-items in a work-group execute concurrently on the PEs of a single CU.

2.2 The OpenCL Memory Model

OpenCL defines four distinct memory regions in a compute device: global, constant, local and private as shown in Figure 2. To distinguish these memory regions, OpenCL C has four address space qualifiers: `__global`, `__constant`, `__local`, and `__private`. They are used in variable declarations in the kernel code.

An OpenCL memory object is a handle to a region of the global memory in the device. The host program can dynamically create memory objects and enqueue commands to read from and write to memory objects. A memory object is not associated with a specific compute device. Thus, different compute devices can share memory objects. A memory object in the global memory is typically a *buffer object*, called a *buffer* in short. A buffer stores a one-dimensional collection of elements that can be a scalar data type, a vector data type, or a user-defined structure. Even though the space for a buffer is allocated in the global memory of a specific device, the buffer is not bound to the compute device in OpenCL[11]. Binding a buffer and a compute device is implementation dependent.

OpenCL defines a relaxed memory consistency model for consistent memory. An update to a memory location by a work-item may not be visible to all the other work-items at the same time. Instead, the local view of memory from each work-item is guaranteed to be consistent at synchronization points. Synchronization points include work-group barriers, command-queue barriers, and events.

3 The OpenCL Framework

In this section, we describe the design and implementation of our OpenCL framework for the GPU clusters.

3.1 Organization of the Runtime

Figure 3 shows the organization of our OpenCL runtime. The runtime for the host node runs two threads: *host thread* and *command scheduler*. The runtime also maintains OpenCL command-queues for each compute device and a *completion-queue* in the host node. The completion-queue contains event objects that are associated with completed kernel-execution or memory commands. When a user launches an OpenCL application in the host node, the host thread executes the host program in the application. The command scheduler in the host node schedules enqueued commands across compute nodes in the cluster.

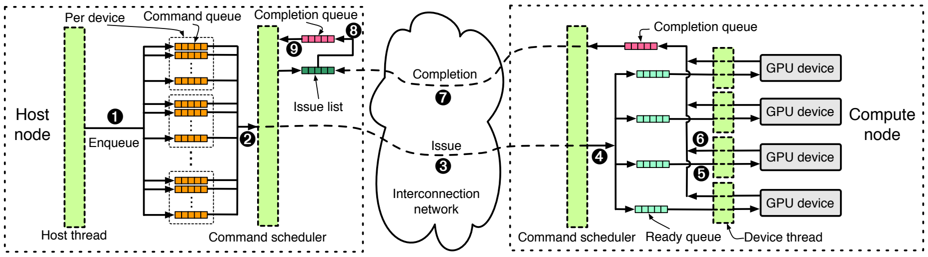


Fig. 3. The organization of our OpenCL runtime

The runtime for a compute node maintains a *ready-queue* for each compute device and a *completion-queue* in the compute node. The ready-queue contains commands that are issued but not launched to the associated compute device yet. The completion-queue contains event objects that are associated with completed kernel-execution or memory commands at the compute node side. The runtime runs a *device thread* for each compute device in the compute node. The device thread dequeues a command from its ready-queue and launches the kernel to the associated compute device when the command is a kernel-execution command and the compute device is idle. If it is a memory command, the device thread executes the command directly. The runtime for a compute node also runs a *command scheduler thread*. The command scheduler receives commands from the host node and schedules them across compute devices in the compute node.

The command scheduler in each node repeats scheduling commands and checking the completion queue in turn until the OpenCL application terminates. When the completion queue is not empty, the command scheduler dequeues the completion queue and updates the status of the dequeued events from issued to completed. It is also in charge of communicating with other nodes. Communications between different nodes are implemented with a lower-level communication API, such as MPI. To implement the runtime for each compute node, an existing CUDA or OpenCL runtime for a single system can be used.

In the host node, the host thread and the command scheduler share the OpenCL command-queues. A compute device may have one or more command-queues as shown in Figure 3. The command scheduler dequeues each command-queue one by one.

Our OpenCL runtime assigns a unique ID to each OpenCL object, such as context, compute device, memory, program, kernel, event, etc.

OpenCL supports synchronization between work-items in a work-group using work-group barriers. Every work-item in the work-group must execute the barrier and cannot proceed beyond the barrier until all other work-items in the work-group reach the barrier. Between work-groups, there is no synchronization mechanism available in OpenCL.

Synchronization between commands in a single command-queue can be specified by a command-queue barrier command. To synchronize commands between different command-queues, *events* are used. Each OpenCL API functions that enqueues a command return an event object that encapsulates the command status. Most of OpenCL API functions that enqueue a command take an *event list* as an argument. This command cannot be issued for execution until all the commands associated with the event list complete.

The command scheduler in the host node honors the type (in-order or out-of-order) of each command-queue and (event) synchronization enforced by the host program. When the command scheduler dequeues a synchronization command, the command scheduler uses it for determining execution ordering between queued commands. It maintains a data structure to store the events that are associated with queued commands and bookkeeps the ordering between the commands. When there is no event for which a queued command waits, the command is dequeued and issued to its target compute node that contains the target compute device.

When the command scheduler in the host node dequeues a kernel-execution command from a command-queue, the command scheduler *issues* the command by sending a *kernel-execution command message* to the target compute node that contains the target compute device associated with the command-queue.

A kernel-execution command message contains the information required to execute the original command. It contains target compute device ID, kernel ID, number of kernel arguments, values of the kernel arguments, IDs of buffers that are accessed by the kernel, and kernel index space information. In addition, it contains the ID of the event object that is associated with the kernel-execution command. The command scheduler sends the message to the compute node that contains the target compute device. The event ID is used later by the command scheduler in the target compute node to notify the completion of the command to the host node.

After the command scheduler sends the command message to the target compute node, it calls a non-blocking receive communication API function to wait for the completion message from the target node. The command scheduler encapsulates the receive request in the command event object and adds the event in the *issue list*. The issue list contains event objects associated with the commands that have been issued but not completed yet. When the receive request completes, the associated event object is removed from the issue list and inserted to the *completion queue* in the host node.

The command scheduler in the target compute node receives the kernel-execution command message from the host command scheduler. It creates a command object and an associated event object from the message. After extracting the target device ID from the message, the command scheduler enqueues the command object to the ready-queue of the target device. The target device thread dequeues the ready-queue and launches the kernel to the device when the target compute device is idle. When the compute device completes executing the kernel, the device thread updates the status of the associated event to completed, and then inserts the event to the completion queue. The command scheduler in the target compute node dequeues the event and sends a completion message to the host node.

When the dequeued command from a command-queue is a memory command, the host command scheduler sends memory command messages to target compute nodes. Depending on the type of the memory command, the number of target compute nodes is one or two. The remaining procedure to process the memory command in the host node and the target compute node is similar to that of a kernel-execution command. But, the target device thread executes the memory command directly. In the next section, we elaborate on the memory management techniques used by our OpenCL runtime.

3.2 Memory Allocation to Buffers

In OpenCL, the host program creates a buffer object by invoking an API function `clCreateBuffer()`. Since an OpenCL buffer is not associated with a specific compute device, `clCreateBuffer()` has no parameter that specifies a compute device. This implies that when a buffer is created, the runtime has no information about which compute device uses the buffer. Thus, our OpenCL runtime does not allocate any memory space to the buffer when `clCreateBuffer()` is invoked. Instead, when the host program enqueues a memory command that manipulates the buffer or a kernel execution command that uses the buffer, the runtime allocates a memory space to the buffer.

However, there is an exception to this rule. When the host program invokes `clCreateBuffer()` with the `CL_MEM_COPY_HOST_PTR` flag, it wants to allocate a space to the buffer and copy the data from the host main memory to the buffer. If the runtime delays the space allocation until the buffer binds to a specific compute device, the runtime may lose the original data to be copied because the data may have been changed. Thus, the runtime allocates a temporary space in the host main memory to the buffer and copies the data to the temporary space. After the compute device associated with the buffer is known, the runtime allocates a space in the device's global memory and copies the data from the temporary space to the space in the global memory. Then, it discards the temporary space for the buffer.

Our OpenCL runtime maintains a *device list* for each buffer. The device list contains compute devices that have the same latest copy of the buffer in their global memory. It is empty when the buffer is created. When the host command scheduler dequeues a memory command or kernel-execution command, it checks the device list of each buffer that is accessed by the command. If the target compute device is in the device list of a buffer, the compute device has a copy of the buffer. Otherwise, the runtime allocates a space for the buffer in the global memory of the target device. Then, the runtime copies the buffer to the target device from the nearest compute device in the device list. A device in the same compute node is preferred to a device in the different compute node to avoid extra data transfers.

When the command that accesses the buffer completes, the host command scheduler updates the device list of the buffer. If the buffer contents are modified by the command, it empties the list and adds the device that has the modified copy of the buffer in the list. Otherwise, it just adds in the list the device who has recently obtained a copy of the buffer due to the command.

3.3 Buffer Manipulation

To copy an array in the host program to a buffer, the host program enqueues a memory command to a command-queue by invoking an OpenCL API function `clEnqueueWriteBuffer()`. When the host command scheduler dequeues the memory command, the command scheduler sends a *buffer-write* command message to the compute node that has the target compute device associated with the command-queue. The command scheduler of the target compute node allocates a temporary space in the main memory to receive the data. This is because the lower-level communication API does not typically support writing directly to the compute device memory such as GPU memory. After receiving the data in the temporary space, the command scheduler enqueues a buffer-write command in the ready-queue of the compute device using the temporary space as the source. The associated device thread dequeues the buffer-write command from the ready-queue and copies the data from the temporary space to the compute device memory. When the compute device notifies the completion of the buffer-write command to the device thread, it updates the status of the event object of the original buffer-write command from the host node and inserts the object in the completion queue.

To copy a buffer to an array in the host program, the host program enqueues a memory command to a command-queue by invoking `clEnqueueReadBuffer()`. Since the buffer is not bound to a compute device in OpenCL, it is not guaranteed that the compute device associated with the command-queue has the latest copy of the buffer. Thus, the host command scheduler first checks the device list of the buffer when it dequeues the memory command. It selects a device in the list and sends a buffer-read command message to the compute node that contains the device. Then, it receives the data from the target compute node through the lower-level communication API.

By invoking `clEnqueueCopyBuffer()`, the host program enqueues the buffer-copy command to a command-queue. The command copies the contents of one buffer to another buffer. Since the binding between buffers and compute devices is implementation dependent, our OpenCL framework assumes that the target buffer is bound to the compute device that is associated with the command-queue.

When the host command scheduler dequeues the buffer-copy command, it identifies the nearest device to the target device by checking the device list of the source buffer. The selected device becomes the source device for the copy process. There are three cases: (1) The source and target devices are the same. (2) The source and target devices are different but they are in the same node, and (3) The source and target devices are in different nodes.

For the first and second cases, the host command scheduler sends a *buffer-copy* command message to the compute node that has the source and target compute devices. When the command scheduler in the compute node receives the message, it checks if the source and target devices are the same. If they are the same, the command scheduler enqueues a buffer-copy command to the ready-queue of the compute device. Otherwise, the command scheduler allocates a temporary space in the main memory. Then, it enqueues a buffer-read command and a buffer-write command to the ready-queues of the source device and the target device, respectively. The commands are synchronized with an event so that the buffer-read command finishes before the buffer-write commands starts. The device thread associated with the source copies the buffer from the source device memory to the temporary space, and the device thread associated with the target copies the contents of the temporary space to the target buffer.

For the third case, the host command scheduler sends a buffer-copy command message to the source compute node and another buffer-copy command message to the target compute node. When the command scheduler in the source node receives the message, it copies the buffer from the source device to a temporary space in the main memory by enqueueing a buffer-read command to the ready-queue of the source device. Then it sends the copy to the target compute node using the lower-level communication API.

When the command scheduler in the target compute node receives the buffer-copy command message, it receives the copy in a temporary space in the main memory and enqueues a buffer-write command in the ready-queue of the target device. At the end, the device thread associated with the target device inserts the event object of the buffer-copy command it in the completion queue. After dequeueing the event from the completion queue, the command scheduler sends a completion message to the host node.

```
__kernel void vec_add(__global float *A, __global float *B,
                      __global float *C) {
  int id = get_global_id(0);
  C[id] = A[id] + B[id];
}
```

(a)

```
int vec_add_memory_flags[3] = {
    CL_MEM_READ_ONLY, // A
    CL_MEM_READ_ONLY, // B
    CL_MEM_WRITE_ONLY // C
};
```

(b)

```
__global__ void vec_add(float *A, float *B, float *C) {
  int id = blockDim.x * blockIdx.x + threadIdx.x;
  C[id] = A[id] + B[id];
}
```

(c)

Fig. 4. (a) An OpenCL kernel. (b) The buffer access information of kernel `vec_add` for the runtime. (c) The CUDA C code generated for a GPU.

3.4 Consistency Management

A buffer object can be shared between different compute devices in OpenCL. Multiple kernel-execution and memory commands can be executed simultaneously, and each of them may access a copy of the same buffer. If they update the same set of locations in the buffer, we may choose any copy as the last update for the buffer according to the OpenCL memory consistency model. However, when they update different locations in the same buffer, the problem is similar to the false sharing problem that occurs in a traditional, page-level software shared virtual memory system for clusters[3].

One solution to this problem is introducing a multiple-writers protocol[3] that maintains a twin for each writer of the buffer and updates the original copy of the buffer by comparing the modified copy with its twin. Each node that contains a writer device performs the comparison and sends the result (e.g., diff) to the host who maintains the original buffer. The host updates the original buffer with the result. However, this introduces a significant communication and computation overhead in the cluster environment if the degree of buffer sharing is high.

Instead, our OpenCL runtime solves this problem by serializing executions of those commands in addition to keeping the most up-to-date copies using the device list. When the host command scheduler issues a memory command or kernel-execution command, it records the buffers that are written by the command in a list called *written-buffer list*. When the host command scheduler dequeues a command, and the command writes to any buffer in the written-buffer list, it delays issuing the command until the buffers accessed by the command are removed from the written-buffer list. Whenever a kernel-execution or memory command completes its execution, the command scheduler removes the buffers written by the completed command.

To detect the set of buffers written by an OpenCL kernel, our framework performs a conservative pointer analysis on the kernel source code when the kernel is built. Since OpenCL imposes a restriction on the usage of global memory pointers in a kernel[11]. Specifically, a pointer to address space A can only be assigned to a pointer to the same address space A. Casting a pointer to address space A to a pointer to address space B ($\neq$ A) is illegal. Thus, a simple, conservative pointer analysis is enough to obtain the buffers written by the kernel.

When the host builds a kernel by invoking `clBuildProgram()`, our OpenCL-C-to-C translator at the host node generates the buffer access information for the runtime from the OpenCL kernel code. Figure 4 (b) shows the information generated from the OpenCL kernel in Figure 4 (a). It is an array of integer for each kernel. The Nth element of the array represents the access information of the Nth buffer argument of the kernel. Figure 4 (b) indicates that the first and second buffer arguments (A and B) are read, and the third buffer argument (C) is written by kernel `vec_add`. The runtime uses this information to manage buffer consistency.

3.5 Distributing the Kernel Code

When the host finishes building a kernel by invoking `clBuildProgram()`, it sends the binary of the kernel to each compute node. The receiver compute node stores it and accesses it when the kernel-execution command is issued from the host node. Our OpenCL-C-to-CUDA-C translator (we assume that the runtime in a compute node is implemented with the CUDA runtime) generates the code for a GPU. Figure 4 (c) shows the code generated for a GPU device.

4 Evaluation

This section describes the evaluation methodology and results for our OpenCL framework.

4.1 Methodology

Target Cluster Architecture. We evaluate our OpenCL framework using a GPU cluster system that consists of one host node and eight compute nodes. The host node has two Intel Xeon X5680 hexa-core CPUs with 72GB DDR3 main memory. Each compute node consists of two Intel Xeon X5660 hexa-core CPUs with 24GB DDR3 main memory and four NVIDIA GTX 480 GPUs, resulting in total of 32 GPUs across the entire cluster.

Table 1. Applications used

Application	Source	Description	Input	Global Mem. Size
BinomialOption	AMD	Binomial option pricing	65504 samples, 512 steps, 100 iterations	2MB
BlackScholes	PARSEC	Black-Scholes PDE	33538048 options, 10000 iterations	895.6MB
CP	Parboil	Coulombic potential	16384x16384, 100000 atoms	1024.1MB
EP	NAS	Embarrassingly parallel	Class D	0.8MB
MatrixMul	NVIDIA	Matrix multiplication	10752x10752	1323MB
Nbody	NVIDIA	N-Body simulation	5242880 bodies	320MB

Benchmark Applications. We use six OpenCL applications from various sources: AMD[1], NAS[14], NVIDIA[15], Parboil[20], and PARSEC[5]. The characteristics of the applications are summarized in Table 1. The applications from NAS, Parboil and PARSEC are translated to OpenCL applications manually. For an OpenCL application written for a single GPU, we modify the application to distribute workload across multiple GPUs and manage data between the host main memory and multiple GPU device memories.

Runtime and Source-to-Source Translators. We have implemented the OpenCL runtime and source-to-source translators. The runtime in the compute node is implemented with CUDA 3.2[16]. We have implemented our OpenCL-C-to-C and OpenCL-C-to-CUDA-C translators by modifying clang that is a C front-end for the LLVM compiler infrastructure[13].

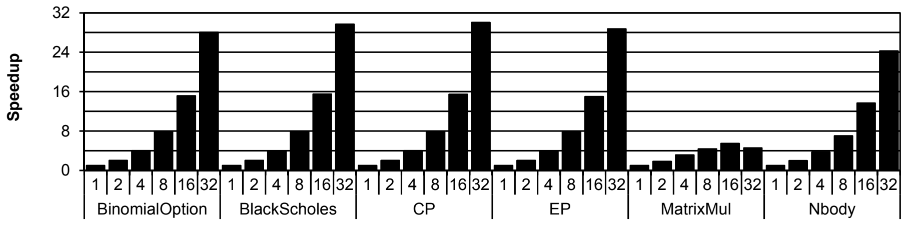


Fig. 5. Speedup of the GPU cluster over a single GPU

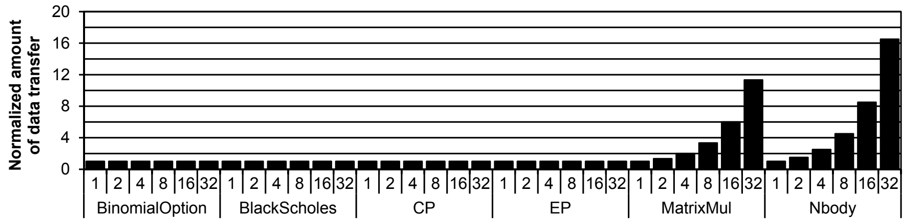


Fig. 6. Normalized total amount of data transfer between nodes

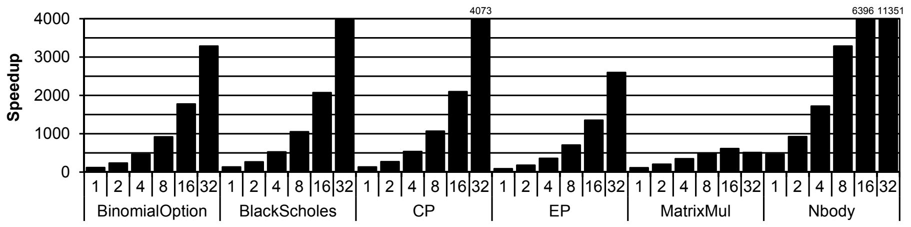


Fig. 7. Speedup of the GPU cluster over a single CPU core

4.2 Results

Figure 5 shows the speedup of our OpenCL framework over a single GPU for each application. We vary the number of GPUs from 2 to 32 in powers of two. Since a compute node contains four GPUs, we use up to eight compute nodes. All applications but MatrixMul scale well. The performance of the applications is almost proportional to the number of GPUs.

The total execution time of an application mainly consists of computation (kernel execution) and communication time. The kernel execution time decreases linearly with the number of GPUs for all applications. This accounts for the even distribution of the kernel workload across GPUs.

The communication time increases as the amount of data transfer increases. Figure 6 shows the normalized total amount of data transfer between nodes. The amount is normalized to that of a single GPU. The amount of data transfer in BinomialOption, BlackScholes, CP, and EP does not change as the number of GPUs increase. This is because there is no data sharing between work-items in their kernels. In MatrixMul and Nbody, work-items share data with each other. Thus, when the kernel workload is distributed across GPUs, some data items are duplicated in multiple GPU device memories. As a result, the amount of data transfer between nodes increase as the number of GPUs increases. In turn, the communication overhead also increases as the number of GPUs increases due the data transfer.

Since the communication overhead dominates the performance of MatrixMul, it does not scale well. Unlike MatrixMul, the kernel execution time of Nbody is much bigger than the data transfer time. In addition, the absolute amount of data interchanged between GPUs is small for Nbody. Thus, the reduced kernel execution time by the multiple GPUs dominates Nbody's total execution time.

For your reference, Figure 7 shows the speedup of each application over a single CPU core for different number of GPUs. The sequential CPU version of each application is obtained from the same source in Table 1. For MatrixMul, we optimize the sequential CPU version using the tiling technique; this is because the OpenCL version of MatrixMul exploits the same tiling optimization.

5 Related Work

Heterogeneous computing has been drawn much attention due to its parallelism, energy efficiency and cost effectiveness. Recently, there have been many studies done on GPU clusters[8,18,7]. However, there are few literature found that show a unified programming model for such clusters.

Chen *et al.*[6] propose new language extensions to Unified Parallel C (UPC) in order to take advantage of GPU clusters. They extend UPC with hierarchical data distribution and introduce the implicit thread hierarchy. They implement the compiler and runtime system, and show that their model has better programmability than the mixed MPI/CUDA approach, and the model is effective to achieve good performance on GPU clusters. We, on the other hand, show that the original OpenCL semantics naturally fits to the GPU clusters, and present the OpenCL framework for such clusters.

Kim *et al.*[12] propose an OpenCL framework for multiple GPUs in a system. The OpenCL framework provides an illusion of a single compute device to the programmer for the multiple GPUs available in the system. It automatically partitions the work-group index space of the kernel at run time. To find an optimal partition that minimizes data transfer through the PCI-E bus between the host and GPUs, they use a sampling technique that analyzes the buffer access ranges in the kernel. To achieve a single compute device image, the runtime maintains a virtual device memory and copies them to each device memory when required. While their proposed OpenCL framework provides an illusion of a single compute device to the programmer, our OpenCL framework provides an illusion of a single system to the user for the multiple compute nodes available in GPU clusters.

6 Conclusions and Future Work

We introduce the design and implementation of an OpenCL framework that provides an illusion of a single system to the programmer for GPU clusters. It allows the application to utilize GPU devices in a remote node as if they were in the local node. Thus, it enables OpenCL applications written for a single system to run on the GPU cluster without any modification and makes the application more portable. To achieve an illusion of a single system for the GPU cluster, our OpenCL runtime hides communication between nodes from the user. The user can launch a kernel to a compute device or manipulate memory using OpenCL API functions. We implement the OpenCL runtime and source-to-source translator. The experimental results with six OpenCL benchmark applications and a GPU cluster that consists of one host and eight compute nodes indicate that our approach is practical and promising.

There are several avenues for future work. First, we would like to extend our OpenCL framework to support heterogeneous CPU/GPU clusters. Multicore CPUs in compute nodes can be compute devices to execute OpenCL kernels. In addition, we would like to extend OpenCL functionality for copying buffers between compute devices. Unlike MPI, OpenCL has no APIs for collective communications. With the help of collective communication APIs, the OpenCL framework can achieve both high performance and ease of programming. Finally, our longer-term plan is to provide an illusion of a single compute device to the programmer for the multiple compute devices available in heterogeneous CPU/GPU clusters.

Acknowledgements. This work was supported in part by grant 2009-0081569 (Creative Research Initiatives: Center for Manycore Programming) from the National Research Foundation of Korea. This work was also supported in part by the Ministry of Education, Science and Technology of Korea under the BK21 Project. ICT at Seoul National University provided research facilities for this study.

References

1. AMD: AMD Accelerated Parallel Processing SDK v2.3, `http://developer.amd.com/gpu/AMDAPPSDK/Pages/default.aspx`
2. AMD: AMD Accelerated Parallel Processing (APP) SDK With OpenCL 1.1 Support (2011), `http://developer.amd.com/gpu/atistreamsdk/pages/default.aspx`
3. Amza, C., Cox, A.L., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., Yu, W., Zwaenepoel, W.: TreadMarks: Shared Memory Computing on Networks of Workstations. Computer 29, 18–28 (1996)
4. Barak, A., Ben-nun, T., Levy, E., Shiloh, A.: A Package for OpenCL Based Heterogeneous Computing on Clusters with Many GPU Devices. In: Proceedings of the Workshop on Parallel Programming and Applications on Accelerator Clusters, PPAAC 2010 (2010)
5. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The PARSEC benchmark suite: characterization and architectural implications. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, PACT 2008, pp. 72–81 (2008)

6. Chen, L., Liu, L., Tang, S., Huang, L., Jing, Z., Xu, S., Zhang, D., Shou, B.: Unified Parallel C for GPU Clusters: Language Extensions and Compiler Implementation. In: Cooper, K., Mellor-Crummey, J., Sarkar, V. (eds.) LCPC 2010. LNCS, vol. 6548, pp. 151–165. Springer, Heidelberg (2011)
7. Chen, Y., Cui, X., Mei, H.: Large-scale FFT on GPU clusters. In: Proceedings of the 24th ACM International Conference on Supercomputing, ICS 2010, pp. 315–324 (2010)
8. Fan, Z., Qiu, F., Kaufman, A., Yoakum-Stover, S.: GPU cluster for high performance computing. In: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing, SC 2004, pp. 47–58 (2004)
9. IBM: OpenCL Development Kit for Linux on Power (2011), http://www.alphaworks.ibm.com/tech/opencl
10. Intel: Intel OpenCL SDK (2011), http://software.intel.com/en-us/articles/intel-opencl-sdk/
11. Khronos OpenCL Working Group: The OpenCL Specification Version 1.1 (2010), http://www.khronos.org/opencl
12. Kim, J., Kim, H., Lee, J.H., Lee, J.: Achieving a single compute device image in OpenCL for multiple GPUs. In: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, PPoPP 2011, pp. 277–288 (2011)
13. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization, CGO 2004, pp. 75–86 (2004)
14. NASA Advanced Supercomputing Division: NAS Parallel Benchmarks version 3.2, http://www.nas.nasa.gov/Resources/Software/npb.html
15. NVIDIA: NVIDIA CUDA Toolkit 3.2, http://developer.nvidia.com/cuda-toolkit-32-downloads
16. NVIDIA: NVIDIA CUDA C Programming Guide 3.2 (2010)
17. NVIDIA: NVIDIA GPU Computing Developer Home Page (2011), http://developer.nvidia.com/object/gpucomputing.html
18. Phillips, J.C., Stone, J.E., Schulten, K.: Adapting a message-driven parallel application to GPU-accelerated clusters. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, SC 2008, pp. 8:1–8:9 (2008)
19. Seoul National University and Samsung: SNU-SAMSUNG OpenCL Framework (2010), http://opencl.snu.ac.kr
20. The IMPACT Research Group: Parboil Benchmark suite, http://impact.crhc.illinois.edu/parboil.php

CellCilk: Extending Cilk for Heterogeneous Multicore Platforms

Tobias Werth[1], Silvia Schreier[2], and Michael Philippsen[1]

[1] University of Erlangen-Nuremberg, Computer Science Department,
Programming Systems Group, Germany
{werth,philippsen}@cs.fau.de
[2] University of Hagen, Faculty of Mathematics and Computer Science,
Chair of Data Processing Technology, Germany
silvia.schreier@fernuni-hagen.de

Abstract. The potential of heterogeneous multicores, like the Cell BE, can only be exploited if the host and the accelerator cores are used in parallel and if the specific features of the cores are considered. Parallel programming, especially when applied to irregular task-parallel problems, is challenging itself. However, heterogeneous multicores add to that complexity due to their memory hierarchy and specialized accelerators. As a solution for these issues we present CellCilk, a prototype implementation of Cilk for heterogeneous multicores with a host/accelerator design, using the Cell BE in particular. CellCilk introduces a new keyword (`spu_spawn`) for task creation on the accelerator cores. Task scheduling and load balancing are done by a novel dynamic cross-hierarchy work-stealing regime. Furthermore, the CellCilk runtime employs a garbage collection mechanism for distributed data structures that are created during scheduling. On benchmarks we achieve a good speedup and reasonable runtimes, even when compared to manually parallelized codes.

Keywords: Cilk, work stealing, heterogeneous multicores, parallel computing, Cell BE.

1 Introduction

In the ongoing trend to multicore one approach to tackle the memory wall, which is caused by different growth rates for processor vs. memory speed, are heterogeneous architectures. These platforms often consist of a host CPU plus some accelerators that use a different instruction set. Usually the accelerator cores are not connected to the host's main memory. Instead, the programmer has to use direct memory access (DMA) transfers to move data between main memory and their smaller scratchpad memories. Such a hierarchical memory design promises increased performance at the price of more programming complexity.

The Cell BE is such a heterogeneous multicore. Its host CPU is one PowerPC Unit (PPU). Its accelerators are Synergistic Processing Units (SPUs) with a single-instruction multiple-data (SIMD) instruction set. They are connected by a high bandwidth bus. Code and data reside in a 256 KB sized small local store on each SPU that can be used as a scratchpad memory. The potential of the Cell BE can only be exploited by using all of the SPUs in parallel. While dealing with the increased hardware complexity may be tedious but feasible for data-parallel problems it is much more complex for task-parallel or irregular problems.

Without a suited programming model, programmers have to decide which parts of their program may run in parallel and how the parallel code is scheduled and executed. The core idea of *Cilk* [7] is to offload this to its runtime system and scheduler. *Cilk* extends C and adds a runtime system for task creation/deletion and a work stealing for provably efficient load balancing. One of the main advantages of Cilk over other programming models is its ability to achieve a good speedup also for irregular problems [12]. Although it was originally designed for shared memory systems and although there have been several approaches to port Cilk to distributed shared memory (DSM) systems [2], there are no real attempts to target heterogeneous architectures like the Cell BE.

This paper shows how Cilk can be extended to run efficiently also on heterogeneous multicores and it presents our prototype *CellCilk* for the Cell BE. With CellCilk it is possible to create tasks for the accelerators that are then dynamically scheduled to them. Task creation and scheduling on the PPU work as in Cilk. To cope with the heterogeneity of machines with a host/accelerator design, we introduce a new scheduling mechanism that spawns tasks across different types of cores. Experiments show a good speedup.

2 Extending Cilk to CellCilk

Cilk extends C by only five keywords (`cilk`, `spawn`, `sync`, `inlet`, and `abort`). Cilk2c translates a Cilk program to C source which is then compiled and linked against the Cilk runtime system for execution on a shared memory system. A work stealing scheduler dynamically assigns the tasks to the available cores. We analyze how this idea can be ported to heterogeneous multicores and adapt the keywords and the scheduling strategy where needed. After explaining the Cilk keywords and their extensions, we define the execution model for heterogeneous architectures, followed by a section on the scheduling mechanisms. Then we show garbage collection for distributed data structures and explain which function variants are generated from the CellCilk source.

2.1 Keywords

The most important Cilk keyword[1] is `spawn`, indicating a function call that may be executed in parallel with the calling function. Every function, even a pure C function, can be spawned. A `sync` statement works like a local barrier: Execution waits for all (possibly) concurrently executing children before proceeding. The example code on the left of Fig. 1 shows a Cilk function with return value. Using a return value of a spawned function before a `sync` might cause a race condition.

[1] Functions using one of the Cilk keywords have to be marked with `cilk`.

```
cilk int fib(int n) {
    if (n < 2) {
        return n;
    } else {
        int x, y;
        x = spawn fib(n-1);
        y = spawn fib(n-2);
        sync;
        return (x+y);
    }
}
```

```
cilk int fib(int n) {
    int ret = 0;
    inlet void sum(int result) {
        ret += result;
    }
    if (n < 2) {
        return n;
    } else {
        sum(spawn fib(n-1));
        sum(spawn fib(n-2));
        sync;
        return ret;
    }
}
```

Fig. 1. Fibonacci code as Cilk function; Fibonacci code with inlet

Because the potential of heterogeneous multicores can only be exploited when the processor type is kept in mind, we have changed the semantics of `spawn` so that the spawned function will be executed *on the same core type* (in the CellCilk prototype, either PPU or SPU) as the spawner. In order to invoke a function on an accelerator core, we introduce a new keyword (called `spu_spawn` in CellCilk). We will elaborate on the performance and other reasons for this decision below.

CellCilk's `sync` statement still represents a local barrier for *all* spawned functions, regardless of the type of core they are executing on. Other Cilk features, e. g., inner functions written in pure C (so-called inlets) can also be used on heterogeneous multicores without changes. They enable processing results from spawned functions, e.g. in a reduction. The first argument of such an inlet is the result of a spawned function call but it may have more arguments (but no additional results of spawned functions). Fig. 1 shows on the right a variant of the Fibonacci code that uses inlets. An inlet is executed as soon as the spawned function returns. They are in a way piggy-backed onto the spawned function. However, inlets are executed as an atomic block to avoid race conditions. The keyword `abort` may only be used in conjunction with inlets. It indicates that all other spawned children of the surrounding function should stop their execution.[2] In case of a parallel search in a large search space, the use of `abort` makes sense as soon as one child finds a solution.[3]

On a heterogeneous multicore it is challenging to efficiently implement `inlet` and `abort` because functions may be spawned both to the host and to the accelerator cores. Section 2.5 describes in detail how inlets are generated and executed.

2.2 Execution Model

Each Cilk function consists of so-called *Threads* that are uninterruptible sequences of statements (i. e., a sequence without `spawn` and `sync`). In Fig. 2 they are represented with uppercase letters. The Cilk model guarantees that a single *Thread* is always executed without interruption by other *Threads* of the same function instance and its inlets. Using this model, each Cilk program can be

[2] However, the function may spawn future children after aborting the current children.
[3] Actually, this not an immediate abort but a flag is set at the corresponding workers.

```
cilk a() {
    // A1
    spawn b();
    // A2
    spawn c();
    // A3
    spu_spawn d();
    // A4
    sync;
    // A5
}
```

```
cilk b() {
    // B1
    spawn c();
    // B2
    spu_spawn c();
    // B3
    sync;
    // B4
}
```

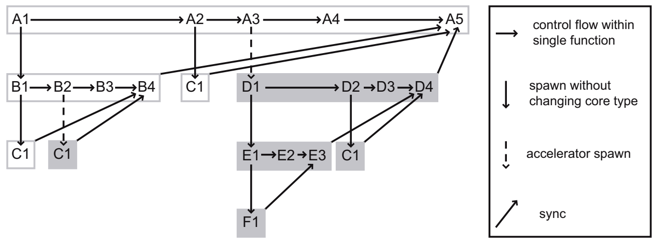
```
c() {
    // C1
}
```

```
cilk d() {
    // D1
    spawn e();
    // D2
    spawn c();
    // D3
    sync;
    // D4
}
```

```
cilk e() {
    // E1
    spawn f();
    // E2
    sync;
    // E3
}
```

```
f() {
    // F1
}
```

(a) Threads of a CellCilk program


A1
A2
A3
A4
A5
B1
B2
B3
B4
C1
D1
D2
D3
D4
C1
C1
E1
E2
E3
C1
F1
control flow within single function
spawn without changing core type
accelerator spawn
sync

(b) Dependency DAG

Fig. 2. Representation of a CellCilk program as a DAG

represented as a directed acyclic graph (DAG) using the *Threads* as nodes and the dependencies between them as edges (see Fig. 2 (b)). This model can be extended for heterogeneous multicores by introducing new types of nodes and edges for accelerator spawns. Horizontal arrows represent the control flow within single functions (inside a rectangle) where shaded rectangles mark functions that are executed on an accelerator. A downward arrow corresponds to a `spawn`, a dashed one to an accelerator spawn, and an upward arrow indicates a `sync`. As will be explained later, the CellCilk scheduler respects those dependencies.

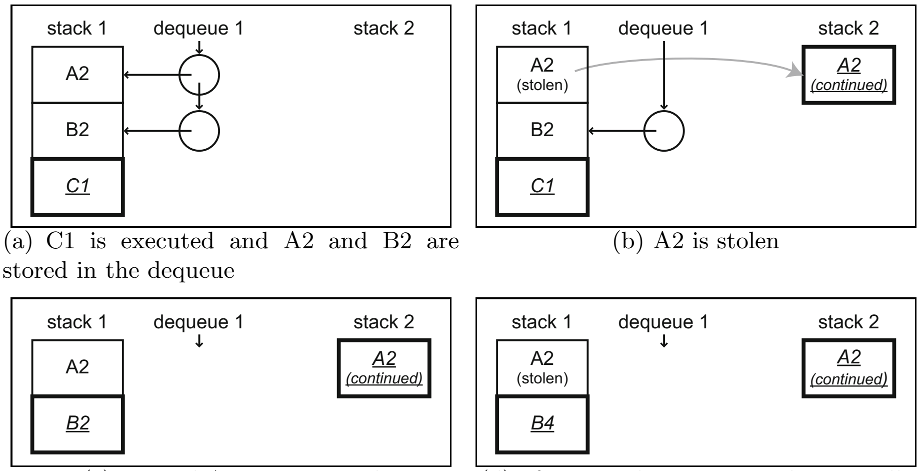
For each of the *Threads*, one could specify execution times. Using those times, the critical path in the dependency DAG, i. e., the path with the longest total execution time can be calculated. As this path gives an lower bound on the total execution time, programmers can use it to guide their tuning efforts.

In ANSI C, a parent function cannot access the local memory of its children or siblings as functions free their local memory (placed on stack) once they return. Whereas ANSI C functions can access the frames of their parents, this is no longer possible in CellCilk because it is neither straightforward nor efficient to implement on heterogeneous multicores.

With the changed semantics of `spawn` and the added `spu_spawn` better efficiency can be achieved – albeit only with the help of the programmer who needs to select the proper `spawn` and sometimes has to give up access to parent frames.

2.3 Scheduling and Work Stealing

For heterogeneous multicores, Cilk's traditional two-level scheduling (called nano and micro scheduling) needs to be extended by a new cross-hierarchy macro scheduling to cope with the different types of cores.



(a) C1 is executed and A2 and B2 are stored in the dequeue

(b) A2 is stolen

(c) B2 and A2 are executed

(d) after completing B4, worker 1 is idle

Fig. 3. Possible micro and nano scheduling of the program in Fig. 2

Every single worker uses nano scheduling to traverse the DAG in depth-first order (see Fig. 2), similar to the way a regular C program is executed. CellCilk uses this method for spawning statements that do not require a change of core type. Cilk's micro scheduling is based on randomized work stealing by means of the THE-protocol [16]. That means that if a worker finishes its current task it selects a randomized worker to steal work from it. CellCilk applies this mechanism as well, but only between workers that run on the same type of core. To do so, whenever the program steps on a `spawn` statement the current state is saved as stack frame and added to the worker's double ended queue (dequeue). In Fig. 3(a), the worker has already finished the execution of both A1 and B1 (cf. Fig. 2). Before it starts executing C1, it adds the current state to the end of its local dequeue. In Fig. 3(b) worker 2 is idle, selects worker 1 as victim, marks the oldest work from the front of the dequeue as stolen, and starts to execute A2. As soon as worker 1 finishes execution of C1, it continues with B2, see Fig. 3(c). After completing B4, worker 1 notices that A1 was stolen (see Fig. 3(d)), stops execution, and tries to steal work from others.

When work is moved from one processor type to another by an accelerator spawn, e. g., a `spu_spawn` on a PPU, there is also a switch from nano scheduling to the new *macro scheduling*, as the micro scheduling mechanisms are no longer applicable. The host worker adds the spawned function to a global first-in first-out (FIFO) spawn queue located in host memory (see Fig. 4(a)). When an accelerator finishes its task, it first searches for a new task in this global FIFO spawn queue before selecting another accelerator as victim for stealing. If the global dequeue holds at least one entry, the oldest entry is removed, transferred to the accelerator, and executed, see Fig. 4(b). This corresponds to the *oldest* work stealing mechanism in micro scheduling. If a spawn is executed on the accelerator the frame is added to the local dequeue, see Fig. 4(c). If the global spawn

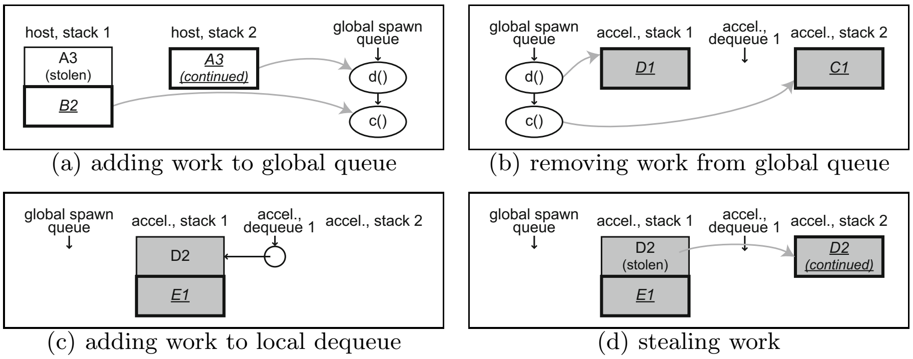


Fig. 4. Possible macro scheduling of the program in Fig. 2

queue is empty an idle accelerator worker switches to micro scheduling and tries to steal work from the local dequeue of another worker (see Fig. 4(d)). As multiple workers (either on the host or on an accelerator) may access the global spawn queue concurrently, the access to the queue has to be synchronized.

Here is another reason for the introduction of a `spu_spawn` instead of trying to build a general purpose `spawn` that abstract from the types of cores: There would be two variants to implement such a type independent `spawn` and both have drawbacks. The first approach is to implement work stealing across different types of cores and across the memory hierarchy. That would cause slow communication. Furthermore, the different compilers for the different cores of the heterogeneous system in general use different struct layouts. Hence, instead of simply copying structs, some elaborate (de)serialization to/from a common exchange format would be needed. The other approach would use a global queue. This would imply even more synchronization so that the advantages of Cilk would be lost because the minimal overhead of a `spawn` that is based on the depth-first traversal of the call graph is crucial for Cilk's performance.

If all Cilk keywords are removed from a Cilk program we get the so-called *serial elision* that has the same behavior as the execution of the Cilk program with one thread. Contrary to Cilk, no *serial elision* of a heterogeneous Cilk program that has the same behavior as a single worker execution can be created by simply deleting all keywords even if the processor type is ignored. This is caused by the specialization in heterogeneous multicores, e. g., on the Cell BE SPU intrinsics are not available on the PPU and DMA transfers work differently. But it is possible to create a serial variant without CellCilk keywords by removing all keywords except the accelerator spawn. This spawn has to be replaced by code that executes an accelerator task while the host is waiting. Thus, the code is executed sequentially without using the CellCilk runtime.

Because of the differences the programmer needs to know which spawned function can run on which core type anyway. Therefore, it would not only be useless to hide the core type behind a general purpose `spawn`. But it is even better to make the differences explicit and to avoid confusion by different types of `spawn`.

2.4 Memory Management on the Accelerator Cores

Concurrent use of global memory easily leads to race conditions. On heterogeneous multicores the situation is even worse as the memory neither between host and accelerator nor between multiple accelerator cores is shared. A global memory would have to be simulated at a high cost, both with respect to runtime as each variable access would cause some slow communication, and with respect to space as a compound data structure would consume precious space on all the small scratchpad memories of the accelerator cores. Therefore there are no global variables in CellCilk.

Only pointers that are relative to the own stack frame or pointers that address the main memory are allowed in CellCilk. The reason is that only those pointers remain valid when a stack frame is stolen and executed on another core. We may integrate a software managed cache as in [17] to circumvent this restriction.

But we cannot define away all memory management problems. Every function has to store a list of all spawned children, e. g., to be able to process them during a `sync` statement. If a function is spawned, a so-called spawn entry is added to the list of children. It stores information about the child, e. g., the currently executing core and whether it is already finished. If a function is executed by different cores the spawn entries of its children are created by different cores. While this does not cause any problem in shared memory environments a distributed list of children is needed on distributed memory systems.

When accelerator 1 in Fig. 5 starts to execute function `d()` the function's prologue creates the list of children in the scratchpad memory of that accelerator. When function `e()` is spawned, a spawn entry for `e()` is created there as well. If later on accelerator 2 steals D2, a copy of the function frame is created in the scratchpad memory of accelerator 2. But the children list of `d()` and the spawn entry for `e()` have to remain in the other scratchpad memory until `d()` is finished even if `e()` is finished earlier and accelerator 1 starts searching for new work. If `d()` spawns another function `c()` on accelerator 2 the corresponding spawn entry is created there and added to the distributed children list. If later on `d()` runs into a local barrier on accelerator 2 (i. e., a `sync`), the whole distributed list of children has to be processed. After that all spawn entries can be freed. But a core can only free its own scratchpad memory. So the elements in the scratchpad memory of accelerator 1 cannot be freed. Even worse, it is unknown to accelerator 1 when the data structures can be freed.

To solve this problem, we integrated a garbage collection mechanism based on a mark and sweep collector [11]. The CellCilk runtime registers dynamically allocated structures (as spawn entries and children lists) for memory management. If the memory can be freed, it is remotely marked as "free"-able via a simple runtime call. The runtime scans the list of registered structures from time to time and sweeps all marked memory chunks. We found that it is enough to free

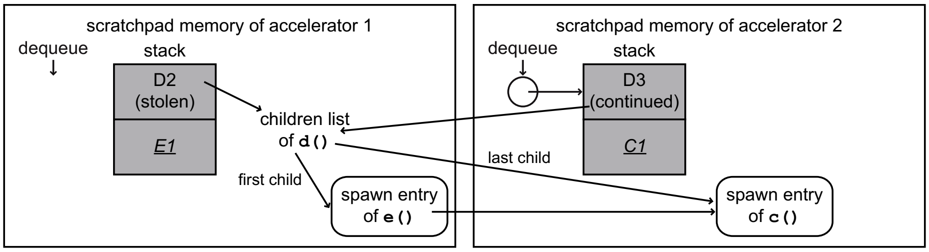


Fig. 5. Illustration why garbage collection is necessary

memory while a processor is searching for new work instead of interrupting the execution for garbage collection.

An alternative to the distributed children list would be to move the children when the corresponding function is stolen. This would not only introduce the copying overhead but would also require changes to addresses in the spawn entries. As spawn entries are also referenced from function frames, these would have to be updated causing the need for synchronization. Thus, we found a distributed children list plus a simple garbage collector to be preferable.

2.5 CellCilk Code Generation and Function Variants

One of the drawbacks of programming heterogeneous multicores with a host/accelerator design is the complex programming, as the programmer has to write the source code for the PPU and SPU in different files (at least for the Cell BE). With CellCilk this restriction no longer applies as a programmer only has to annotate a function call with `spu_spawn`. Fig. 6 shows how our source-to-source compiler CellCilk2C generates separate files for host (PPU) and accelerator (SPU) from original CellCilk source. As the accelerator cores typically are only equipped with a small scratchpad memory for code and data the source code (at least) for the accelerator should be as small as possible. To do so, CellCilk2C analyzes the call graph starting from the `main` function, determines (at least within the limits set by aliases and function pointers) that certain functions will never be called on a certain type of core, and skips generating code for them.

Ordinary (non-Cilk) C functions that are not called across processor type boundaries are simply copied to the generated source without any changes. The spawning functions require more work. The overhead for task creation, work stealing, and function continuation can be minimized by using function variants that are specialized for their use cases. CellCilk2C generates two function variants (slow, fast) for the host and three function variants (slow, fast, host-call) for the accelerator cores. The fast variant is used until it is stolen, the slow variant is executed when stolen and provides several continuation points. The host-call-variant is used to switch from host to accelerator.

The fast variant pretty much looks like a regular C method. It has a very low overhead compared to plain C, is executed until the corresponding *Thread*

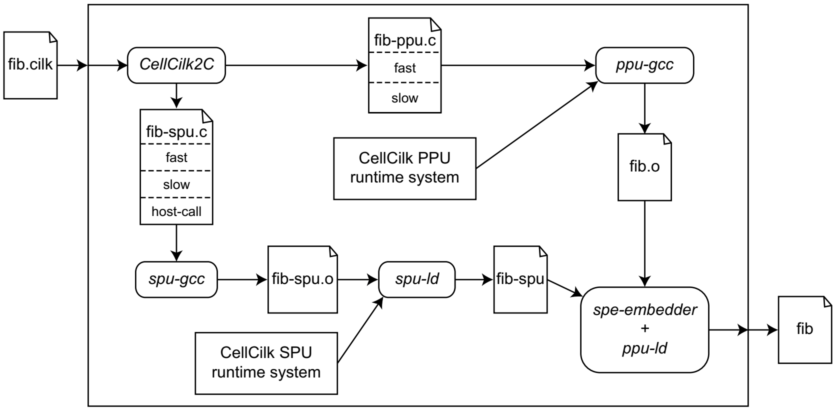


Fig. 6. Translation from CellCilk to C and compilation for PPU and SPU

is stolen, and is only used in nano scheduling. It differs from plain C only in dequeue administration. A slow variant is executed from a stack frame stored on the dequeue. To be able to interrupt and later continue its execution, the stack frame also holds the current position.[4] The code variant looks like a special form of Duff's Device [6] that uses the current position in an initial switch to jump to the statement/position where the function should be continued. The slow version is hence a form of a closure object.

For the SPUs, we create an additional host-call variant for every accelerator function that is potentially called directly from the host. Due to the change of architectures and the disjunct memories, arguments cannot be passed through the stack. Instead, the host-call variant uses two extra parameters: the SPU process information and a pointer to a structure holding the arguments. The host-call variant first applies a DMA transfer to copy this structure to the corresponding worker and then calls the fast variant. As there are several restrictions on the size and boundary of DMA transfers on the Cell BE, the structure is padded with dummy bytes if necessary. This idea is also used for the return value.

CellCilk uses internally function pointers to know which function has to be executed or continued. Therefore, an array of function pointers is created for the host-call variant. Because function pointers created on the host are not valid on the accelerators, there is also a mapping table.

The fast versions can ignore `sync` since the function is executed sequentially (without stealing), there cannot be any children to wait for. Otherwise, the slow variant would have been executed that waits for all children to be finished. For this, we use condition variables in order to avoid busy waits. If a slow function reaches a `sync` two scenarios are possible: if there is still at least one outstanding child on the same processor type the execution of the function is suspended (and

[4] In addition to the usual values, a CellCilk stack frame holds: the size of the frame, the current code position, a list of (spawned) children, a stolen-flag, references to the parent function as well as to the function variants.

other work will be executed if possible). If all children are done, the suspended function can be continued.

An inlet can be executed in Cilk like in C if the parent frame was not stolen because it is then executed sequentially. Thus, no race condition can occur and the atomicity of the inlet is ensured. But if the parent frame was stolen the inlet is executed during the `sync` statement. This holds true in CellCilk for `spawn` statements without processor type change as well. If an inlet contains a `spu_spawn` it can not be executed after returning from the child function because there is always parallelism that could cause race conditions after the execution of an accelerator spawn . So inlets containing an `spu_spawn` will always be executed during the `sync` statement. `abort` (within an inlet) is handled through the workers' dequeue. On the host a flag of the dequeue can be set. In CellCilk, we use the mailbox mechanism to abort the work of a SPU from the PPU. The SPUs can use DMA transfers for setting the abort flag of each other's dequeues. All dequeue functions check if an abort signal has been received. As `sync`, the `abort` statement cannot be ignored in fast variants containing a `spu_spawn` because children may still run on the accelerator.

3 Performance Evaluation

We have evaluated CellCilk with several benchmarks using the C compiler from the Gnu Compiler Collection. We have used a Playstation 3 utilizing the Cell BE equipped with one PPU with two-way hyperthreading and with 6 SPUs.

First, we have tested our CellCilk prototype with three small standard examples in different implementations. The first example is the computation of the Fibonacci function similar to the code shown above.[5] The second example is based on the 0/1-Knapsack problem. It also does not use memoization but recursively searches for the best solution starting with the most valued item. The currently best solution is stored in a variable located in main memory. Each possibility is tried using two spawns (with and without current item). The search tree is pruned by calculating an upper bound using the global best value.[6] For both examples (Fibonacci and Knapsack), we defined a threshold – below this threshold their values are computed without further spawning. Otherwise, the spawned functions are too fine-grained (only few instructions) and the overhead of task creation would be too high. The third example is an implementation of the discrete Fourier transformation (DFT) that uses more floating point computations than the other examples.

Fig. 7(a) compares the runtimes of the CellCilk examples with up to two PPU and up to six SPU threads with the execution time of ANSI C. The overhead of CellCilk is moderate when executed with a single PPU thread (at most 2%). With two or more PPU threads, only the DFT benefits because of the heavy computational part. The timings of the other example programs do not change

[5] Note: this code does not do memoization, values are computed multiple times.

[6] The race condition due to the unsynchronized access to the global best value does not affect the final result. It only delays pruning.

as they mostly switch tasks. For Fibonacci (5.93) and DFT (5.96) the speedup is almost linear on 6 SPUs. For Knapsack the speedup is superscalar. Although Knapsack is quite slow on a single SPU, it gets much faster when executed with more threads (speedup up to 22.17). This is caused by the pruning technique mentioned above. The shared best value is updated by one thread while the others can prematurely prune their search trees.

Furthermore, we have evaluated the performance of CellCilk with more "real-world" benchmarks. In addition to the mentioned DFT example, we have written two recursive sorting algorithms (merge sort and bitonic sort), matrix multiplication, and the lattice boltzmann method (LBM) on a 2D grid. The work in the DFT is distributed over the SPE threads using geometrical decomposition. The sorting algorithms are inherently recursive, the recursive functions are annotated with `spu_spawn` and executed as CellCilk tasks. Matrix multiplication and LBM are cache oblivious variants [8,9,19] that are recursive and hence allow simple parallelization with CellCilk.

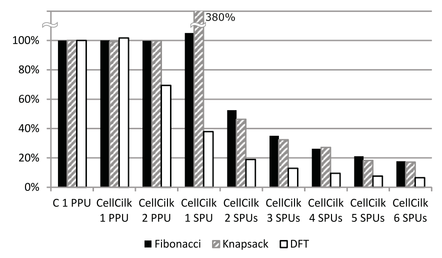
Fig. 7(b) shows the almost linear speedup of DFT, bitonic sort, and LBM. We achieved still a good speedup with matrix multiplication while it is slightly worse for mergesort because the last merge step has to be done sequentially. Bitonic sort is a fully parallelizable variant of the merge sort idea and achieves a speedup of 5.85 on 6 threads.

We also compare the matrix multiplication and LBM with implementations using CellSs [1] and OpenMP with the xlc compiler; each of these implementations is optimized towards the used programming model. In addition to a manually written and hand-optimized matrix multiplication in ANSI C, and two variants using OpenMP resp. CellSs, we also compare our CellCilk implementation to a "fast matrix multiplication" [10] written in highly optimized assembly code. As can be seen in Fig. 7(c), this version (TU-DD) has the best speedup of course. Both the ANSI C and our CellCilk version achieve good speedups that are much better than what the OpenMP and the CellSs version deliver (below 2). Fig. 7(d) gives the absolute run times. As OpenMP and CellSs are very slow (66 resp. 13 seconds with 6 SPU threads), we did not include them in the chart. Of course, the runtime of the assembly is better than the CellCilk version but we have slightly outperformed the ANSI C implementation. This is probably due to the good caching strategy of the cache oblivious algorithm.

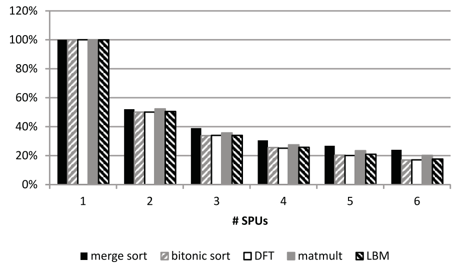
For LBM Fig. 7(e) shows that both the ANSI C as well as the CellCilk implementation scale nicely, up to a speedup of almost 6 on 6 SPUs. In contrast, the speedup of CellSs is below 3 and OpenMP cannot benefit from the parallel cores at all. In absolute run times, CellCilk is only 2-3% slower than the C variant, see Fig. 7(f). Again, the runtimes of CellSs and OpenMP are too slow (89 resp. 34 seconds with 6 threads) to be included in the chart.

4 Related Work

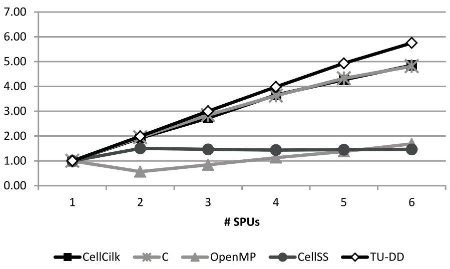
Peng et al. implemented SilkRoad [15] (based on distributed Cilk [16]) and tested several memory consistency models for distributed shared memory systems. SilkRoad provides a shared memory to the programmer and keeps this



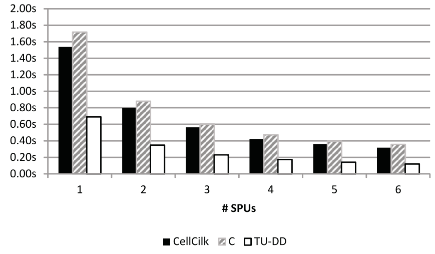
(a) Runtime of CellCilk example programs on PPU/SPU compared to versions in plain C = 100%.



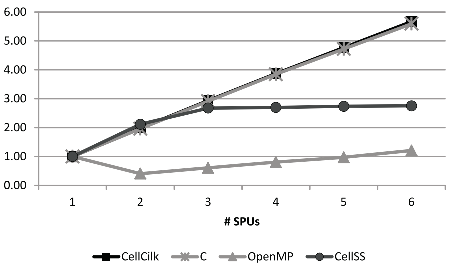
(b) Runtime of several CellCilk benchmark programs on the SPUs compared the runtime on a single SPU = 100%.



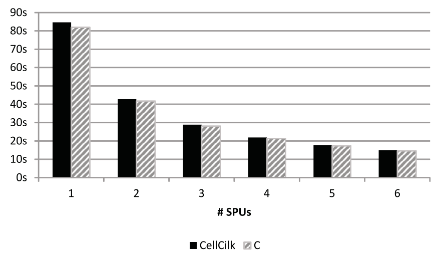
(c) Speedup of matrix multiplication.



(d) Runtime of matrix multiplication.



(e) Speedup of LBM.



(f) Runtime of LBM.

Fig. 7. Performance evaluation of the CellCilk prototype on a Cell BE

memory consistent by creating and copying memory diffs if necessary. Due to its messaging overhead, SilkRoad is highly dependent on data locality of the application. CellCilk outperforms SilkRoad and achieves better speedups. Cl<u>i</u>k [13] proposes an improvement over SilkRoad with two different policies for selecting the victim node. Instead of choosing a random victim it increases data locality while decreasing message overhead by selecting the most overloaded or the last selected node first. These policies may accelerate CellCilk as well. Dinan et al. [5] show how scalable work stealing can be done on large-scale, non-heterogeneous systems by reducing the number of synchronized operations on work queues and a modified stealing scheme where not only one task is stolen.

Offload [4] extends C++ so that code snippets can be offloaded to accelerators. The system distinguishes between pointers to local and host memory and can type-check pointer assignments. Furthermore, if values are assigned from/to a host pointer, they may be automatically fetched via DMA (possibly employing double buffering / software caches). As the approach does not focus on parallelization, it is orthogonal to CellCilk and may be included in future work.

Several approaches try to simplify the complexity of parallel programming for heterogeneous multicores. In the CellSs framework [1] the programmer annotates functions as tasks. These tasks are ordered in a dependency graph at runtime. Subgraphs are partitioned and scheduled over the SPUs. As too fine grained tasks (that may be created by recursive functions) lead to a large task dependency graph that has too much overhead for the PPU, CellSs prefers coarse grained tasks. CellCilk beats CellSs in both runtime and speedup in our benchmarks.

OpenMP [14] does work sharing basically by annotating loops. These loops are generated into functions that are executed in parallel on the PPU and the SPUs. This model works efficiently for data-parallel applications but has its drawbacks for task-parallelism. Recently, Cao et al. proposed an extension for OpenMP tasks[7] on the Cell BE [3]. They use explicitly managed local queues on the SPUs and a global queue. Scheduling is done with two different strategies, one is efficient for irregular applications, while the other is designed for applications with a good data locality. As CellCilk has a very low overhead for task creation in fast function variants. Task creation in OpenMP is more explicit. Thus, if all cores are busy doing their own work, the estimated overhead in the OpenMP tasking model slows down the execution compared to CellCilk.

5 Conclusions

We presented CellCilk, a programming model for heterogeneous multicores with a host/accelerator design. CellCilk is based on Cilk and introduces a new keyword that enables the programmer to write a parallel program in a single source file using annotations (`spawn` resp. `spu_spawn`). Our source-to-source compiler CellCilk2C generates all necessary function variants for the different cores including one variant that is used for spawns from host to accelerator.

The Cilk scheduling mechanism and runtime system were extended to cope with all challenges of a heterogeneous architecture, e. g., distributed memory. The CellCilk runtime schedules dynamically the spawned tasks over the available host resp. accelerator threads. Load balancing in-between the workers of one processor type is done by stealing the oldest work from a random victim. The newly introduced macro scheduling based on a global FIFO queue takes care of work that is scheduled on the host to be executed on an accelerator. The runtime system requires a (mark and sweep) garbage collector that takes care of distributed data structures.

As benchmarks on a Cell BE show, CellCilk brings the advantages and efficiency of Cilk to heterogeneous architectures and scales almost linearly when

[7] Their implementation does not obey the OpenMP 3.0 task specification.

the SPUs are used. As the CellCilk runtime does dynamic load balancing, CellCilk on a heterogeneous multicore is well suited for parallelizing irregular applications as well as geometrically decomposed applications for heterogeneous platforms. Data-parallel applications may be parallelized more efficiently using a data-parallel programming model. The benchmarks further show that it is possible to achieve reasonable runtimes using CellCilk especially if compared to other programming models such as OpenMP and CellSs.

6 Future Work

CellCilk may be further improved in some areas. As mentioned before, CellCilk2C analyzes the call graph and detects which functions are only executed on the host. These functions are not included in the SPU binary to save precious scratchpad memory. However, code garbage collection techniques as in [18] can be used to further reduce the code size.

Currently, CellCilk packs function arguments and local variables into a structure that is copied when the function is stolen. This structure may be split into several parts that are needed at different points (possible entry points after a (`spu_`)`spawn`). With help of live variable analysis we can determine which variables to place into which part to further reduce memory consumption.

In several applications most of the functions access most of the memory in a read-only manner (e.g. the items in the Knapsack problem). A programmer would make these data globally available when not using CellCilk. We plan to extend CellCilk with processor type global variables that are copied to the corresponding cores if necessary.

Acknowledgements. This project has been supported in part by the Embedded Systems Institute (ESI) and the ESI-Anwendungszentrum, `http://esi-anwendungszentrum.de/`.

References

1. Bellens, P., Pérez, J.M., Cabarcas, F., Ramírez, A., Badia, R.M., Labarta, J.: CellSs: Scheduling techniques to better exploit memory hierarchy. Scientific Programming 17(1-2), 77–95 (2009)
2. Blumofe, R.D., Frigo, M., Joerg, C.F., Leiserson, C.E., Randall, K.H.: An Analysis of Dag-Consistent Distributed Shared-Memory Algorithms. In: SPAA 1996: Proc. Symp. Parallel Algorithms and Architectures, Padua, Italy, pp. 297–308 (June 1996)
3. Cao, Q., Hu, C., He, H., Huang, X., Li, S.: Support for OpenMP Tasks on Cell Architecture. In: Hsu, C.-H., Yang, L.T., Park, J.H., Yeo, S.-S. (eds.) ICA3PP 2010, Part II. LNCS, vol. 6082, pp. 308–317. Springer, Heidelberg (2010)
4. Cooper, P., Dolinsky, U., Donaldson, A.F., Richards, A., Riley, C., Russell, G.: Offload – Automating Code Migration to Heterogeneous Multicore Systems. In: Patt, Y.N., Foglia, P., Duesterwald, E., Faraboschi, P., Martorell, X. (eds.) HiPEAC 2010. LNCS, vol. 5952, pp. 337–352. Springer, Heidelberg (2010)

5. Dinan, J., Larkins, D.B., Sadayappan, P., Krishnamoorthy, S., Nieplocha, J.: Scalable work stealing. In: SC 2009: Proc. Conf. High Performance Computing Networking, Storage and Analysis, Portland, OR, pp. 53:1–53:11. ACM (November 2009)
6. Duff, T.: Duff's device. Usenet posting (November 1983), http://www.lysator.liu.se/c/duffs-device.html
7. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. In: PLDI 1998: Proc. Conf. Programming Language Design and Impl., Montreal, Canada, pp. 212–223 (June 1998)
8. Frigo, M., Strumpen, V.: Cache oblivious stencil computations. In: ICS 2005: Proc. Intl. Conf. Supercomputing, Cambridge, MA, pp. 361–366 (June 2005)
9. Frigo, M., Strumpen, V.: The cache complexity of multithreaded cache oblivious algorithms. In: SPAA 2006: Proc. Symp. Parallel Algorithms and Architectures, Cambridge, MA, pp. 271–280 (July 2006)
10. Hackenberg, D.: Fast Matrix Multiplication on Cell (SMP) Systems (2009), http://www.tu-dresden.de/zih/cell/matmul
11. Jones, R., Lins, R.: Garbage Collection: Algorithms for Automatic Dynamic Memory Management. Wiley & Sons (1996)
12. Leiserson, C.E.: Programming Irregular Parallel Applications in Cilk. In: Lüling, R., Bilardi, G., Ferreira, A., Rolim, J.D.P. (eds.) IRREGULAR 1997. LNCS, vol. 1253, pp. 61–71. Springer, Heidelberg (1997)
13. Mendes, R., Whately, L., de Castro, M.C.S., Bentes, C., de Amorim, C.L.: Runtime System Support for Running Applications with Dynamic and Asynchronous Task Parallelism in Software DSM Systems. In: SBAC-PAD 2006: Symp. Computer Architecture and High Performance Computing, Ouro Preto, Brasil, pp. 159–166 (October 2006)
14. O'Brien, K., O'Brien, K., Sura, Z., Chen, T., Zhang, T.: Supporting OpenMP on Cell. In: Chapman, B., Zheng, W., Gao, G.R., Sato, M., Ayguadé, E., Wang, D. (eds.) IWOMP 2007. LNCS, vol. 4935, pp. 65–76. Springer, Heidelberg (2008)
15. Peng, L., Wong, W.F., Yuen, C.K.: The Performance Model of SilkRoad - A Multithreaded DSM System for Clusters. In: CCGRID 2003: Intl. Symp. Cluster Computing and the Grid, Tokyo, Japan, pp. 495–501 (May 2003)
16. Randall, K.H.: Cilk: Efficient Multithreaded Computing. Ph.D. thesis, Massachusetts Institute of Technology (June 1998)
17. Seo, S., Lee, J., Sura, Z.: Design and implementation of software-managed caches for multicores with local memory. In: HPCA 2009: Intl. Conf. High-Performance Computer Architecture, Raleigh, NC, pp. 55–66. IEEE (February 2009)
18. Werth, T., Floßmann, T., Klemm, M., Schell, D., Weigand, U., Philippsen, M.: Dynamic Code Footprint Optimization for the IBM Cell Broadband Engine. In: IWMSE 2009: Proc. ICSE Workshop on Multicore Software Engineering, Vancouver, Canada, pp. 64–72 (May 2009)
19. Zeiser, T., Wellein, G., Iglberger, K., Nitsure, A., Rüde, U., Hager, G.: Introducing a parallel cache oblivious blocking approach for the Lattice Boltzmann Method. Progress in Computational Fluid Dynamics 8(1-4), 179–188 (2008)

OPELL and PM: A Case Study on Porting Shared Memory Programming Models to Accelerators Architectures

Joseph B. Manzano, Ge Gan, Juergen Ributzka, Sunil Shrestha, and Guang R. Gao

Department of Electrical and Computer Engineering
University of Delaware, USA
{jmanzano,gan,ggao}@capsl.udel.edu,
{ributzka,sunilaachaju}@gmail.com

Abstract. Limits on applications and hardware technologies have put a stop to the frequency race during the 2000s. Designs now can be divided into homogeneous and heterogeneous ones. Homogeneous types are the easiest to use since most toolchains and system software do not need too much of a rewrite. On the other end of the spectrum, there are the type two heterogeneous designs. These designs offer tremendous computational raw power, but at the cost of hardware features that might be necessary or even essential for certain types of system software and programming languages. An example of this architectural design is the Cell processor which exhibits both a heavy core and a group of simple cores designed as a computational engine. Even though the Cell processor is very well known for its accomplishments, it is also well known for its low programmability. Among many efforts to increase its programmability, there is the Open OPELL project. This framework tries to port the OpenMP programming model to the Cell architecture. The OPELL framework is composed of four components: a single source toolchain, a very light SPU kernel, a software cache and a partition / code overlay manager. To reduce the overhead, each of these components can be further optimized. This paper concentrates on optimizing the partition manager by reducing the number of long latency transactions. The contributions of this work are as follows.

1. The development of a dynamic framework that loads and manages partitions across function calls to bypass the problem with restrictive memory spaces.
2. The implementation of replacement policies that are useful to reduce the number of DMA calls across partitions.
3. A quantification of such replacement policies given a selected set of applications
4. An API which can be easily ported and extended to several types of architectures.

1 Introduction

During this decade, the multi / many core architectures have seen a renaissance, due to the insatiable hunger for performance. Limits on applications and hardware technologies have put a stop to the frequency race around 2006. Designs now can be divided into homogeneous and heterogeneous ones. Homogeneous designs are the easiest to use since most toolchain and system software do not need too much of a rewrite. On the other end of the spectrum, there are heterogeneous designs. These designs offer tremendous computational raw power, but at the cost of hardware features that might be necessary or even essential for certain types of system software and programming languages. An example of this architectural design is the Cell processor which will be explained in the next section.

1.1 The Cell Broadband Engine

The Cell B.E. has been placed in the public eye thanks to being a central component in one of the fastest super computer, being the main processing unit of the Sony's Playstation 3 videogame console, and being the bane of programmers everywhere. This architecture is a project in which three of the big computer / entertainment companies, IBM, Sony and Toshiba, worked together to create a new chip for the seventh generation of home video game consoles[1]. The chip possesses a heavy core, called the PowerPC Processing Element (or PPE for short), which acts as the system's brain. The workers for the chip are called the Synergistic Processing Elements (or SPE for short) which are modified vector architectures which huge computational power. The SPE possesses 256 KiB of local memory and a Memory Flow Controller which takes care of external Input / Output operations.

Both processing elements coexist on the die with a ratio of 1 to 8 (one PPE to eight SPEs), but more configurations are possible. Finally, all the components are interconnected by a four-ring bus called the Element Interconnect Bus (or EIB for short). Figure 1 shows a high level overview of the Cell B.E. This chip is capable of around 200 Giga Floating Point Operations Per Seconds (FLOPS) for single precision and around 102.4 Giga FLOPS for double precision[1].

Although the heavy core possesses all the "standard" hardware components, the computational engine lacks many of these features. The SPEs exhibit limited local memory, lack caches of any type, and it has no virtual memory support. Communication between the host (PPE) and the computational engine (the SPEs) is achieved through explicit Direct Memory Access (DMA) operations between the main memory and the local memory of the computational engine. This puts more responsibilities on the system software, programmers and users to take advantage of the system raw computational power by orchestrating all components using the features of the computational engine.

[1] These numbers come from the revised PowerXCell 8i Boards.

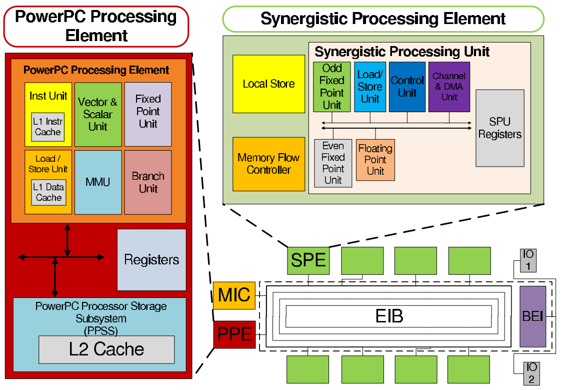


Fig. 1. Block Diagram of the Cell Broadband engine

1.2 Problem Formulation

The lack of programmability in the heterogeneous designs, especially in the Cell B.E., can be attributed to the loss of many hardware features, such as caches, reorder buffers, etc and a lack of runtime systems to take advantage of the architecture's performance. The need for new software stacks is evident. Due to this need, the OPELL framework was introduced. This framework tries to bring the OpenMP parallel programming model (De facto shared memory parallel programming paradigm) to the Cell architecture. The OPELL framework is composed of four components: a single source toolchain, a very light SPU kernel, a software cache and a partition / code overlay manager. This extra layer greatly increases the system's programmability, but it comes at the cost of additional overhead from the framework. To reduce the overhead, each of the components can be further optimized. This paper concentrates on optimizing the partition manager components by reducing the number of long latency transactions (DMA operations) that it produces. The contribution of this paper can be summarized as follows:

1. The development of a dynamic framework that loads and manages partitions across function calls. In this manner, the restrictive memory problem can be alleviated and the range of applications that can be run on the co-processing unit is expanded.
2. The implementation of replacement policies that are useful to reduce the number of DMA calls across partitions. Such replacement policies aim to optimize the most costly operations in the proposed framework. Such replacements can be of the form of buffer divisions, rules about eviction and loading, etc.
3. A quantification of such replacement policies given a selected set of applications and a report of the overhead of such policies. Several policies can be given but a quantitative study is necessary to analyze which policy is best in which application since the code can have different behaviors.

4. An API which can be easily ported and extended to several types of architectures. The problem of restricted space is not going away. The new trend seems to favor an increasing number of cores (with local memories) instead of more hardware features and heavy system software. This means that frameworks like the one proposed in this paper will become more and more important as the wave of multi / many core continues its ascent. Moreover, the same concepts presented here can be extended to run on other heterogeneous accelerator type architecture like GPGPUs and FPGAs.

This paper is divided as follows. Section 2 introduces relevant related work. Section 3 introduces the OPELL framework and each of its components. Section 4 shows the partition manager framework and its features. Section 5 presents the results for the partition manager different features. Finally, Section 7 shows the conclusions and future work.

2 Related Work

There have been many attempts to increase the programmability in the Cell B.E. The most famous ones are the ALF and DaCS[3] frameworks and the CellSS project[2]. The ALF and DaCS frameworks are designed to facilitate the creation of tasks and data communication respectively for the Cell B.E. The Accelerator Library Framework (ALF) is designed to provide a user-level programming framework for people developing for the Cell Architecture. It takes care of many low level approaches (like data transfers, task management, data layout communication, etc). The DaCS framework provides support for process management, accelerator topology services and several data movement schemas. It is designed to provide a higher abstraction to the DMA engine communication. Both frameworks can work together and they are definitely a step forward from the original Cell B.E. primitives. They are not targeted to Cell B.E. application programmers, but to library creators. Thus, the frameworks are designed to be lower level than expected for an OpenMP programmer.

The Cell SuperScalar project (the CellSS) [2] is designed to automatically exploit the function parallelism of a sequential program and distribute them across the Cell B.E. architecture. It accomplishes this with a set of pragma based directives. It has a locality aware scheduler to better utilize the memory spaces. It uses a very similar approach as OpenMP. However, it is restricted to task level parallelism in comparison to OpenMP that can handle data level parallelism. Under our framework, the parallel functions are analogous to CellSS tasks and the partition manager is their scheduler. Many of the required attributes of the tasks under CellSS are hidden by the OpenMP directives and pragmas which make them more programmable.

Finally, there have been efforts to port OpenMP to the Cell B.E.. The most successful one is the implementation in IBM's XL compiler[7]. The implementation under the XL compiler is analogous to the OPELL implementation with very important differences. The software cache under the XL compiler is not configurable with respect to the number of dirty bytes that can be monitored in the line. This allows the implementation of novel memory models and frameworks as shown in [4]. The other difference is that the partition manager under the XL uses static GCC like overlays. Under OPELL, the partitions can be dynamically loaded anywhere in the memory which is not possible under the XL compiler.

3 Opell Framework

The Open Source OpenMP on CELL (or Open OPELL for short) developed at the University of Delaware [6] is a porting of a very popular high performance parallel language to a heterogeneous accelerator type architecture. Its main objective is to provide an open source OpenMP framework for the Cell B.E. architecture. It is composed of an OpenMP toolchain, which produces Cell B.E. code from a single OpenMP source tree; and a runtime that hides the heterogeneity of the architecture from the user. The framework provides the following features: a single source compiler, a simple micro kernel, software cache, and the partition / overlay manager.

3.1 Single Source Compilation

The Cell B.E uses two distinct toolchains to compile its code for the architecture. This adds more complications to an already complex programming environment. In OPELL, the OpenMP source code is read by the driver program. The driver clones the OpenMP source for both toolchains and calls the respective compiler to do the work. The PPU compiler continues as expected, even creating a copy of the parallel function (which is the body of the parallel region in OpenMP) and inserting the appropriate OpenMP runtime function calls when needed. The SPU compiler has a different set of jobs. First, it keeps the parallel functions and discards the serial part of the source code. Second, it inserts calls to the SPU execution handler and its framework to handle the parallel calls and OpenMP runtime calls. Third, it inserts any extra function calls necessary to keep the semantics of the program. Finally, it creates any structures needed for the other components of the runtime system, links the correct libraries and generates the binary. After this step is completed, the control returns to the driver which merges both executables into a single one. Figure 2 shows a high level graphical overview of the whole single source process.

3.2 Simple Execution Handler

This small piece of code[2] deals with the communication between the PPU and SPU during runtime and how runtime and parallel function calls are handled. Since each of the SPUs have very limited memory, it is in everybody best interest to keep the SPU threads very light. To achieve this, the SPU thread will be

[2] In this paper, the terms simple execution handler and SPU micro kernel will be used interchangeably.

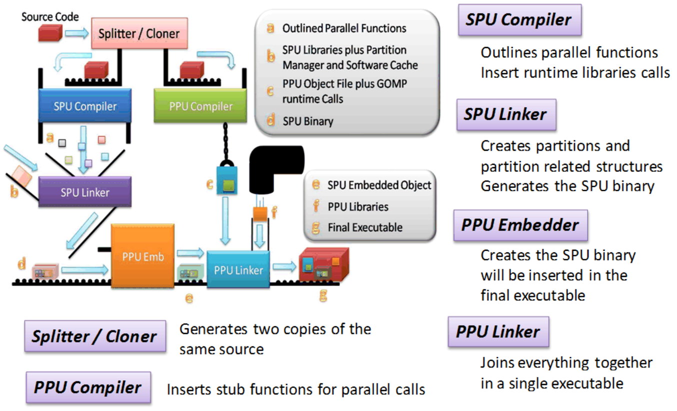
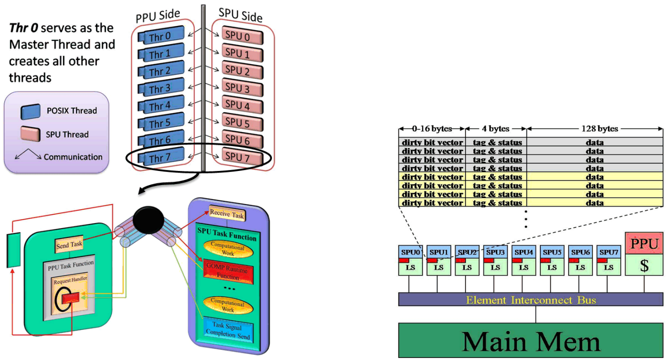

Source Code
Splitter / Cloner
SPU Compiler
PPU Compiler
SPU Linker
PPU Emb
PPU Linker
a Outlined Parallel Functions
b SPU Libraries plus Partition Manager and Software Cache
c PPU Object File plus GOMP runtime Calls
d SPU Binary
e SPU Embedded Object
f PPU Libraries
g Final Executable
SPU Compiler
Outlines parallel functions
Insert runtime libraries calls
SPU Linker
Creates partitions and partition related structures
Generates the SPU binary
PPU Embedder
Creates the SPU binary will be inserted in the final executable
PPU Linker
Joins everything together in a single executable
Splitter / Cloner
Generates two copies of the same source
PPU Compiler
Inserts stub functions for parallel calls

Fig. 2. A high level overview of the single source toolchain. Under this framework the SPU Embedder will "generate" a new SPU binary (i.e it wraps it with a special API) so it can communicate with the host.

loaded only with a minimal set of the code (the simple execution handler and a set of libraries). This SPU resident code does not include the parallel regions of the OpenMP code nor the OpenMP runtime libraries. Since both are needed during runtime, they are both loaded or executed on demand, but by different mechanisms. The parallel regions are loaded and executed by another component, i.e. the partition manager, which loads and overlays code transparently. The OpenMP runtime libraries require another framework to execute. Under this framework, there exists an extra command buffer per thread that is used to communicate between the SPE and PPE frameworks. Moreover, there exists a complementary PPE thread for each SPE thread which is called the mirror or shadow threads which services all the requests from its SPE.

When a SPE thread is created[3], the simple execution handler starts and goes immediately to polling. When a parallel region is found by the master thread (which runs on the PPE), a message is sent to the simple execution handler with the identifier's ID and its arguments' address. When it is received, the SPU calls the code in the parallel region (through the partition manager). The SPU continues executing the code, until an OpenMP runtime call is found. In the SPU, this call creates a PPU request to the command buffer. This request is composed of the operation type (e.g. limit calculations for iteration space) and its arguments. While the SPU waits for the results, the PPU calls the runtime function and calculates the results. The PPU saves the results back to the Command buffer and sends a signal to the SPE to continue. Finally the SPU receives the signal and reads the results. The SPU thread ends polling when the PPU shadow thread sends a self terminate signal, effectively ending the thread's life. Figure 3a shows a graphical representation of the SPE micro kernel and communication framework.

[3] Which happens before the application code is run.



(a) A high level overview of the OPELL runtime

(b) A high level overview of the Software cache structure

Fig. 3. Components of the Simple Execution handler and the Software cache

3.3 Software Cache

As stated before, the SPU component of the Cell B.E. does not have caches (at least not across the SPU local storages) or any other way to maintain coherence. This presents a peculiar problem for the pseudo shared memory which Open OPELL presents[4]. This heterogeneity hindrance is resolved by the software cache. This framework component is designed to work like a normal hardware cache with the following characteristics. It has 64 sets with 4-way associativity and a cache line of 128 bytes (most efficient size for DMA transfers). Its total size is 32 KiB and it has a write back and write allocate update policy. As a normal cache, each line possesses a dirty-bit vector which keeps track of the modified bytes of the line. When the effective (global) address is found in the cache, a hit is produced and the operation is performed, i.e. read or write.

In case that the effective address is not in the cache, a miss is produced. A read miss or write miss causes an atomic DMA operation to be issued to load the desired value from memory and may produce a write back operation if any of the bits in the dirty bit vector are set. The write process only touches the dirty bytes and leaves the clean ones untouched. A graphical overview of the software cache is presented by figure 3b.

This component has been used in the testing and creation of weak memory models presented in [4].

3.4 Overlay / Partition Manager

As the software cache is used for data, the partition manager is used for code. This small component is designed to load code on demand and manage the code

[4] Open OPELL is designed to support OpenMP which is a shared memory programming model.

overlay space when needed. When compiling the source code, certain functions are selected to be partitioned (not loaded with the original source code in the SPU memory). The criteria to select these functions are based on the Function Call Graph, their size and their runtime purpose, e.g. like parallel regions in OpenMP. Finally, the partitions are created, descriptive structures are formed and special calling code is inserted when appropriate. During runtime, the function call proceeds as usual (i.e. save registers, load parameters, etc), up to the point of the actual call. Instead of jumping to the function, the control is given to the partition manager runtime and several decoding steps are done. With information extracted from the actual symbol address, a loading decision is made and the code is loaded into memory or not (if the code already resides in the overlay). Then, the partition manager runtime passes control to the function. When the function finishes, the control returns to the partition manager so any cleaning task can be performed, like loading the caller partition if it was previously evicted. Finally, the partition manager returns to its caller without leaving any trace of its activities.

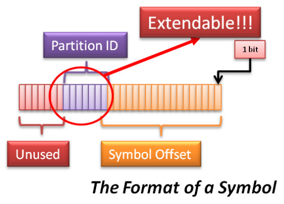
A more detailed description of this component is given in the next section.

4 The Partition Manager

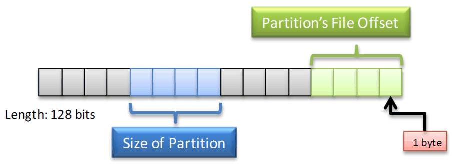
The Partition Manager framework depends on four structures and some binary image changes. Some of them are created by the compiler, while others are created and maintained during runtime. The partition manager major components are described next.

4.1 Major Toolchain Changes

Under the Partition Manager framework, all partitionable code's symbols will be modified. These symbols represents the offset in bytes of the given symbol in its partition. The symbol's partition id is saved in the upper 14 bits. If the symbol is not in a partition, the upper 14 bits are zero and the lower bits represents the absolute address of the function. The format of symbol is described in figure 4a.



(a) A symbol address bit range



(b) The Partition list entry

Fig. 4. The symbol address bit range and the Partition list entry format

4.2 The Partition List

This structure is created by the toolchain. It consists of two parts which defines the partition offset on the file and the partition size. Moreover, the partition list resides in the computational element local memory; just after the program's data section. Under this framework, a partition is defined as a set of functions for which their code has been created to be position independent (PIC); thus they can be moved around the memory as the framework sees fit. The actual partition code is not loaded with the program, but left in the global memory of the machine. The partition offset part of a list element shows the offset (in bytes) from the binary entry point. Finally, the size section of the entry contains the size in bytes of the partition on the memory image. Under this model, each of the partitions is identified by a unique number that index them into this list. When a partition is required, the element's partition is loaded using the partition list information and the correct buffer state is set before calling the function. The format and the bit range of the partition list entries are described in figure 4b.

4.3 The Partition Stack

The Partition Stack is a meta-structure which records the calling activity between partitions. It was designed to solve a very simple problem: how to return from a function call which was called from another partition? By keeping the partition stack, the framework can know who the caller of the current function call is, load the partition back if it is required and save function state, i.e. registers which must be saved across partition manager calls. Although the partition code is defined as PIC, when returning from a function call, the framework must load the partition back to its original position. If this is not the case then a large amount of binary rewriting and register manipulation is needed to ensure the correct execution of the function. This is true even for PIC code since registers might have stale addresses to the original sub-buffer location.

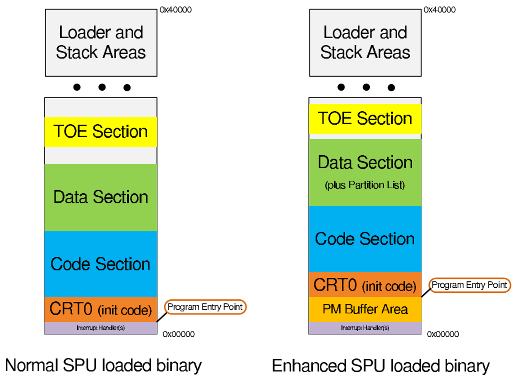


Fig. 5. A comparison between the modified SPE binary image and a normal one

4.4 The Partition Buffer

The Partition Buffer is a special region of the local memory, in which the partition code is swap in and out. It is designed to have a fixed value per application, but it can be divided into sub-buffers if required. Moreover, it contains certain state, like the current Partition index and the lifetime of the code in this partition; which is used for book-keeping and replacement policies. This buffer is managed by the partition manager kernel.

The partition buffer and the partition list modifies the SPE binary image a bit. It adds the list to the end of the data segment and the buffer after the interrupt table. The modified image compared with a normal SPE image is given in figure 5.

4.5 The Partition Manager Kernel

At the center of all these structures lies the Partition Manager. This small function takes care of the loading and management of the partitions in the system. During initialization, the partition manager may statically divide the partition buffer so that several partitions can co-exist with each other. It also applies the replacement policy to the buffers if required. The sequence of operations involve in a simple partition manager call is presented in Figure 6.

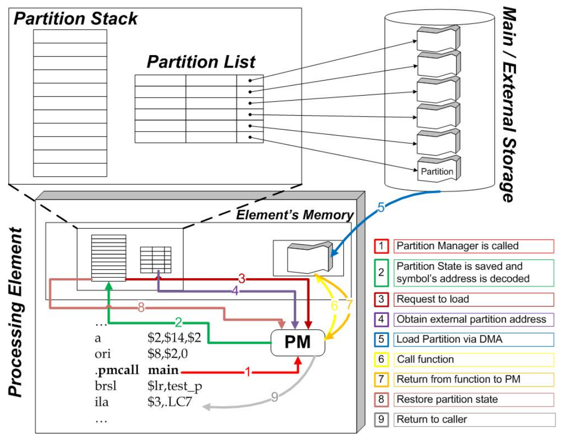


Fig. 6. A typical partition manager call

The next section explains a replacement policy and an enhancement which is applied to the partition manager framework and its effect on the number of operations.

Table 1. The Four States of a Partition

State	Location	Description
Evicted	Main Memory	Partition was not loaded into local memory or it was loaded, evicted and it will not be popped out from the partition stack.
Active	Local Memory	Partition is loaded and it is currently in use
In-active	Local Memory	Partition is not being used, but still resides in local memory
EWOR	Main Memory	Evicted With the Opportunity of Reuse. This partition was evicted from local memory but one of the element of the partition stack will pop its partition id in the near future.

5 The N Buffer: The Lazy Reuse Approaches

Since the partition buffer might be mostly empty most of the time, it can be broken down into sub-buffers to further utilize the hardware resources. This opens many interesting possibilities on how to manage the sub-buffers to increase performance. Even though this area is not new, these techniques are usually applied in hardware. The techniques applied for replacement in this buffer are cache like in which that they try to take advantage of partition locality. The first technique is when the buffer subdivisions are treated as FIFO (first in first out) structures. In this context, this technique is called *Modulus* due to the operation used to select the next replacement. The second one is based on one of the most famous (and successful) cache replacement policies: Least Recently Used (LRU). First, we need to introduce the challenges of dividing the buffer under our framework and how it affects each component.

The partition buffer is enhanced by adding extra state. Each sub-buffer must contain the partition index residing inside of it and an extra integer value to help achieve advanced replacement features (i.e. the integer can represent lifetime for LRU or the next partition index on a pre-fetching mechanism). Moreover, the partition that resides in local memory becomes stateful under this model. A partition now can be *active*, *in-active*, *evicted* or *evicted with the opportunity of reuse*. For a description of the new states and their meanings, please refer to table 1.

Every partition begins in the *evicted* state in main memory. When a partition is used, the partition is loaded and becomes *active*. From this state the partition can become *in-active*, if a new partition is needed and this one resides into a sub-buffer which is not replaced; back to *evicted*, if it replaced and it doesn't belong to the return path of a chain of partitioned function calls; or *Evicted with an Opportunity to Reuse*, in the case that a partition is kicked out but it lies on the return path of a chain of partitioned function calls. An *in-active* partition may transition to evicted and *EWOR* under the same conditions as an active one. An *EWOR* partition can only transition to an *active* partition.

These states can be used to implement several levels of partitioning. One of them is described in Section 5.3.

When returning from a chain the partition function calls, the partition must be loaded into the same sub-buffers that they were called from. To achieve this, the partition stack node must know where the partition originally resided. Thus, this structure must save the sub-buffer id.

5.1 Replacement Policies: The Modulus Approach

Under this approach, sub-buffers form a type of First-In First-Out (FIFO) structure in which the oldest partition is always replaced. It follows the normal formula in which the next sub-buffer to be replaced is selected by the formula $next = (next + 1) mod NSB$ where the $next$ is the sub-buffer in which the new partition is loaded and NSB represents the total number of sub-buffers.

5.2 Replacement Policies: The LRU Approach

Under this approach, each of the sub-buffers has a lifetime counter which decrements every time that a function is called on another partition. The formula to select the next buffer to be replaced becomes $next = MIN(LTA)$ where $next$ is the sub-buffer where the next partition is put and LTA is the Lifetime Array of values. In case that the minimum of the array is a set, this group of elements is managed as if it was a FIFO buffer across different calls of the replacement policy functions. It is important to note that by having multiple sub-buffers, duplication might be possible, the partition framework disallows this. In this way, the framework would not get "confused" when figuring out which sub-buffer to jump in. In the case that a partition is duplicated (for example when returning from a function call into a different sub-buffer), the framework moves the partition to the correct sub-buffer and nullify its old locations. This move saves a load to main memory or prevents the need to adjust all the address in the partition to match the new sub-buffer.

5.3 The Victim Cache for the Partition Framework

Under this framework, the victim cache is a dynamically allocated piece of memory that is created when EWOR partition are called. The EWOR partition is recognized by setting a bit in a partition mask (which has support for 128 partition indexes) every time that a partition stack frame is pushed. When the

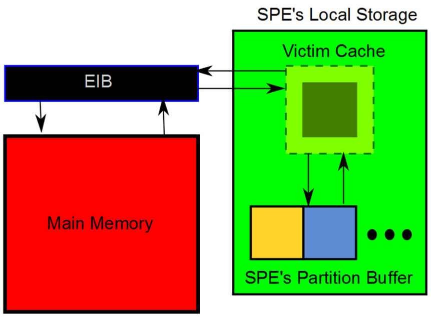


Fig. 7. The victim cache scheme

partition stack frame is popped, the bit on the mask is unset[5]. When a new partition is being loaded into the main memory, the evicted partition index is checked against the partition mask. If they match, the partition code which resides on the sub-buffer is copied to a newly allocated memory block. When an EWOR partition is needed back, the victim cache is checked and the partition is copied back to the sub-buffer if found. Under the current implementation, there is only a single entry on the victim cache. This means that it can only provide support for the most recent EWOR partition on the function chain. A high level overview of the victim cache is given in figure 7.

Since the victim cache can be created dynamically, it can also be brought down in the same way. The framework offers two wrappers for the memory allocators (i.e. malloc and free) which can check the memory pool for availability. If the pool is empty or near it, the victim cache can be brought down to free up memory for the application.

6 Experimental Testbed and Results

The partition manager framework uses a small suite of test programs dedicated to test its functionality and correctness. The testbed framework is called Harahel and it is composed of several Perl scripts and test applications. The next subsections will explain the hardware and software testbeds and presents results for each of the test programs.

6.1 Hardware Testbed

For these experiments, we use the Playstation 3's CBE configuration. This means a Cell processor with 6 functional SPE, 256 MiB of main memory, and 80 GiB of hard drive space. The two disabled SPEs are used for redundancy and to support the hypervisor functionality. Besides these changes, the CBE processor has the same facilities as high end first generation CBE processors. We take advantage of the timing capabilities of the CBE engine. The CBE engine has hardware time counters which ticks at a slower rate than the main processor (in our case, they click at 79.8 MHz). Since they are hardware based, the counters provided minimal interference with the main program. Each of the SPEs contains a single counter register which can be accessed through our own timing facilities.

6.2 Software Testbed

For our experiments, we use a version of Linux running on the CBE, i.e. Yellow Dog with a 2.6.16 kernel. Furthermore, we use the CBE toolchain version 1.1 but with an upgraded GCC compiler, 4.2.0, which was ported to the CBE architecture for OpenOPELL purposes.

[5] This might create false positives in long chain of functions, but it is acceptable in practice.

Table 2. Applications used in the Harahel testbed

Name	Description
DSP	A set of DSP kernels (a simple MAC, Codebook encoding, and JPEG compression) used at the heart of several signal processing applications.
GZIP	The SPEC benchmark compression utility.
Jacobi	A benchmark which attempts to solve a system of equations using the Jacobi method.
Laplace	A program which approximate the result of an integral using the Laplace method.
MD	A toy benchmark which simulates a molecular dynamic simulation.
MGRID	A simplified program used to calculate Multi grid solver for computing a 3-D potential field.
Micro-Benchmark 1	Simple test of one level partitioned calls.
Micro-Benchmark 2	Simple chain of functions across multiple files.
Micro-Benchmark 3	Complete argument register set test.
Micro-Benchmark 5	Long function chain example 2.
Micro-Benchmark 6	Long function chain example 3: Longer function chain and reuse.
Micro-Benchmark 7	Long function chain example 4: Return values and reuse.
Micro-Benchmark 8	Long function chain example 5: Victim cache example.

The applications being tested include kernels used in many famous benchmarks. This testbed includes the GZIP compression and decompression application which is our main testing program. Besides these applications, there is also a set of micro-benchmarks designed to test certain functionality for the partition manager. For a complete list, please refer to 2.

In the next section, we will present the overhead of the framework using a very small example.

6.3 Partition Manager Overhead

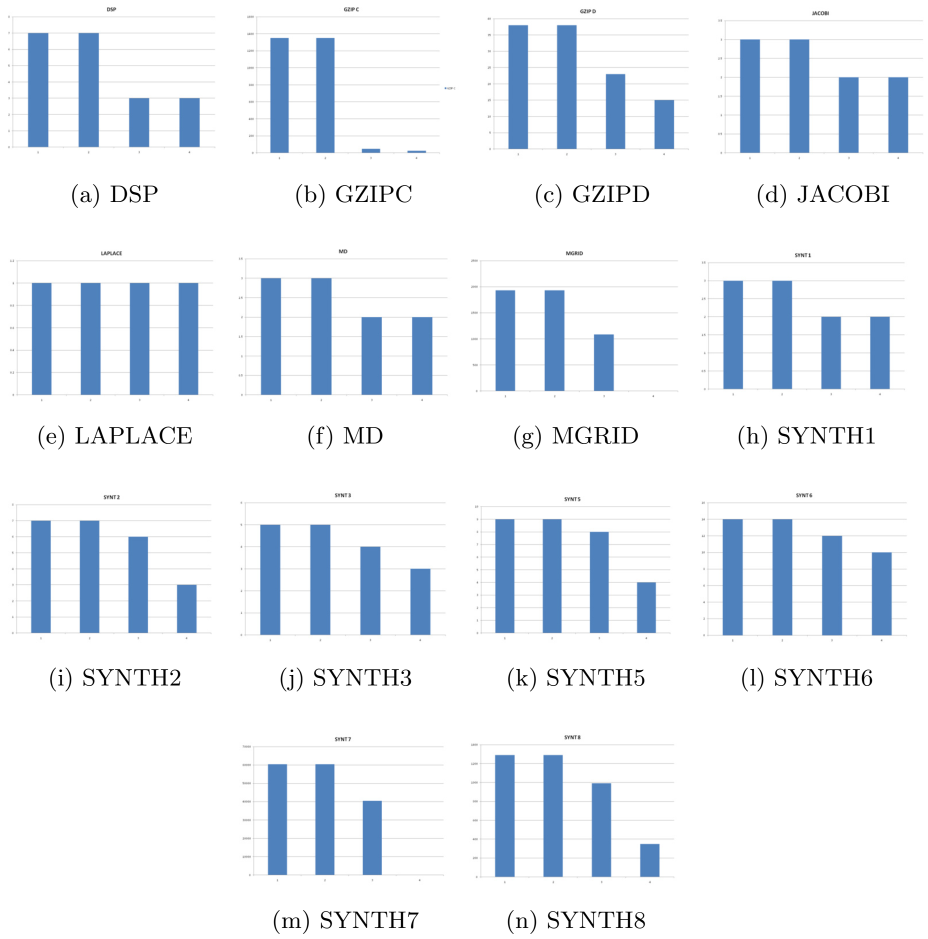
Since this framework represents an initial implementation, the main metric on the studies presented will be the number of DMA transfer produced by an specific replacement policy or/and partition feature. However, we are going to present the overhead for each feature and policy.

The first version represents the original design of the partition manager in which every register is saved and the sub-buffer is not subdivided. The improved version is with the reduction of saved registers but without any subdivision. The final sections represent the policy methods with and without victim cache.

On this model, the overhead with the DMA is between 160 to 200 monitoring cycles. Although this is a high number, these implementations are proof of concepts and they can be greatly optimized. For this reason, we concentrate on the number of DMA transfers since they are the most cycle consuming operation on the partition manager. Moreover, some of these applications will not even run without the partition manager.

6.4 Partition Manager Policies and DMA Counts

Figure 9 and 8 show the relation between the number of DMA and the number of cycles that the application takes using a unoptimized buffer (saving all register



(a) DSP (b) GZIPC (c) GZIPD (d) JACOBI

(e) LAPLACE (f) MD (g) MGRID (h) SYNTH1

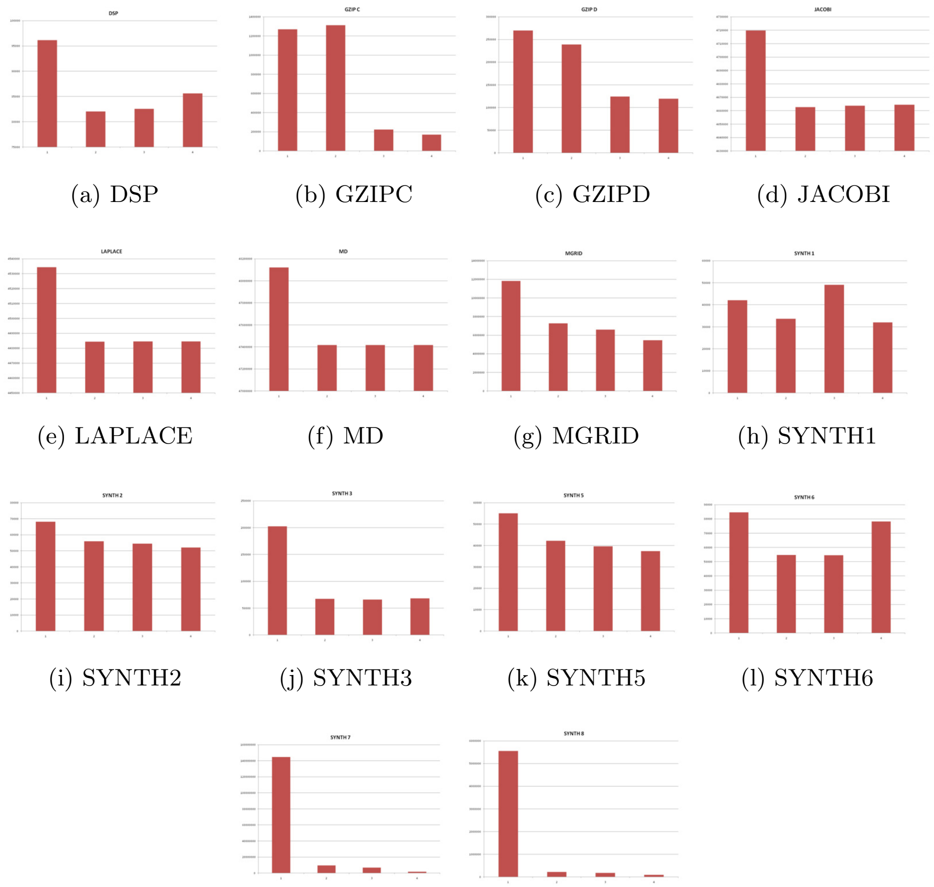
(i) SYNTH2 (j) SYNTH3 (k) SYNTH5 (l) SYNTH6

(m) SYNTH7 (n) SYNTH8

Fig. 8. DMA counts for all applications for an unoptimized one buffer, an optimized one buffer, optimized two buffers and optimized four buffer versions

file), optimized one buffer (rescheduled and reduction of the number of registers saved), optimized two buffers and optimized four buffers. For most applications, there are a correlation between a DMA's reduction and a reduction of execution time. However, for cases in which the number of partition can fit in the buffers, the cycles mismatch like in Synthetic case 1 and 6.

Figure 10 show the ratio of Partition manager calls versus the number of DMA transfers. The X axis represents the applications tested and the ratios of calls versus one, two and four buffers. As the graph shows, adding the extra buffers will dramatically lower the number of DMA transfers in each partition manager call.



(a) DSP (b) GZIPC (c) GZIPD (d) JACOBI

(e) LAPLACE (f) MD (g) MGRID (h) SYNTH1

(i) SYNTH2 (j) SYNTH3 (k) SYNTH5 (l) SYNTH6

(m) SYNTH7 (n) SYNTH8

Fig. 9. Cycle counts for all applications for an unoptimized one buffer, an optimized one buffer, optimized two buffers and optimized four buffer versions

Figure 11 selects the GZIP and MGRID applications to show the advantage of using both replacement policies. In the case of MGRID, both policies gives the same counts because the number of partitions is very low. In the case of the GZIP compression, the LRU policy wins over the Modulus policy. However, in the case of decompression, the Modulus policy wins over the LRU one. This means that the policy depends on the application behavior which opens the door to smart application selection policies in the future.

Finally, in Figure 12, we show that the victim cache can have drastically effects on the number of DMA transfers on a given application (Synthetic case 8). As the graph shows, it can produce a 88x reduction in the number of DMA transfers.

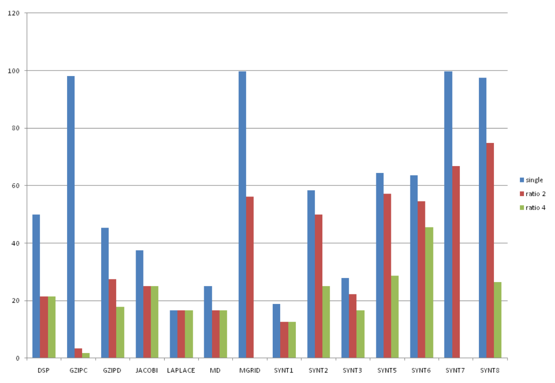


Fig. 10. Ratio of Partition Manager calls versus DMA transfers

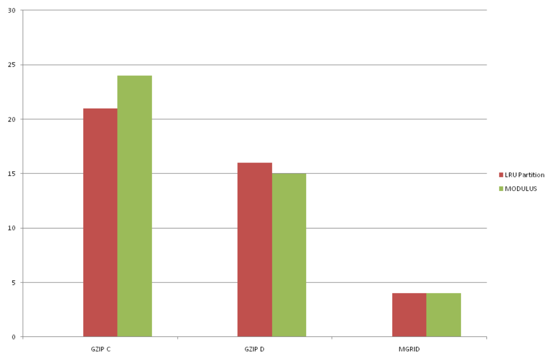


Fig. 11. LRU versus Modulus DMA counts for selected applications

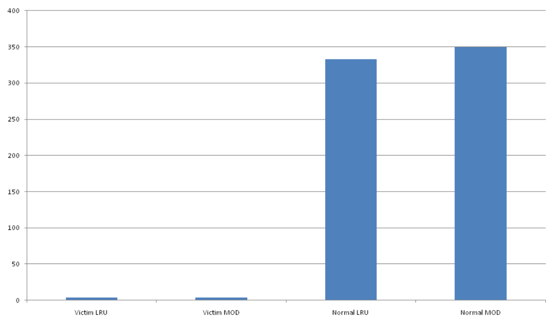


Fig. 12. The victim cache comparison with LRU and Modulus policies

7 Conclusions and Future Work

Ideas presented in this paper show the trend of software in the many core age: the software renaissance. Under this trend, old ideas are coming back to the plate: Overlays, software caches, dataflow execution models, micro kernels, among others. This trend is best shown in architectures like Cyclops-64[5] and the Cell B.E.'s SPE units. Both designs exhibit explicit memory hierarchy, simple pipelines and the lack of virtual memory. The software stacks on these architectures are in a heavily state of flux to better utilize the hardware. This fertile research ground allows the reinvention of these classic ideas. The partition manager frameworks rise from this flux.

This paper shows a framework to support the code movements across heterogeneous accelerators components. It shows how these effort spans across all components of the software stack. Moreover, it depicts its place on a higher abstraction framework for a high level parallel programming language. It shows the effect of several policies dedicated to reduce the number of high latency operations. Future work on this area include the creation of a partition based function call graph which can be used for pre-fetching schemes and the extension of task based framework that allows percolation of code.

References

1. CBE Architectural Manual
2. Bellens, P., Perez, J.M., Badia, R.M., Labarta, J.: Cellss: a programming model for the cell be architecture. In: ACM/IEEE Conference on Supercomputing, p. 86. ACM (2006)
3. Caubet, J.: Programming ibm powerxcell 8i/qs22 libspe2, alf, dacs (May 2009)
4. Chen, C., Manzano, J.B., Gan, G., Gao, G.R., Sarkar, V.: A Study of a Software Cache Implementation of the OpenMP Memory Model for Multicore and Manycore Architectures. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010, Part II. LNCS, vol. 6272, pp. 341–352. Springer, Heidelberg (2010)
5. del Cuvillo, J., Zhu, W., Hu, Z., Gao, G.R.: Tiny threads: A thread virtual machine for the cyclops64 cellular architecture. In: International Parallel and Distributed Processing Symposium, vol. 15, p. 265b (2005)
6. Manzano, J.B., Hu, Z., Jiang, Y., Gan, G., Song, H.-J., Park, J.-G.: Toward an Automatic Code Layout Methodology. In: Chapman, B., Zheng, W., Gao, G.R., Sato, M., Ayguadé, E., Wang, D. (eds.) IWOMP 2007. LNCS, vol. 4935, pp. 157–160. Springer, Heidelberg (2008)
7. O'Brien, K., O'Brien, K., Sura, Z., Chen, T., Zhang, T.: Supporting openmp on cell. Int. J. Parallel Program. 36, 289–311 (2008)

Optimizing the Concurrent Execution of Locks and Transactions

Justin E. Gottschlich and JaeWoong Chung

Programming Systems Lab (PSL), Intel Corporation
{justin.e.gottschlich,jaewoong.chung}@intel.com

Abstract. Transactional memory (TM) is a promising alternative to mutual exclusion. In spite of this, it may be unrealistic for TM programs to be devoid of locks due to their abundant use in legacy software systems. Consequently, for TMs to be practical they may need to manage the interaction of transactions and locks when they access the same shared-memory. This paper presents two algorithms, one coarse-grained and one fine-grained, that improve the state-of-the-art performance for TMs that support the concurrent execution of locks and transactions. We also discuss the programming language constructs that are necessary to implement such algorithms and present analyses that compare and contrast our approach with prior work. Our analyses demonstrate that, *(i)* in general, our proposed coarse- and fine-grained algorithms improve program concurrency but *(ii)* an algorithm's concurrent throughput potential does not always lead to realized performance gains.

1 Introduction

Transactional memory (TM) [6,11] is a promising alternative to mutual exclusion because it simplifies parallel programming by moving some of the complexity of shared memory management away from the programmer's view. While TM shows promise for future software, most TMs have undefined behavior when locks and transactions are used to concurrently synchronize the same shared-memory. This creates a notable void in TM applicability for programmers who wish to use transactions in legacy software that already use locks.

Zyulkarov et al. explored one possible solution to this problem by converting all the locks used in the Quake game server, a large-scale multiplayer software engine [1], to transactions [19]. During this process they encountered significant transition challenges, such as unstructured locking (i.e., non-block-structured critical sections), I/O and system calls, error handling, privatization and thread local memory, and compensating and escape actions necessary to handle up to nine levels of dynamic nesting. They also found that only using transactions resulted in reduced performance when compared to only using locks or using a combination of the two. These results seem to indicate that converting all locks to transactions may be unrealistic for all but expert parallel programmers and, if such a conversion were successful, it might result in a performance degradation.

Another alternative, such as that proposed by Volos et al.'s TxLocks [15] and Ziarek et al.'s P-SLE and atomic serialization [17], is to provide support for transactions and locks so that they can safely access the same shared-memory. In this paper, we focus on this approach and present a system that improves performance over the aforementioned research. A difference in our approach and the prior work of Volos et al. and Ziarek et al. is that our system extends the programming language constructs for locks and transactions so that they contain static (compile-time) information about the conflicts that persist between them, while prior systems deduce conflicts between locks and transactions dynamically (run-time). We demonstrate that our static extensions reduce the number of false conflicts produced at run-time, resulting in improved concurrent throughput.

Throughout this paper we gradually extend and refine the notion of concurrent lock and transaction execution. Our algorithms, one coarse-grained and one fine-grained, manage two general cases of conflicts: (i) non-nested cases, when locks and transactions that have no nesting are executed side-by-side and (ii) nested cases, when locks are nested within transactions and transactions or locks execute along side such transactions.[1]

The algorithms require two programming interface enhancements that are similar in structure, but different in purpose, to those proposed by Usui et al. for adaptive locks [14]. We augment the `atomic` transaction block so it takes a single parameter, `locks[]`, which is a list of locks that conflict with the transaction. We also introduce a `TmLock` that behaves like a typical mutex and additionally communicates with the TM subsystem before it is acquired and after it is released so the TM system can manage conflicts of, and the forward progress between, `TmLock`s and transactions. We also present an interesting finding, which extends the prior findings of Gottschlich et al. [4], and reflects a counterintuitive result. Our benchmarks show that our coarse-grained policy is *always* faster than the prior systems, while our fine-grained policy is usually faster, but in some cases it can be notably slower (up to $\approx 2\times$).

This paper makes the following technical contributions.

1. We present two algorithms, one coarse-grained and one fine-grained, that allow for the concurrent execution of locks and transactions in the same program and improve concurrent throughput beyond the state-of-the-art.
2. We propose two new TM language constructs: `TmLock` and an extended `atomic` block structure that are used statically (compile-time) to capture the conflicts between locks and transactions.
3. We include a qualitative analysis that precisely captures the potential concurrent throughput of existing systems compared to our algorithms.
4. Our experimental results show that our fine-grained algorithm yields up to $\approx 2.0\times$ performance improvements over prior systems but can sometimes degrade performance. Our coarse-grained algorithm yields up to $\approx 1.5\times$ performance improvements and never performs worse than prior systems for our tested benchmarks.

[1] A third case, when transactions are nested within locks, is not discussed as Volos et al. [15] and Gottschlich et al. [4] show it does not require special effort.

2 Background and Related Work

When transactions and locks are executed concurrently, they can behave inconsistently due to the differences in their critical section execution [10,15,17]. Mutual exclusion locks generally use pessimistic critical sections that are limited to one thread of execution [3,18]. Transactions can use optimistic critical sections that support unlimited concurrent thread execution and resolve conflicts at various points during transaction execution [7,8]. This difference can incur correctness issues (e.g., transactions being aborted after executing a nested locked region with I/O operations) and cause pathological interferences (e.g., blocking, livelock, and deadlock) [15].

2.1 Conflicts between Locks and Transactions

For transactions and locks to execute concurrently, the notion of when they conflict must be understood. A lock *conflicts* with a transaction (and vice versa) when both access the same memory location and at least one of those accesses is a write. Although this notion of a conflict is principally the same as that for transactions alone, the conflict resolution [5,12] and concurrent execution guarantees may be notably different. For this paper, we assume mutually exclusive critical sections are not failure atomic and they execute pessimistically, that is, without write buffering or speculative lock elision [9]. Therefore, conflicts that arise between transactions and a given mutex must be identified before the mutex's critical section is executed.

TxLocks. Volos et al. propose a transaction-aware lock primitive, TxLock, to handle the conflicts between locks and transactions without special hardware support [15]. When used outside of a transaction, TxLocks execute pessimistically and no information is provided from TxLocks to the transactions that might concurrently execute alongside them. This is done to minimize the programmer's burden of using TxLocks. TxLocks must therefore assume that all transactions can conflict with any TxLock and, likewise, prohibit TxLocks and transactions from executing concurrently. While TxLocks correctly manages conflicts between locks and transactions, and minimizes programming overhead, such an approach can limit concurrent throughput because it conservatively overestimates the conflicts that exist between TxLocks and transactions.

P-SLE and Atomic Serialization. Ziarek et al. introduce two key concepts: pure-software lock elision (P-SLE) and atomic serialization [17]. P-SLE eliminates conflicts between locks and transactions by converting locks into transactions. However, as noted by the authors, and as enumerated by Zyulkarov et al. [19], there are numerous reasons why the atomic regions protected by locks cannot seamlessly transition into transaction-based atomic regions (see Section 1). P-SLE handles these cases by reinstating all locks and using a single global lock to serialize transaction execution. This behavior, called *atomic serialization*, guarantees mutual exclusion between transactions [17]. Although atomic serialization is correct because it serializes all transactions, such a strict serialized ordering may be unnecessary and may adversely effect performance.

Full Lock Protection. TxLocks [15] and atomic serialization [17] provide essentially the same guarantee: they prevent a transaction from executing in one thread while a lock-based critical section is active in another. We call this behavior *full lock protection* because the shared-memory accessed within a lock is fully protected from transaction interference.

However, TxLocks and atomic serialization are not identical. Atomic serialization allows irrevocable operations to be used within locks that can then be placed inside of transactions. TxLocks does not allow such behavior. TxLocks' nesting model, therefore, differs from our own and atomic serialization. Therefore, when we discuss locks nested within transactions, we only consider atomic serialization. For non-nested cases, we consider both TxLocks and atomic serialization because their behavior is identical. For the remainder of the paper, we refer to TxLocks and atomic serialization as implementations of full lock protection (under the above restrictions), as it simplifies the discussion.

3 Language Constructs

We propose two interface extensions to enable programmers to efficiently manage conflicts between locks and transactions. Using these extensions, the programmer can choose to provide coarse-grained, fine-grained, or no information about potential conflicts between locks and transactions.

3.1 Coarse-Grained Conflict Management: TmLock

The `TmLock` data structure is used to allow programmers to provide coarse-grained information about potential conflicts between locks and transactions. The code shown below is a pseudocode example for the `TmLock` declaration and usage. `TmLock`s are used for locks that could potentially conflict with *any* transaction. When a `TmLock` is acquired at run-time, the TM system aborts or commits all in-flight transactions and then prevents any transaction from (re)starting until the the `TmLock` is released. Contention among `TmLock`s are handled in the same way as normal locks are handled. By allowing programmers to differentiate between locks that potentially conflict with transactions from the locks that do not, `TmLock` can improve concurrency because normal locks can run in parallel alongside transactions without conflict.

```
class TmLock {                                TmLock tmLock;
public:
  void lock()                                 tmLock.lock();
  { /* Arbitration Algorithm */ }             ...
  void unlock()                               tmLock.unlock();
  { /* TM Subsystem Communication */ }
};
```

For unmanaged languages, if the programmer is unsure about potential conflicts between a lock and any transaction, a `TmLock` can be used in place of a normal lock as a conservative overestimation. Likewise, this same approach can be used for managed languages where all locks can be automatically converted to `TmLock`s by the compiler to guarantee conservative correctness. In the event of external locking conflicts (e.g., OS-level or library-level conflicts), the programmer can wrap external interfaces with `TmLock`s.

3.2 Fine-Grained Conflict Management: atomic()

We extend the *atomic* block to allow programmers to manage conflicts between locks and transactions at a finer granularity than supported by `TmLock` alone. Our extended atomic block, `atomic (TmLock [])`, takes an array of `TmLock`s as an argument and is shown in the example below. The array contains the list of `TmLock`s that can conflict with the transaction.

If a programmer is unsure about potential conflicts, she can conservatively overestimate them by using the atomic block with no argument, indicating the transaction may conflict with any `TmLock`. If the programmer knows the transaction does not conflict with any `TmLock`, she can pass `NULL` to the atomic block, indicating no conflicts. Under this programming model, transactions behave correctly with any number of `TmLock`s without *any* modification to the `atomic` construct. We chose this design over others because it reduces the initial challenge of integrating transactions into lock-based software and because it creates an iterative optimization path for programmers, where `atomic` blocks that have been overestimated to conflict with all `TmLock`s can be optimized at a later date.

```
    // syntax: atomic (TmLock []) {}
    TmLock L1, L2; TmLock locks[] = {L1, L2};
    atomic (locks) {...}; // conflict with L1 and L2
    atomic (NULL) {...};  // no conflicts
    atomic {...}; // conflicts with all TmLocks
```

4 Algorithms

The steps to (re)start and end a transaction for both the fine- and coarse-grained algorithms are shown in Algorithm 1. The definition of a *conflicting* `TmLock`, represented by *conflictingLocks*, is context-sensitive. The coarse-grained algorithm defines all `TmLock`s as conflicting, so *conflictingLocks* is equal to all `TmLock`s. For the fine-grained algorithm, *conflictingLocks* is equal to those `TmLock`s specified in the atomic `TmLock` list. The *IsolatedTx*() procedure returns true if an isolated transaction (i.e., a transaction that forbids the concurrent execution of other transactions) is active, otherwise it returns false. The *obtainedMutexes* set collects the `TmLock`s that have been acquired during a transaction's execution and is used to ensure a transaction's state remains isolated by preventing other threads from acquiring such locks until the transaction has committed [15].

Due to space limitations, we have omitted the serialization used to access the shared data in all the algorithms shown in this section. In the `Begin` procedure, lines 3-4 are serialized. In the `End` procedure, the entire procedure is serialized.

Algorithm 1. Begin and End Transaction Procedures

```
1: procedure BEGIN(Transaction $tx$)
2:     loop
3:         Set $L$ = TmLocks.locked()        ▷ Returns set of currently locked TmLocks
4:         if ($L \cap tx.conflictingLocks \equiv \emptyset \wedge\ !IsolatedTx()$) then return
5:         $sleep()$; $continue$
6: procedure END(Transaction $tx$)
7:     $tx.obtainedMutexes = \emptyset$
8:     $Unblock(tx.conflictingLocks)$    ▷ Fine-grained only: unblock $tx.conflictingLocks$
```

4.1 Coarse-Grained Algorithm

Algorithm 2 shows the `TmLock.lock` and `TmLock.unlock` procedures for the coarse-grained algorithm when a `TmLock` is both nested, and not nested, within a transaction. When a `TmLock` is acquired inside a transaction, the transaction requests permission from the contention manager (CM) to become isolated. If successful, this request aborts all active transactions and prevents new transactions from starting. This is done because `TmLock`s, by the coarse-grained definition, can conflict with any transaction. Hence, when a `TmLock` is acquired within a transaction, no other transactions can execute alongside it.

If the `TmLock` is not nested within a transaction, the algorithm calls $AbortTxes()$ which requests permission from the CM to abort all transactions if there are any active ($ActiveTxes()$). In both cases, once there are no active transactions, except for the transaction that may nest the `TmLock`, the `TmLock` can be acquired.

When a `TmLock` is nested within a transaction, the *tx.partialCommit*() procedure is called as soon as the `TmLock` is acquired. This procedure commits the transaction's executed operations to the program state so non-transactional reads and writes, performed inside the `TmLock`'s critical section, will access the correctly updated memory. In Algorithm 2, lines 2-6 and lines 8-9 are serialized if the `while` loop's condition is false. If it is true, only the procedures called within the `while` loop's condition are serialized. For `TmLock.unlock`, the entire procedure is serialized.

4.2 Fine-Grained Algorithm

Algorithm 3 shows the fine-grained algorithm for `TmLock.lock` and `TmLock.unlock`. It includes cases where the procedures are nested, and not nested, within a transaction. The serialization used in Algorithm 3 is identical to Algorithm 2.

When a `TmLock.lock` call is not nested within a transaction, the `TmLock` requests the CM's permission to abort conflicting transactions, via $AbortConflictTxes()$. Because only those transactions whose extended atomic block has included the `TmLock` are aborted (checked by the $ConflictTxes()$

Algorithm 2. Coarse-Grained Lock and Unlock TmLock Procedures

```
Require: threadId is the global and unique id of thread that called lock()
 1: procedure TmLock.lock
 2:     Transaction* $tx = ActiveTx(threadId)$                      ▷ Pointer to active tx
 3:     if $(tx \neq NULL)$ then                        ▷ TmLock.lock() is nested within tx
 4:         if $(tx.makeIsolated(CM))$ then                    ▷ Request CM permission
 5:             Acquire TmLock mutex; $tx.obtainedMutexes.insert(this)$
 6:             $tx.partialCommit()$
 7:     else                                             ▷ TmLock.lock() not nested in tx
 8:         while $(ActiveTxes() \neq \emptyset \vee AbortTxes() \equiv false)$ do {}
 9:         Acquire TmLock mutex
10: procedure TmLock.unlock
11:     Transaction* $tx = ActiveTx(threadId)$
12:     if $(tx \neq NULL)$ then                      ▷ TmLock.unlock() is nested within tx
13:         if $tx.obtainedMutexes \cap this \equiv \emptyset$ then   ▷ TmLock.lock() not nested in tx
14:             Throw EarlyReleaseDeadLock exception          ▷ Prevent deadlock [4]
15:     Release TmLock mutex
```

procedure), those transactions that did not include the `TmLock` in their atomic block can continue to execute while a `TmLock` is acquired.

When `TmLock.lock` is called inside of a transaction, the transaction first requests permission to become irrevocable [16]. Irrevocable transactions cannot be aborted, just like isolated transactions, but unlike isolated transactions, they may yield greater concurrent throughput because other non-conflicting, revocable transactions may execute alongside them. Once the transaction is made irrevocable, all transactions that conflict with the `TmLock` are aborted, via $AbortConflictTxes()$. When $AbortConflictTxes()$ is called within an irrevocable transaction, it always returns true.

By allowing transactions that acquire `TmLock`s to become irrevocable, rather than isolated, the fine-grained algorithm has the opportunity to produce more concurrent throughput than the coarse-grained algorithm. Transactions that acquire `TmLock`s using the fine-grained algorithm can be made irrevocable, rather than isolated, because the fine-grained algorithm requires that all conflicts between transactions and `TmLock`s be listed within in the extended atomic block. Because of this, non-conflicting revocable transactions may execute alongside an irrevocable transaction that has acquired a `TmLock`.

5 Qualitative Comparison

In this section, we compare the concurrency potential of full lock protection to our coarse- and fine-grained algorithms. We consider two general cases: (i) when locks are not nested within transactions and (ii) when locks are nested within transactions.

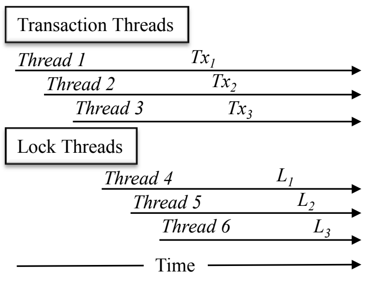
Algorithm 3. Fine-Grained Lock and Unlock TmLock Procedures

```
Require: threadId is the global and unique id of thread that called lock()
 1: procedure TmLock.lock
 2:     Transaction* $tx = ActiveTx(threadId)$                         ▷ Pointer to active tx
 3:     if ($tx \neq NULL$) then                             ▷ TmLock.lock() is nested within tx
 4:         if ($tx.makeIrrevocable(CM)$) then                       ▷ Request CM permission
 5:             $AbortConflictTxes(this)$; Acquire TmLock mutex
 6:             $tx.obtainedMutexes.insert(this)$; $tx.partialCommit()$
 7:     else                                                ▷ TmLock.lock() not nested within tx
 8:         while ($ConflictTxes(this) \neq \emptyset \vee AbortConflictTxes(this) \equiv false$) do {}
 9:         Acquire TmLock mutex
10: procedure TmLock.unlock
11:     Transaction* $tx = ActiveTx(threadId)$
12:     if ($tx \neq NULL$) then                           ▷ TmLock.unlock() is nested within tx
13:         if $tx.obtainedMutexes \cap this \equiv \emptyset$ then    ▷ TmLock.lock() not nested in tx
14:             Throw EarlyReleaseDeadLock exception               ▷ Prevent deadlock [4]
15:     else                                                ▷ TmLock.unlock() is not inside tx
16:         $Unblock(this)$          ▷ Send message to unblock threads waiting for this
17:     Release TmLock mutex
```

5.1 No Nesting between Locks and Transactions

We use the six threaded example shown in Figure 1 to contrast the three algorithms when locks are not nested within transactions. Each thread executes a single function, shown in Figure 2, in a staggered fashion. Three of the threads use transactions, thread T_1 runs `Tx1()`, T_2 runs `Tx2()`, and T_3 runs `Tx3()`, while the other three use locks, thread T_4 runs `L1()`, T_5 runs `L2()`, and T_6 runs `L3()`. The example has a single conflict between T_3's transaction (`Tx3()`) and T_4 and T_5's locks (`L1` and `L2`). Threads T_1, T_2, and T_6 do not exhibit any conflicts and only exist to draw out the differences in the concurrency potential of the algorithms. The coarse-grained conflict management algorithm only uses the information from `TmLock`s, while the fine-grained algorithm uses both `TmLocks` and the extended `atomic` blocks. Full lock protection uses no information from either construct, which is identical to both TxLocks and atomic serialization.

Full Lock Protection. Transactions and locks are not allowed to concurrently execute when using full lock protection. This is because no information is provided about the conflicts that may persist between them. Because of this, the maximum concurrent lock and transaction throughput achievable by full lock protection is: $m(NoNesting_{fl}) = max(L_n, T)$. L_n is the maximum number of lock-based critical sections that do not conflict with one another and T is the maximum number of transactions that can be concurrently executed. Because no conflict information is used by full lock protection, only one type of critical section can be executed, locks or transactions, at a time. Figure 3 presents a visual model of full lock protection under the six threaded example.



```
TmLock L1, L2; Lock L3;
TmLock list[] = {L1,L2};

void Tx1() { atomic(NULL) {...} }
void Tx2() { atomic(NULL) {...} }
void Tx3() { atomic(list) {...} }
void L1() { /* lock/unlock L1 */ }
void L2() { /* lock/unlock L2 */ }
void L3() { /* lock/unlock L3 */ }
```

Fig. 1. Threads With No Nesting

Fig. 2. Functions With No Nesting

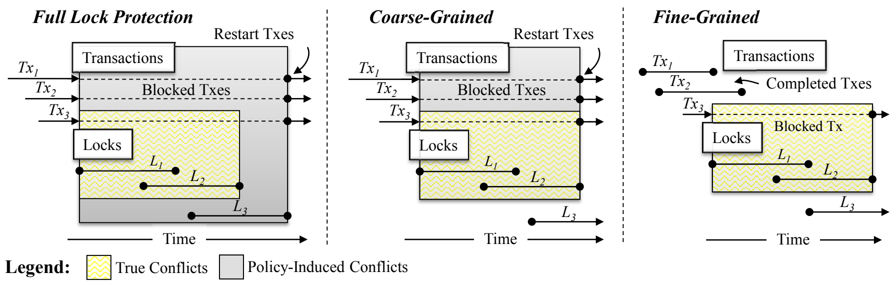


Fig. 3. Non-Nested Example: Full Lock Protection, Coarse-, and Fine-Grained

The transactions used in threads $T_1 - T_3$ are blocked while the lock-based critical sections of threads $T_4 - T_6$ execute, even though thread T_6's lock does not conflict with any of the transactions and threads T_1 and T_2's transactions do not conflict with any of the locks.

Coarse-Grained Conflict Management. The coarse-grained algorithm leverages `TmLock` information that distinguishes locks that might conflict with transactions from those that are guaranteed to be conflict-free. This yields the following concurrency potential: $m(NoNesting_{tm}) = max(L_{nl}, (L_{na} + T))$. L_{nl} is the total number of locks that do not conflict with one another, but do conflict with transactions. L_{na} is the total number of locks that do not conflict with one another and do not conflict with transactions. T is the maximum number of transactions that can be executed. In the six threaded example, the coarse-grained approach allows the transactions in threads $T_1 - T_3$ to execute while only T_6 is executing, because lock `L3` does not conflict with any transaction. As seen in Figure 3, this optimization shortens the overall TM run-time compared to full lock protection by allowing $T_1 - T_3$ to restart their transactions as soon as `L1` and `L2`'s critical section execution has completed.

Fine-Grained Conflict Management. The fine-grained algorithm uses both `TmLock` and the extended `atomic` block information, enabling it to capture the greatest amount of potential concurrency. This is expressed as: $m(NoNesting_{tx}) = C_{lt} + L_{na} + T_{na}$. C_{lt} is the largest system selected set of locks and transactions that can be run concurrently without any conflicts. L_{na} is the total number of locks that do not conflict with one another and have not been flagged as conflicting with any transaction. T_{na} is the total number of transactions that can be executed without conflicting with a lock. In the six threaded example, the fine-grained algorithm only stalls thread T_3 when locks L1 and L2, of threads T_4 and T_5, are executing. The conflict time introduced by this approach is equal to the actual conflict time between the transactions and locks, resulting in the maximum amount of concurrent execution of locks and transactions.

5.2 Nesting Locks Inside of Transactions

It complicates conflict management for locks to be lexically nested within transactions. Locks must remain mutually exclusive, and, for this paper, we assume that locks do not have *failure atomicity* (i.e., they do not emit the property of side-effect free failure such as those found in transactions [2]). To ensure mutual exclusion among locks, when a lock is acquired inside of a transaction, the transaction becomes irrevocable (i.e., it cannot be aborted) [13, 16] or isolated (i.e., it is irrevocable and executes without any concurrently executing transactions).

Irrevocable and isolated transactions limit concurrency amongst transactions because conflicts between such transactions must be prevented pessimistically. That is, other transactions of the same type must be prevented from running concurrently even though conflicts between them may not exist. For isolated transactions, only one transaction can execute at a time, whereas with irrevocable transactions, only one irrevocable transaction may execute at a time, but any number of revocable transactions can concurrently execute alongside it.

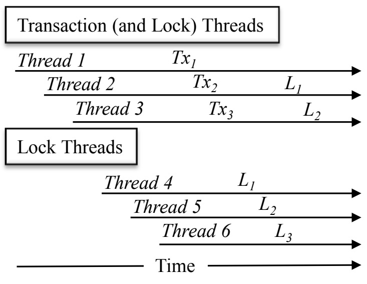


Fig. 4. Threads With Nesting

```
TmLock L1, L2; Lock L3;
TmLock tm1[] = {L1}, tm2[] = {L2};

void Tx1() { atomic(NULL) {...} }
void Tx2() { atomic(tm1) {L1();} }
void Tx3() { atomic(tm2) {L2();} }
void L1() { /* lock/unlock L1 */ }
void L2() { /* lock/unlock L2 */ }
void L3() { /* lock/unlock L3 */ }
```

Fig. 5. Functions With Nesting

We use a six threaded example shown in Figures 4 and 5 to draw out the differences in potential concurrency of the three algorithms when locks are nested within transactions. Three of the threads, $T_1 - T_3$, execute transactions while the other three, $T_4 - T_6$, execute locks. Thread T_1 runs the transaction Tx1(), which contains no nested locking. Thread T_2 runs transaction Tx2() which lexically

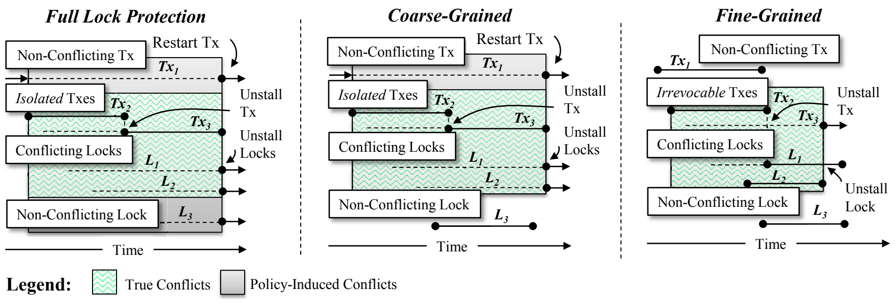


Fig. 6. Nested Example: Full Lock Protection, Coarse-, and Fine-Grained

nests lock `L1`. Thread T_3 runs transaction `Tx3()` which nests lock `L2`. Threads $T_4 - T_6$ access locks `L1 - L3`, respectively.

Only two conflicts exist in the six threaded nested example. Threads T_2 and T_4 conflict on lock `L1`, while threads T_3 and T_5 conflict on lock `L2`. Threads T_1 and T_6 have no conflicts and are included only to highlight the differences in potential concurrency of the three algorithms.

Full Lock Protection. Full lock protection conservatively assumes that all transactions may contain nested locks. Because of this, it disallows other transactions from executing until a transaction that has acquired a nested lock has committed. In addition, once a lock is acquired within a transaction, full lock protection must assume other locks may also be acquired within the transaction. Because such transactions guarantee isolation, and because locks cannot be released until transaction commit time, as explained in Section 4, full lock protection must prevent all lock-based critical sections from executing until the transaction has committed. This behavior is identical to atomic serialization, but not TxLocks for of the reasons mentioned in Section 2.1.

The maximum concurrent throughput given these restrictions is: $m(Nesting_{fl}) = max(t_l, L_n, T_{nl})$. t_l is a single transaction that acquires a lock inside of it. L_n is the maximum number of locks that do not conflict with one another. T_{nl} is the maximum number of transactions that can be executed that do not have locks inside of them. Full lock protection only supports the execution of one of the following: an isolated transaction that contains nested locks, non-conflicting locks, or revocable transactions (as denoted by the *max* function).

Coarse-Grained Conflict Management. When using the coarse-grained approach, if a transaction acquires a `TmLock`, the transaction becomes isolated because the algorithm must assume all transactions can conflict with a `TmLock`. Because isolated transactions must not abort, no `TmLock`s can be acquired while the isolated transaction is active. This results in the following maximum concurrent throughput potential: $m(Nesting_{tm}) = max((t_l + L_{nt}), (L_n + T_{nl}))$. t_l is a single transaction that acquires a `TmLock` inside of it. L_{nt} is the maximum number of lock-based critical sections that do not conflict with one another and do not conflict with t_l. L_n is the maximum number of locks that do not conflict with each other, but do conflict with t_l. T_{nl} is the maximum number of transactions that can be executed which do not have `TmLock`s inside of them and do not conflict with L_n.

The coarse-grained approach is an improvement over full lock protection, and likewise, atomic serialization, because non-conflicting locks (i.e., locks that are not `TmLock`s) can be executed alongside an isolated transaction that has acquired a lock. In addition, another improvement over full lock protection is that non-conflicting locks can concurrently execute with other non-conflicting transactions, as illustrated with Figure 6.

Fine-Grained Conflict Management. The fine-grained approach offers the greatest potential concurrent throughput by allowing `TmLock`s and revocable transactions to be run in parallel with a transaction that has acquired a `TmLock`. When using the fine-grained algorithm, if a `TmLock` is acquired by a transaction, the transaction becomes irrevocable, not isolated. This is an improvement over the coarse-grained algorithm because irrevocable transactions allow other revocable transactions to execute in parallel. Likewise, because each transaction using the fine-grained algorithm has its conflicting `TmLock`s listed, the `TmLock`s that are not listed as conflicting by a transaction can be acquired while the transaction executes, even if it is irrevocable.

This results in the following concurrency potential: $m(Nesting_{tx}) = max((t_l + L_{nt} + T_r), (L_n + T_{nl}))$. All variables of the fine-grained algorithm are the same as coarse-grained algorithm except T_r. T_r is the maximum number of *revocable* transactions that can be executed concurrently alongside t_l that do not conflict with the set of locks, L_{nt}. As shown in Figure 6, the conflict time introduced by the fine-grained algorithm is equal to the actual conflict time.

6 Experimental Results

Our benchmark data was gathered on a 1.0 GHz Sun Fire T2000 supporting 32 concurrent hardware threads. For all benchmarks with the exception of the red-black trees in Figure 10, the x-axis shows the number of active threads and the y-axis shows the total execution time in seconds. In Figure 10, the x-axis shows the number of inserts and lookups and the y-axis shows the total execution time in seconds. Smaller total time indicates more efficient execution.

For cases where locks are not nested within transactions, we use an experimental model that ranges from $4-32$ threads in multiples of four. In the four threaded version, the threads populate three containers of the same type (linked list, hash table, or red-black tree) and then perform sanity checks (i.e., `lookup()`s) on the values inserted. One thread populates a container with locks, another thread populates a different container with transactions, and the third and fourth threads populate a third container with locks and transactions, respectively. Figures 7 and 8 display the execution time of these benchmarks ranging from 4-32 threads.

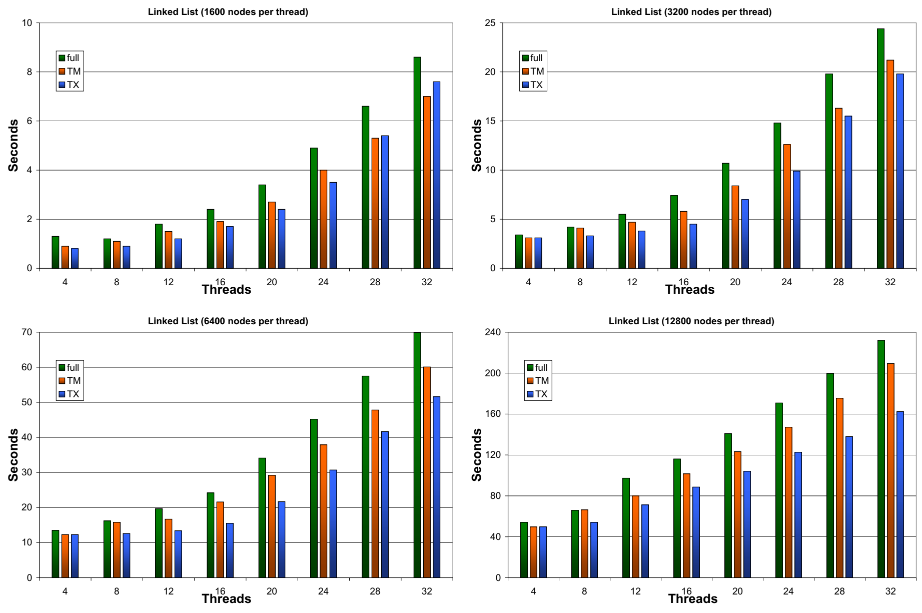


Fig. 7. Linked List Using Locks and Transactions Without Nesting

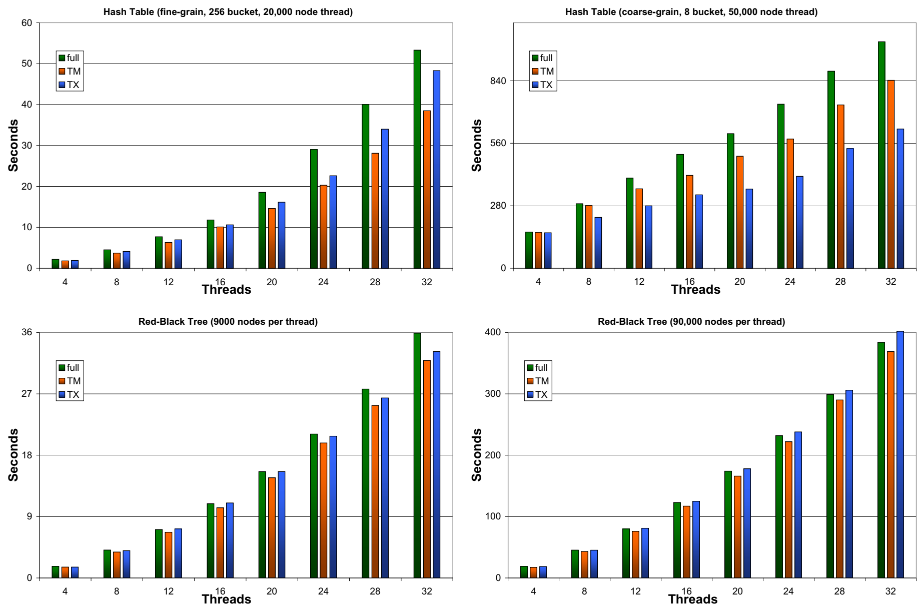


Fig. 8. Hash Table and Red-Black Tree Using Locks and Transactions Without Nesting

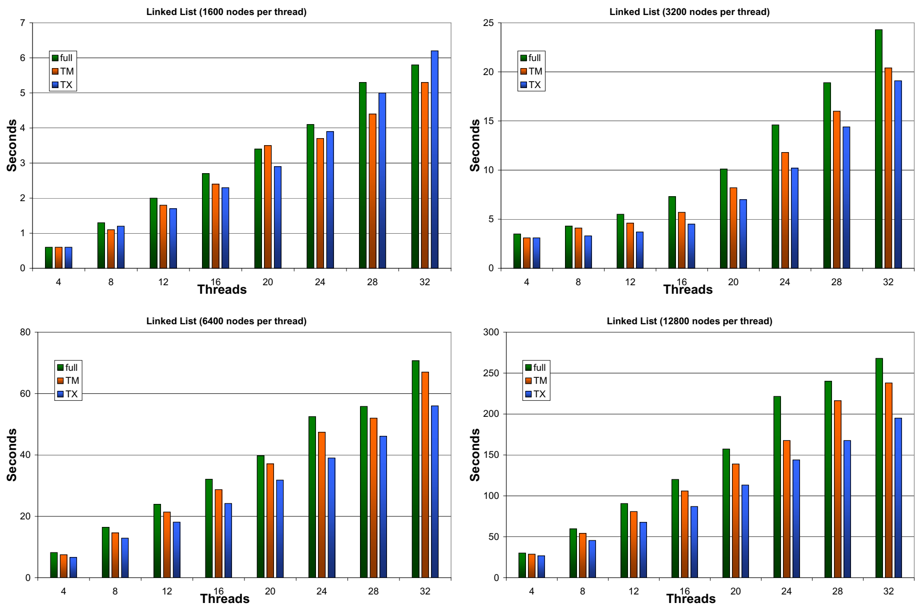


Fig. 9. Linked List With Locks Nested Inside of Transactions

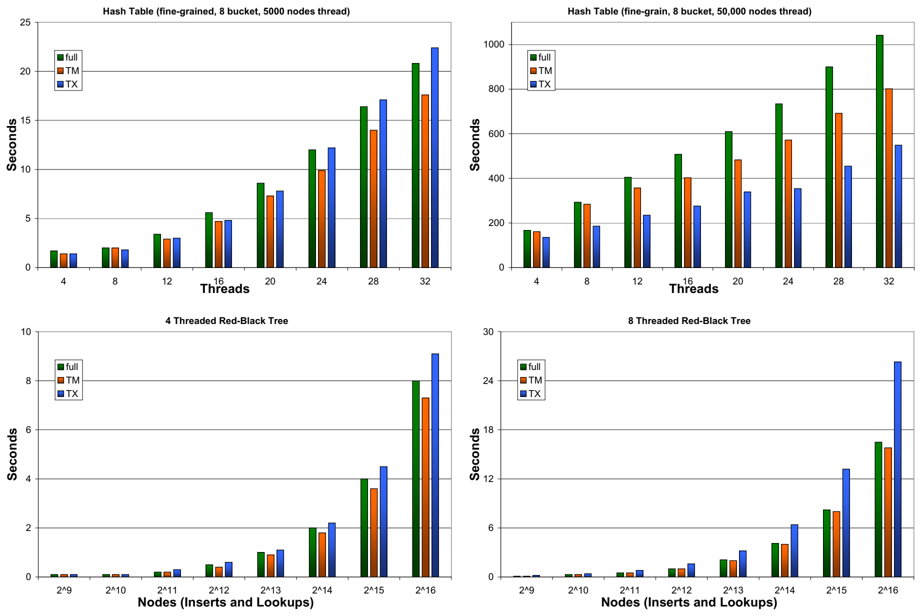


Fig. 10. Hash Table and Red-Black Tree With Locks Nested Inside of Transactions

Each benchmark was run using full lock protection (left bar), coarse- (abbreviated TM, middle bar), and fine-grained (abbreviated TX, right bar) algorithms.

For cases where locks are nested within transactions, shown in Figures 9 and 10, we use a similar model to those used in non-nested executions. We use the same basic 4-threaded model as described above, except that each fourth thread's transactions nest calls to lock-based insert and lookup operations. In Figure 10, we slightly deviate from this model to isolate the red-black tree performance using only 4- and 8-threaded experiments while doubling the tree size each iteration. These benchmarks provide insight into the performance degradation of the fine-grained algorithm.

6.1 Performance Summary

Our experimental results are surprising. The coarse-grained algorithm (abbreviated as `TM` in the benchmarks) consistently outperforms full lock protection, while the fine-grained algorithm (abbreviated as `TX` in the benchmarks) ranges from $\approx 2x$ faster to $\approx 2x$ slower than either of the other approaches. Our initial results seem to indicate that the coarse-grained algorithm is a better general candidate than the fine-grained algorithm, in its current form, for software that supports the concurrent execution of locks and transactions. This is because *(i)* the coarse-grained algorithm requires minimal additional code (e.g., each of our benchmarks only required one extra line of code for the coarse-grained algorithm) and *(ii)* its performance efficiency is consistently better than the prior systems for our experimental benchmarks.

These results are not intuitive from the analyses presented in Sections 5.1 and 5.2. Our experimental results capture what was missed from the mathematical analysis in Sections 5.1 and 5.2. That is, the fine-grained algorithm's computational overhead introduce latencies that can degrade overall program performance, even though it can increase the number of locks and transactions that can execute concurrently. Two general factors contribute to this. First, in order for the fine-grained algorithm to yield a performance benefit over the other algorithms, the cumulative critical section overhead when executed serially must be greater than the overhead incurred by the fine-grained algorithm. Second, the fine-grained algorithm must locate false conflicts that are overlooked by the other algorithms, which result in additional concurrent throughput. If both of these conditions are not satisfied, the fine-grained algorithm may not produce enough extra concurrent throughput to offset its algorithmic overhead and therefore it may perform worse than if it did no (full lock protection) or minimal (coarse-grained) conflict management.

An example of a benchmark that does not satisfy the above conditions can seen in the nested red-black tree benchmarks of Figure 10. As can be seen in the 4-threaded red-black tree nested benchmark (Figure 10), as the workload grows, there is a growing divide between the fine-grained algorithm and the other algorithms. This divide demonstrates that the critical section workload of the threads is less than the algorithmic overhead of the fine-grained algorithm, but greater than the other algorithms. Comparing the 4-threaded and 8-threaded red-black tree nested benchmarks to each other (again, Figure 10), one can observe an increased performance degradation of the fine-grained algorithm in the 8-threaded red-black tree compared to the 4-threaded red-black tree. This illustrates that the algorithmic overhead of the fine-grained algorithm is greater than the extra concurrency reclaimed from the false conflicts it finds, because when more threads are added to the benchmark, the fine-grained algorithm performs worse, not better, than the other algorithms.

7 Conclusion

While TM shows promise for future software programs, most TMs have undefined behavior for the concurrent execution of locks and transactions when they are used to synchronize the same shared-memory. This paper presented a performance study between our system and prior works that allow locks and transactions to execute in the same program.

We introduced two new language constructs: the `TmLock` and the extended `atomic` block. We analyzed how these constructs worked with two algorithms at different granularities. Our `TmLock` data structure combined with a coarse-grained algorithm yielded $\approx 1.5x$ improved program performance compared to prior systems and never performed worse than those systems. Our fine-grained algorithm provided up to a $\approx 2.0x$ performance improvement but, in some cases, resulted in a $\approx 2.0x$ performance degradation compared to prior work.

References

1. Abdelkhalek, A., Bilas, A.: Parallelization and performance of interactive multiplayer game servers. In: IPDPS (2004)
2. Bloch, J.: Effective Java, 2nd edn. The Java Series. Prentice Hall PTR, Upper Saddle River (2008)
3. Dijkstra, E.W.: Solution of a problem in concurrent programming control. Commun. ACM 8(9), 569 (1965)
4. Gottschlich, J.E., Siek, J.G., Vachharajani, M., Winkler, D.Y., Connors, D.A.: An efficient lock-aware transactional memory implementation. In: Proceedings of the International ACM Workshop on ICOOOLPS (July 2009)
5. Gottschlich, J.E., Vachharajani, M., Siek, J.G.: An efficient software transactional memory using commit-time invalidation. In: Proceedings of the International Symposium on Code Generation and Optimization, CGO (April 2010)
6. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: Proceedings of the International Symposium on Computer Architecture (May 1993)
7. Moore, K.E., Bobba, J., Moravan, M.J., Hill, M.D., Wood, D.A.: LogTM: Log-based transactional memory. In: HPCA, pp. 254–265. IEEE Computer Society (February 2006)
8. Rajwar, R., Bernstein, P.A.: Atomic transactional execution in hardware: A new high performance abstraction for databases. In: Workshop on High Performance Transaction Systems (2003)

9. Rajwar, R., Goodman, J.R.: Speculative lock elision: enabling highly concurrent multithreaded execution. In: MICRO, pp. 294–305. ACM/IEEE (2001)
10. Rossbach, C.J., Hofmann, O.S., Porter, D.E., Ramadan, H.E., Aditya, B., Witchel, E.: Txlinux: using and managing hardware transactional memory in an operating system. In: SOSP, pp. 87–102. ACM (2007)
11. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of the Principles of Distributed Computing (August 1995)
12. Spear, M.F., Marathe, V.J., Scherer III, W.N., Scott, M.L.: Conflict Detection and Validation Strategies for Software Transactional Memory. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 179–193. Springer, Heidelberg (2006)
13. Spear, M.F., Michael, M.M., Scott, M.L.: Inevitability mechanisms for software transactional memory. In: Proceedings of the 3rd ACM SIGPLAN Workshop on Transactional Computing (February 2008)
14. Usui, T., Behrends, R., Evans, J., Smaragdakis, Y.: Adaptive locks: Combining transactions and locks for efficient concurrency. Journal of Parallel and Distributed Computing, 1009–1023 (2010)
15. Volos, H., Goyal, N., Swift, M.M.: Pathological interaction of locks with transactional memory. In: TRANSACT (February 2008)
16. Welc, A., Saha, B., Adl-Tabatabai, A.-R.: Irrevocable transactions and their applications. In: SPAA (2008)
17. Ziarek, L., Welc, A., Adl-Tabatabai, A.-R., Menon, V., Shpeisman, T., Jagannathan, S.: A Uniform Transactional Execution Environment for Java. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 129–154. Springer, Heidelberg (2008)
18. Zilles, C., Flint, D.: Challenges to providing performance isolation in transactional memories. In: Proceedings of the Fourth Workshop on Duplicating, Deconstructing, and Debunking, pp. 48–55 (June 2005)
19. Zyulkyarov, F., Gajinov, V., Unsal, O.S., Cristal, A., Ayguadé, E., Harris, T., Valero, M.: Atomic quake: using transactional memory in an interactive multiplayer game server. In: PPoPP, pp. 25–34. ACM, New York (2009)

A Study of the Usefulness of Producer/Consumer Synchronization

Hao Lin, Hansang Bae, Samuel P. Midkiff, Rudolf Eigenmann, and Soohong P. Kim

School of Electrical and Computer Engineering, Purdue University[⋆], West Lafayette, IN, USA
http://www.ece.purdue.edu/

Abstract. In the early 1980s, shared memory mini-super-computers had buses and memory whose speeds were relatively fast compared to processor speeds. This led to the widespread use of various producer/consumer (post/wait) synchronization schemes for enforcing data dependences within parallel *doacross* loops. The rise of the "killer micro", instruction sets optimized for serial programs, and rapidly increasing processor clock rates driven by Moore's law, led to special purpose synchronization instructions being replaced by software *barriers* combined with loop fission (to allow the barriers to enforce dependences.) One cost of this approach is poorer cache behavior because variables on which a dependence exists are now accessed in separate loops. With the advent of the multicore era, producer/consumer synchronization again appears plausible. In this paper we compare the performance of hardware and software synchronization schemes to barrier synchronization, and show that either hardware or software based producer/consumer synchronization can provide applications with superior performance.

Keywords: Producer/consumer synchronization, chip multiprocessors, barrier synchronization.

1 Introduction

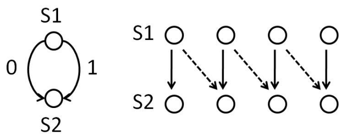
The dominance of chip multiprocessors has ushered in the second era of widespread shared memory parallel computing. The first era was in the 1980s, when mini-supercomputer companies such as Tera and Alliant had a significant market presence. A salient property of the super-minis was memory, processor and bus speeds that were much more balanced than in today's machines. This balance allowed inter-thread and inter-processor producer/consumer (ordering) synchronization to occur in one or a small number of instruction cycles.

The rise and dominance of the "killer micro" in the early to mid-1990s put an end to the super mini-computers. Driven by Moore's law, processor speeds

[⋆] This work was supported, in part, by the National Science Foundation under grants number 070793-CNS, 0833115-CCF and 1012099-CCF. All opinions and conclusions are those of the authors and not the NSF.

increased rapidly, outstripping memory and bus speeds. As well, the broad, sequential general purpose computing market that drove the development of these processors did not need producer/consumer synchronization, and so no design effort was spent implementing it. To cope with this, parallel programs targeting these chips converged on a model similar to what is provided by OpenMP and Pthreads – support for critical and atomic sections is provided, along with barrier synchronizations across all threads. Under this model, dependences are synchronized by placing a barrier between the source and sink accesses of the dependence. To do this often requires *loop fission*, which places the dependence source in the loop that executes first, and the dependence sink in the loop that executes later, with a barrier between the two (as shown in Fig. 1(b)). In addition to the overhead of executing two loops instead of one, temporal locality between the dependent references is destroyed, and performance can suffer as a result.

```
     for (j = 0; j < size - 1; j++) {
S1     array1[j + 1] = array2[j];
S2     array3[j] =
             array1[j + 1] - array1[j];
     }
```



(a) A serial program fragment and its dependences

```
     #pragma omp parallel for
     for (j = 0; j < size - 1; j++) {
S1     array1[j + 1] = array2[j];
     }
implicit barrier
     #pragma omp parallel for
     for (j = 0; j < size - 1; j++) {
S2     array3[j] = array1[j + 1]
                   -array1[j];
     }
```

(b) A parallel version of the fragment of (b) after loop fission and inserting a barrier

Fig. 1. An example loop parallelization using barrier synchronization

This paper makes the following contributions. First, we present a preliminary study of the performance benefits of three major synchronization schemes: (i) barriers (both GOMP and user space barriers); (ii) hardware produce-consumer synchronization; and (iii) software producer/consumer synchronization.

Second, we provide experimental results from three benchmarks showing that GOMP barriers have high overheads. "User space" barriers, implemented as part of the program, give better performance. The best performance comes from producer/consumer synchronization, with software and hardware implementations having similar, but not identical, performance. We describe, and use, an implementation of producer/consumer synchronization in the Simics simulator and Intel i7 860 to obtain our results.

Third, we show that in the benchmarks we study, the differences in performance between the synchronization operations and strategies are not a result of what they allow to be parallelized but arise in large part from differences in the cache performance of the different strategies. Thus, the difficulty of synchronizing doacross type loops with no dependence cycles using currently supported mutex and barrier synchronization does not appear to be a significant impediment to performance in these benchmarks.

Finally, we show that additional transformations, such as loop blocking, are necessary to achieve good performance even with fused loops and producer consumer synchronization. Such transformations can also maintain good performance when the cost of producer/consumer synchronization increases.

The rest of the paper is organized as follows. The next section briefly discusses some necessary background material. Section 3 describes the synchronization techniques used in this paper. This section also covers compiler techniques like loop fusion, loop skewing and loop blocking that aim to increase data reuse. Simulation experiments and preliminary results are given in Section 5. Finally, we discuss our conclusions in Section 6.

2 Background and Related Work

2.1 Locality with Loop Transformations

Much research has been done on analytical models and algorithms to perform loop fusion and fission [10] to enhance parallelism and increase locality. Kennedy and McKinley proposed an algorithm for loop fusion to maximize parallelism, and a polynomial time algorithm to improve locality. Manjikian and Abdelrahman [13] present systematic techniques to perform loop fusion and parallelization by “peeling” off iterations. Qasem and Kennedy [17] propose a cache-conscious analytical model for profitable loop fusion with an empirical tuning framework, but are concerned with locality, not parallelism.

Loop blocking partitions the iteration space into blocks so that the data in blocks fits in cache. Wolf and Lam [20] present a complete approach to loop tiling to improve cache locality. Unroll-and-jam is also proposed as important transformation to increase parallelism by better instruction scheduling in software pipelining [4].

In Section 4 we use these techniques when inserting synchronization to both amortize the cost of synchronization over more references and to increase locality.

2.2 Synchronization and the Cost of Barriers

Data dependences that cross the processor core boundaries must be synchronized, and one way to synchronize these dependences is to use barriers. Traditional software barriers can be quite effective in some loops or with coarser grain parallelism [5]. Many popular libraries such as Pthreads and OpenMP have barrier implementations. However, software-only barriers do not always give speedups due to their relatively high latency. With CMP architectures, fast hardware barriers have been proposed because faster communication is possible within a single chip than across chips [11,18].

2.3 Producer/Consumer Synchronization and Its Benefits

Producer/consumer synchronization to support the synchronization of data dependences has been studied for decades. Midkiff and Padua [15] described several schemes to generate and optimize the placement of `post` and `wait` primitives. Su and Yew [19] present the *Process-Oriented* technique that realizes statement level synchronization and its application. However, they did not perform a performance evaluation of their scheme.

In [9], Kejariwal compares a bit vector based method with other three software data synchronization methods: SYS, MAP and MYS. In SYS and MAP, threads share the same synchronization signal, so the signal must be updated in a sequential order. These schemes have little space overhead, but can lose parallelism since they must serialize the "posts". MYS resolves this problem by using one synchronization signal per thread instead of one per dependence. The loop containing the wait primitive of MYS can add overhead as it potentially reads elements from both synchronization arrays. The effect of MYS on data locality and cache performance, compared to barrier schemes, is not studied.

With faster communication on a single chip available in CMP architectures, hardware data synchronization support has been proposed. The Synchronization State Buffer [22] is simulated on a cacheless architecture, the IBM C64, and shows better performance than SYS, MAP and MYS.

3 Synchronization Primitives Studied in This Paper

3.1 Synchronization Barriers

GNU OpenMP uses `gomp_barrier_wait` for barrier synchronization. A shared counter variable, `arrived`, is used to hold the number of threads that have joined the barrier. When a thread joints the barrier, `arrived` is incremented and the thread waits by calling `gomp_sem_wait()`. After the last thread joins the barrier, it serially signals all the other threads, and each signaled thread then decrements `arrived`, which sets it back to zero.

GOMP barriers make a system call, and have a high system overhead. To avoid system calls and the associated overhead, we have implemented the user-space centralized barrier with sense reversal of [6]. It uses a shared counter and a boolean type local "sense" variable to realize the semantics of multiple barriers.

3.2 Producer/Consumer Synchronization

Because the purpose of this paper is not to propose a better synchronization scheme, but to present comparative results on the impact of reasonable synchronization schemes on program performance, we use a simple "post" and "wait" mechanism [15], with a straight forward but reasonable algorithm to insert the synchronization into loops.

We use a *program position* as the value assigned to each synchronization variable. Similar to a *process counter* [19], the program position in the loop nest is a vector $\langle i_1, i_2, ..., i_n, stmt\rangle$ that gives the current completed statement instance in the source program. The *stmt* element in the vector uniquely identifies the statement whose execution is being signaled, and i_1, i_2, ..., i_n are the iteration values of the n loops that surround the statement *stmt*. The program position vector enables us to signal multiple, and multi-level, dependences. For the code in Fig. 1, the position of `S1` in iteration `j = 3` can be expressed as `<3, 1>`. Since there is only one dependence in the example, we only need to use the iteration number in the vector, i.e. `<3>`. The comparison of position vectors X and Y, is a lexicographic comparison, i.e.

$$X \geq Y \text{ iff } \; x_1 > y_1, \text{ or } x_1 = y_1 \text{ and } \langle x_2, x_3, ..., x_n\rangle \geq \langle y_2, y_3, ..., y_n\rangle \, .$$

The technique of this paper, *PCS* (producer/consumer synchronization), logically uses an array, `pcsync` with one element for each thread. The `pcsync` for thread i holds the process counter vector representing the currently executing position in thread i. A multi-threaded program using PCS performs the following synchronization operations: (1) an initialization phase, to reset the synchronization variables to zero; (2) a posting phase, to allow a thread to post a signal to its own synchronization variables; and (3) a wait phase, where one thread waits for a signal in the synchronization variable of another thread. Thus, PCS implements three primitives: `pcs_init`, `pcs_post`, `pcs_wait`.

A post primitive is inserted immediately after the source of a synchronized dependence, and the wait primitive is placed immediately before the sink. In the `pcs_wait` primitive, the synchronization variable of the thread executing the source holds the last completed position of that thread, which is compared to the source position. If the value of the synchronization variable is greater than, or equal to, source vector value, we know the execution of the dependence source statement is completed, so we can terminate the spin loop and execute the dependence sink statement. Otherwise, the source has not been completed, and the waiting thread must wait. Pseudo-code of the three primitives is listed in Fig. 2.

```
pcs_init ()
  doi = 1, nthreads
    pcsync (i)
        = zero position
  end do
```

(a) **pcs_init**

```
pcs_post (tid, position)
  memory fence
  pcsync (tid) = position
```

(b) **pcs_post**

```
pcs_wait (tid, position)
  while (pcsync (tid) < position)
  memory fence
```

(c) **pcs_wait**

Fig. 2. Pseudo-code of the three PCS primitives, pcs_init, pcs_post, and pcs_wait

Memory fence instructions are needed to prevent hardware from reordering guarded instructions past the synchronization instruction. We use an IA32 Intel `MFENCE` memory fence instruction to force the serialization of both load and store instructions with the respect to synchronization instructions at the hardware level. `asm volatile("":::"memory")` is used to prevent reordering of instructions past synchronization by the compiler.

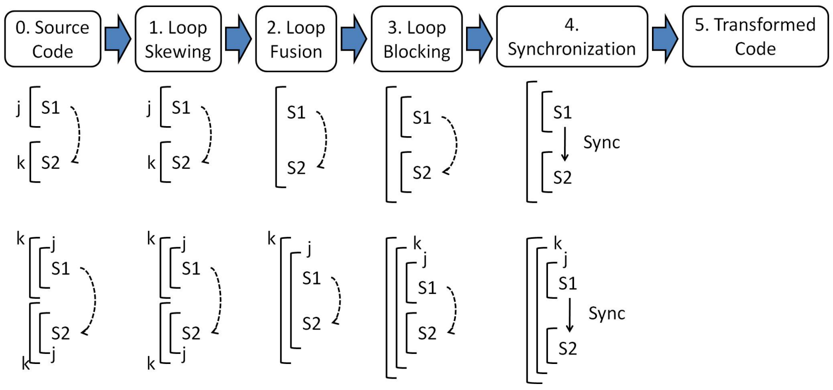


Fig. 3. Overview of the transformation framework

3.3 Hardware Support for Producer/Consumer Synchronization

To implement the `pcsync` array, we use a group of *synchronization registers*, called PCSReg, that are connected to CPU cores through the PCSReg-CPU interconnect. The number of the such registers is at least as large as the number of threads so each thread has a register. The default PCSReg-CPU interconnect has a latency of one cycle.

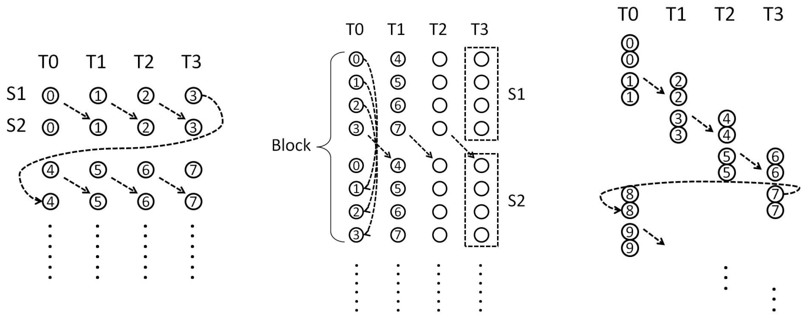
The `pcs_post` primitive stores the position vector into the corresponding thread's register. The `pcs_wait` instruction loads the register from the posting thread and tests it. Because the `pcs_wait` spin-waits, if the test fails the `pcs_wait` reloads the register, performs the test again, and repeats this until the test succeeds.

4 Parallelizing `doacross` Loops with Synchronization Primitives

This section discusses a general transformation framework integrating loop skewing, loop fusion, loop blocking and synchronization insertion, to evaluate the benefits of performing loop fusion and synchronization. A block diagram of the steps performed by the framework is shown in Fig. 3.

Loop skewing is applied to enable legal multi-level loop fusion, which in turn reduces the temporal reuse distance of dependences in a distributed loop nest. Loop blocking aims to improve the data spatial locality and reduce false sharing. Finally, barriers or producer/consumer synchronization primitives are inserted to enable parallelism.

We use the `Cetus` compiler infrastructure [8] as our dependence analysis tool to discover transformation opportunities in multiple loops that contain only forward dependences. All transformations are implemented by hand.



(a) **Cyclic scheduling before blocking** (b) **Cyclic blocking** (c) **Blocking in pipeline fashion**

Fig. 4. Example of the effect of loop blocking on an iteration space

4.1 Loop Transformations

Loop blocking is a loop transformation that can enhance cache performance. By keeping each block small enough, cache capacity misses are reduced and temporal reuse is improved.

We first illustrate the transformations we perform with an example of loop blocking, using code like that found in Fig. 1. A cyclic scheduling of the iterations is shown in Fig. 4(a). In Fig. 4(b), we block four successive iterations together into one task and assign tasks to threads in a round robin fashion. Four dependence source statements (`S1`) are executed first, followed by the four dependence sink statements (`S2`). The dashed line represents the forward dependence in the original program. Instead of executing source and sink statements one after the other, we execute several source statements followed by several sink statements. The reason for this is to allow the source and sink statements to execute in parallel instead of in a pipelined fashion, as in Fig. 4(c). Loop blocking can reduce the number of synchronizations while having a minimal impact, at worst, on parallelism.

False sharing is another hazard that can be reduced by loop blocking. In Fig. 4(a), adjacent array elements will likely be located in the same cache line, and writes to an element will invalidate cached copies in other processors. All threads will invalidate each other's cache entries, which will generate a large number of cache misses and inter-core traffic. By performing loop blocking with properly sized blocks, array elements in the same cache line are almost always accessed by the same thread, reducing false sharing.

From the previous analysis, having the proper block size is important. If the block size is too small the program may suffer from large synchronization overheads and false sharing cache misses; if it is too large cache capacity misses and workload imbalance can result.

4.2 Inserting Barrier Synchronization

Again considering the example of Fig. 3, after applying loop skewing, loop fusion and loop blocking, the loop nests now look like those shown in Fig. 3 under "Loop Blocking". However, the forward dependence (the dashed line) still exists between the inner loop nests and prevents the exploitation of finer grain parallelism. Synchronization is used to enforce the dependence and enable parallelization of the outer loop.

Barrier synchronization is inserted between every pair of inner loops, as shown in the fourth phase of Fig. 3 ("Synchronization"). The transformed example code is shown in Fig 5 as line 8 (ignoring line 1, 9 and 10). We call this version *fine-grained barrier with loop fusion and blocking* in the next section.

4.3 Inserting Producer/Consumer Synchronization

The parallel version of a fused blocked loop with PCS primitives is shown in lines 1, 9 and 10 of Fig. 5 (ignoring line 8). Before the parallel section starts, we initialize the synchronization variables. Suppose that iteration `k` is mapped onto thread T_0, then after executing the first statement, T_0 sets the value of `pcsync[0]` to be $\langle k \rangle$ by executing `pcs_post(0, k)`. If another thread T_1 waiting for iteration `k` on thread T_0, the wait primitive will wait until thread T_0 finishes the first statement in iteration `k`.

```
1     pcs_init()
2     #pragma omp parallel private (j, k, tid, end)
3     {
4       tid = omp_get_thread_num();
5       for (j = tid * block_size; j < size - 1; j+ = block_size * nthreads) {
6         for (k = j; k < end; k + +)
7           array1[k + 1] = array2[k];
8         barrier;
9         pcs_post (tid, k);
10        pcs_wait ((tid-1)%nthreads, j);
11        for (k = j; k < end; k + +)
12          array3[k] = array1[k + 1] - array1[k];
13      }
14    }
```

Fig. 5. Transformed code of the example in Fig. 1 after loop fusion, blocking, parallelization and synchronizations. Line 7 (without line 8 and 9) shows how barrier synchronization is inserted. Line 8 and 9 (without line 7) shows how PCS synchronization is inserted.

5 Experimental Evaluation

The evaluation of both software and hardware implementations is performed in two different ways: using *Simics*-3.0.31 for the IA32 instruction set architecture [12], and a real machine with an Intel®Core^{TM}i7 2.80GHz multiprocessor. Simics is a a cycle-accurate full-system shared memory multi-core processor simulator that models all components of the IA32 and Linux operating system. We created four Simics target configurations (2, 4, 8 and 16 core) and all four systems are running Fedora Core 5 (FC5), hosting Linux kernel 2.6. GEMS's Ruby models the cache memory hierarchy using a cycle-accurate timing model. Ruby models the timing of caches, cache controllers, interconnect, and memory [14]. Ruby is configured with a 32kB private L1 I-cache and D-cache for each processor core, and a shared L2 banked cache with one bank per core. We use the `MOESI_CMP_directory` provided by Ruby for the coherence protocol. It is a two-level directory protocol using non-inclusive L1/L2 caching with blocking caches [2]. Table 1 shows the configuration for target system. Experiments were performed on real hardware using an Intel i7 860 quad-core running at 2.8GHz with 32kB/246kB/8MB of private L1/shared L2/shared L3 cache, respectively. Experiments were run using Ubuntu 10.4, Linux Kernel 2.6, with the same compiler version and switches as the simulated experiments.

We first measure the cost of the software and hardware versions of the PCS scheme and the GOMP and software barrier schemes, using the EPCC OpenMP Microbenchmarks [3]. These costs are reported in cycles in Table 2.

Table 1. The hardware configuration

Processor	Intel Pentium 4, in-order with 2, 4, 8, 16 cores
Private L1 Cache	32kB 4-way, 64Byte-line, L1 hit latency: 1-cycle
Shared L2 Cache	32kB per bank
Cache Coherence Protocol	Ruby's MOESI_CMP_directory
Memory	200-cycle memory access
OS	Fedora Core 5 (Tango machine)
Compiler	GCC 4.3 with OpenMP, flag = -fopenmp -O3

Table 2. Costs of synchronization primitives

software post	30-100 cycles
software wait	100-200 cycles
hardware post	3-5 cycles
hardware wait	6-7 cycles
GOMP barrier	150-650 cycles
user-space barrier	100-400 cycles

5.1 Performance Evaluation

We applied our strategy of loop transformations and synchronization insertion to three benchmarks: `membar`, a micro benchmark we developed and is shown in Fig. 1; `laplace1D`, the kernel extracted from the 1-D Laplace equation solver [21]; and `ll18`, a 2-D explicit hydrodynamics kernel from the Livermore Loops [1]. We compare the results of six versions of each benchmark:

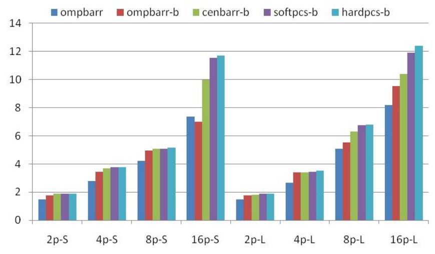
1. The serial or original version (`seq`);
2. A version using a coarse-grained omp barrier (`ompbarr`);
3. A version using a finer-grained omp barrier with loop fusion and blocking (`ompbarr-b`);
4. A version using a finer-grained central barrier with loop fusion and blocking (`userbarr-b`);
5. A version using software producer/consumer synchronization with loop fusion and blocking (`softpcs-b`);
6. A version using hardware producer/consumer synchronization with loop fusion and blocking (`hardpcs-b`).

We set the `GOMP_CPU_AFFINITY` environment variable to minimize the effects of context switches and to ensure that threads are mapped to different processor cores. Two input data sets, small and large, are used on both on 2-, 4-, 8- and 16-core simulations and 2, 4 and 8 threads executions the real hardware. So for each version of a benchmark, we measure eight cases on the simulator (labeled 2p-x, 4p-x, 8p-x and 16p-x, where "x" is "L" or "S" for the large and small data sets, respectively.) On the real hardware, for each version of the benchmark we measure six cases (labeled 2p-x, 4p-x and 8p-x, where "x" is as above. On the real hardware processor, we measure the first five versions, i.e. all but `hardpcs-b`. Different block sizes for loop blocking were measured, and the best was chosen for each benchmark and data set.

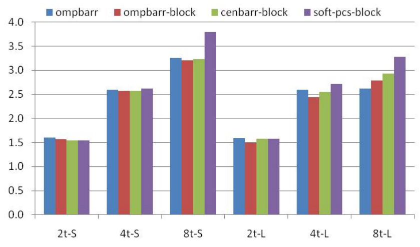
The potential performance improvement from using producer/consumer synchronization with loop transformations can come from exploiting finer grain parallelism, reducing cache misses, memory accesses, and even coherence protocol messages. We measure both speedups and cache misses in our performance evaluation.

Membar. The pseudo-code of each version of this micro benchmark is shown in Fig. 1. Fig. 6 reveals the speedups and cache misses of parallel versions 2 through 6, compared to the speedup of the original version which is normalized to 1. The data sets used are 2^{16} (small) and 2^{17} (large) double-precision array elements in the simulator case and 2^{19} (small) and 2^{23} (large) for the real hardware.

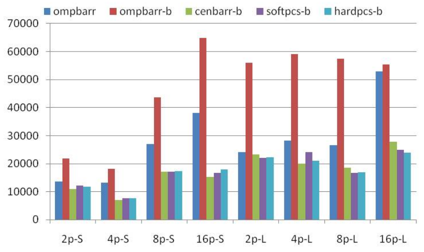
We conclude from Fig. 6 (a) and (b) that in most of the cases with a higher core count, the finer-grained parallel implementations (3-6) generally have superior performance compared to the coarse-grained implementation (2), and producer/consumer synchronization performance is better than that of barriers. The producer/consumer synchronization is less affected by core count, and shows better performance at a higher number of processors. Hardware producer/consumer



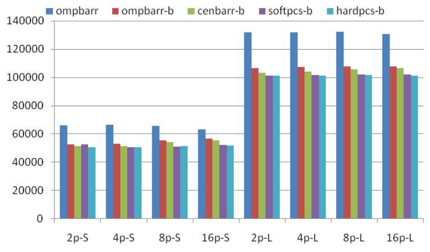
(a) **Speedups with Simics simulator**



(b) **Speedups with Intel processor**



(c) **L1 cache misses**



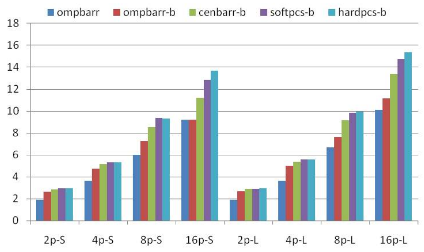
(d) **L1 user cache misses**

Fig. 6. Performance results for the micro benchmark `membar`

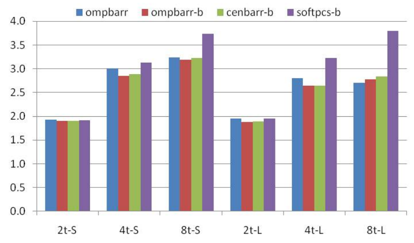
synchronization has approximately a 50% speedup over the coarse-grain barrier version. The difference between software and hardware implementations of producer/consumer synchronization is not large, usually between 5% to 10%. Note that in the case of 16p-S, 8t-S and 8t-L, the `ompbarr-b` version slows down to the general barrier version's performance. We conjecture that the reason is the high overhead of the GOMP barrier with small block sizes.

Fig. 6(c) and (d) present the system and user cache misses in both L1 D-cache and L2 cache. The distributed loop with general OpenMP barrier (`ompbarr`) causes more cache misses than in the sequential program. The `gomp barrier` shows a large number of system cache misses. As blocking leads to more barriers being executed, `ompbarr-b` leads to even more cache misses than `ompbarr`. The user-space barrier implementation has relatively fewer system cache misses. Trends in the data (fused and blocked versions have fewer misses) lead us to conclude, as have many others, that loop fusion and blocking help reduce the number of cache misses. Producer/consumer synchronization has the lowest overhead, the fewest cache misses and the best performance.

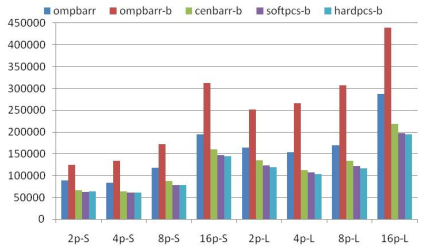
Kernel from 1-D Laplace Equation Solver. The 1-D Laplace solver uses a finite difference method to approximate the numerical solution to a partial differential equation [21]. It is quite similar to the `membar` benchmark except that it performs more computation and there is an enclosing serial time step loop. We take the same strategy and same input data set as `membar`, and the results are presented in Fig. 7. As expected, the results show similar trends to `membar` but with higher speedups. With increasing core counts, barrier overheads



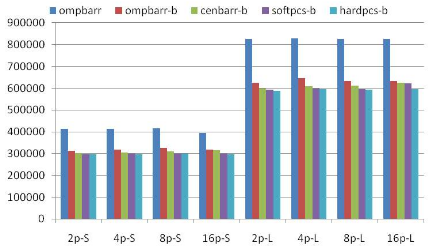
(a) **Speedups with Simics simulator.**



(b) **Speedups with Intel processor.**



(c) **L1 cache misses**



(d) **L1 user cache misses**

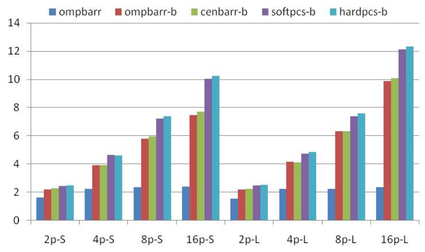
Fig. 7. Performance result for the 1-D Laplace Solver Kernel

increase, and speedups rise more slowly. The producer/consumer synchronization is less affected by core count, and shows better performance at a higher number of processors. Note that in the 2-core, 4-core and 8-core configurations, producer/consumer synchronization achieves a speedup greater than the core count. This superlinear speedup is the result of the cache performance being much improved relative to the serial program.

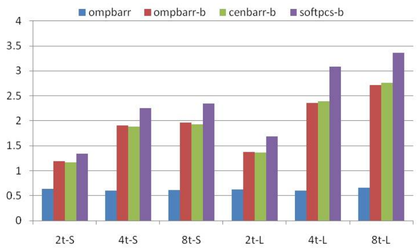
The Intel processor results are similar to the simulation results for 2- and 4-processors (with one thread per processor). In the 8-thread version (hyperthreaded), the performance is limited by the core count.

2-D Explicit Hydrodynamics Kernel (Livermore Loop 18). The `ll18` kernel has three two-dimensional loop nests, all of which have the same loop bounds. Two data sets with 128×128 and 256×256 double-precision (8-byte) floating point values are used on Simics, while 256×256 and 2048×2048 are used on Intel. Qasem [17] states that all three loop nests can be fused in two levels to obtain the best cache performance. We can fuse all three outer loops for the finer-grained barrier synchronization. The difference from the previous two benchmarks is that we need two elements in the synchronization position vector introduced in section 3.2.

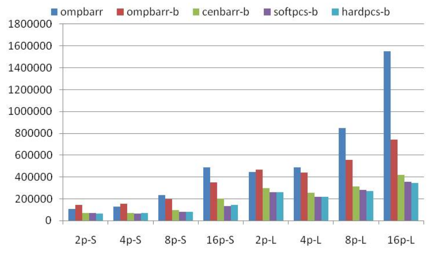
For two-dimensional loop blocking, we cyclically assign one outer loop iteration to each thread, with inner loop iterations blocked. The results for `ll18` are shown in Fig. 8.



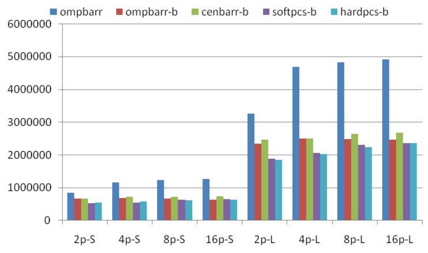
(a) **Speedups in Simics simulator**



(b) **Speedups in Intel processor**



(c) **L1 cache misses**



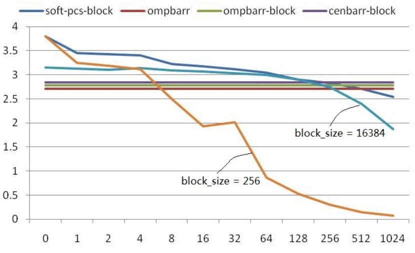
(d) **L1 user cache misses**

Fig. 8. Performance result for `ll18`

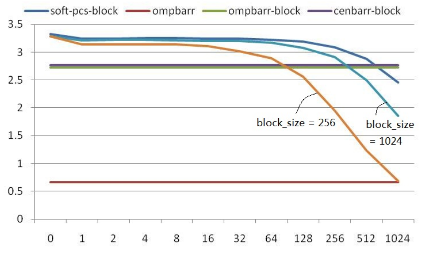
5.2 Evaluation of Impact of the Synchronization Cost

To study the impact of synchronization cost on the performance of DOACROSS loops, we conduct an experiment to present how speedups vary with different synchronization costs. In the `pcs_post` and `pcs_wait` primitives, we insert a delay loop to control the cost of both primitives by varying the number of loop iterations.

The results of `laplace-1D` and `ll18` is listed in the Fig 9. Both of the two results are obtained with a Large data set and eight threads (8t-L). The three straight lines are the three barrier versions. If we fix the loop blocking size (i.e. 256, 16384 in `laplace` and 256, 1024 in `ll18`), we can see that speedups



(a) **laplace.**



(b) **ll18.**

Fig. 9. Performance for PCS primitives with different costs

of software producer/consumer synchronization drops dramatically as its cost increases. But if we can tune the block size for the best performance, we can maintain almost as good performance even when delays are fairly high. When the overheads of the PCS primitives increase, increasing the block size reduces the number of synchronization instructions executed, and the program achieves speedups almost as high as with smaller synchronization instruction costs.

6 Conclusions and Future Work

This paper has presented a preliminary study, using small benchmarks, of the utility of producer/consumer synchronization. In the future we plan to extend this study to whole-program benchmarks. To summarize, we conclude:

1. By providing flexibility to compilers in how to structure parallel loops, producer/consumer synchronization can improve locality and cache performance, increasing overall program performance;
2. Both software and hardware implementations of producer/consumer synchronization provide performance benefits over barriers by exploiting more fine-grained parallelism;
3. Pthreads barriers that require system calls have a negative effect on cache performance, and work should be done to improve this.
4. With the help of loop transformations, the performance of producer/consumer synchronization may be relatively insensitive to its cost.

An interesting question not answered in this study is how much additional parallelism in loops with dependence cycles can be exploited using producer consumer synchronization on multi-cores. During the period of the "super-minicomputer", exploitation of parallelism in less than 100% doacross [16,7] loops (i.e. loops with lexically backward dependences) was a motivator of producer/consumer synchronization.

References

1. Livermore Loops in C Version, `http://www.netlib.org/benchmark/livermorec`
2. Multifacet GEMS Wiki, Protocol,
`http://www.cs.wisc.edu/gems/doc/gems-wiki/moin.cgi/Protocols`
3. Bull, J.M., O'Neill, D.: A Microbenchmark Suite for OpenMP 2.0. SIGARCH Comput. Archit. News 29, 41–48 (2001)
4. Carr, S., Ding, C., Sweany, P.: Improving Software Pipelining With Unroll-and-Jam. In: Proceedings of the 29th Hawaii International Conference on System Sciences, HICSS 1996. Software Technology and Architecture, vol. 1, pp. 183–192. IEEE Computer Society, Washington, DC (1996)
5. Chen, D.K., Su, H.M., Yew, P.C.: The Impact of Synchronization and Granularity on Parallel Systems. In: Proceedings of the 17th Annual International Symposium on Computer Architecture, ISCA 1990, pp. 239–248. ACM (1990)
6. Culler, D., Singh, J., Gupta, A.: Parallel Computer Architecture: A Hardware/Software Approach, 1st edn. Morgan Kaufmann (1998)

7. Cytron, R.: Doacross: Beyond Vectorization for Multiprocessors. In: ICPP, pp. 836–844 (1986)
8. Dave, C., Bae, H., Min, S.J., Lee, S., Eigenmann, R., Midkiff, S.: Cetus: A Source-to-Source Compiler Infrastructure for Multicores. Computer 42, 36–42 (2009)
9. Kejariwal, A., Saito, H., Tian, X., Girkar, M., Li, W., Banerjee, U., Nicolau, A., Polychronopoulos, C.D.: Lightweight Lock-free Synchronization Methods for Multithreading. In: Proceedings of the 20th Annual International Conference on Supercomputing, ICS 2006, pp. 361–371. ACM, New York (2006)
10. Kennedy, K., McKinley, K.S.: Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution. In: Banerjee, U., Gelernter, D., Nicolau, A., Padua, D.A. (eds.) LCPC 1993. LNCS, vol. 768, pp. 301–320. Springer, Heidelberg (1994)
11. Kim, S.P., Midkiff, S.P., Dietz, H.G.: Hardware Support for OpenMP Collective Operations. In: Gao, G.R., Pollock, L.L., Cavazos, J., Li, X. (eds.) LCPC 2009. LNCS, vol. 5898, pp. 31–49. Springer, Heidelberg (2010)
12. Magnusson, P., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., Werner, B.: Simics: A Full System Simulation Platform. Computer 35(2), 50–58 (2002)
13. Manjikian, N., Abdelrahman, T.S.: Fusion of Loops for Parallelism and Locality. IEEE Trans. Parallel Distrib. Syst. 8, 193–209 (1997)
14. Martin, M.M.K., Sorin, D.J., Beckmann, B.M., Marty, M.R., Xu, M., Alameldeen, A.R., Moore, K.E., Hill, M.D., Wood, D.A.: Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. SIGARCH Comput. Archit. News 33, 92–99 (2005)
15. Midkiff, S.P., Padua, D.A.: Compiler Algorithms for Synchronization. IEEE Transactions on Computers C-36(12), 1485–1495 (1987)
16. Padua, D.: Multiprocessors: Discussion of Some Theoretical and Practical Problems. Ph.D. thesis, University of Illinois, Urbana, Illinois, USA (1979)
17. Qasem, A., Kennedy, K.: A Cache-Conscious Profitability Model for Empirical Tuning of Loop Fusion. In: Ayguadé, E., Baumgartner, G., Ramanujam, J., Sadayappan, P. (eds.) LCPC 2005. LNCS, vol. 4339, pp. 106–120. Springer, Heidelberg (2006)
18. Sampson, J., Gonzalez, R., Collard, J.F., Jouppi, N.P., Schlansker, M., Calder, B.: Exploiting Fine-Grained Data Parallelism with Chip Multiprocessors and Fast Barriers. In: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 39, pp. 235–246. IEEE Computer Society, Washington, DC (2006)
19. Su, H.M., Yew, P.C.: On Data Synchronization for Multiprocessors. In: International Symposium on Computer Architecture, vol. 17, pp. 416–423 (1989)
20. Wolf, M.E., Lam, M.S.: A Data Locality Optimizing Algorithm. In: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation, PLDI 1991, pp. 30–44. ACM, New York (1991)
21. Zhu, W.: Synchronization State Buffer: Supporting Efficient Fine-grain Synchronization on Many-core Architectures. Ph.D. thesis, University of Delaware, Newark, DE, USA (2006)
22. Zhu, W., Sreedhar, V.C., Hu, Z., Gao, G.R.: Synchronization State Buffer: Supporting Efficient Fine-grain Synchronization on Many-core Architectures. In: Proceedings of the 34th Annual International Symposium on Computer Architecture, ISCA 2007, pp. 35–45. ACM, New York (2007)

Lock-Free Resizeable Concurrent Tries

Aleksandar Prokopec, Phil Bagwell, and Martin Odersky

École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland

Abstract. This paper describes an implementation of a non-blocking concurrent hash trie based on single-word compare-and-swap instructions in a shared-memory system. Insert, lookup and remove operations modifying different parts of the hash trie can be run completely independently. Remove operations ensure that the unneeded memory is freed and that the trie is kept compact. A pseudocode for these operations is presented and a proof of correctness is given – we show that the implementation is linearizable and lock-free. Finally, benchmarks are presented that compare concurrent hash trie operations against the corresponding operations on other concurrent data structures.

1 Introduction

In the presence of multiple processors data has to be accessed concurrently. Concurrent access to data requires synchronization in order to be correct. A traditional approach to synchronization is to use mutual exclusion locks. However, locks induce a performance degradation if a thread holding a lock gets delayed (e.g. by being preempted by the operating system). All other threads competing for the lock are prevented from making progress until the lock is released. More fundamentally, mutual exclusion locks are not fault tolerant – a failure may prevent progress indefinitely.

A lock-free concurrent object guarantees that if several threads attempt to perform an operation on the object, then at least some thread will complete the operation after a finite number of steps. Lock-free data structures are in general more robust than their lock-based counterparts [10], as they are immune to deadlocks, and unaffected by thread delays and failures. Universal methodologies for constructing lock-free data structures exist [9], but they serve as a theoretical foundation and are in general too inefficient to be practical – developing efficient lock-free data structures still seems to necessitate a manual approach.

Trie is a data structure with a wide range of applications first developed by Brandais [6] and Fredkin [7]. Hash array mapped tries described by Bagwell [1] are a specific type of tries used to store key-value pairs. The search for the key is guided by the bits in the hashcode value of the key. Each hash trie node stores references to subtries inside an array, which is indexed with a bitmap. This makes hash array mapped tries both space-efficient and cache-aware – the bitmap and the array can be stored within the same cache line. A similar approach was taken in the dynamic array data structures [8]. Hash array mapped tries are space-efficient and ensure that they are compressed as elements are being removed. They are well-suited for applications where the size bounds of the data structure are not known in advance and vary through time. In this paper we describe in detail a non-blocking implementation of the hash array mapped trie.

Our contributions are the following:

1. We introduce a completely lock-free concurrent hash trie data structure for a shared-memory system based on single-word compare-and-swap instructions. A complete pseudocode is included in the paper.
2. Our implementation maintains the space-efficiency of sequential hash tries. Additionally, remove operations check to see if the concurrent hash trie can be contracted after a key has been removed, thus saving space and ensuring that the depth of the trie is optimal.
3. There is no stop-the-world dynamic resizing phase during which no operation can be completed – the data structure grows with each subsequent insertion and removal. This makes our data structure suitable for real-time applications.
4. We present a proof of correctness and show that all operations are linearizable and lock-free.
5. We present benchmarks that compare performance of concurrent hash tries against other concurrent data structures. We interpret the results.

The rest of the paper is organized as follows. Section 2 describes sequential hash tries and several attempts to make their operations concurrent. It then presents case studies with concurrent hash trie operations. Section 3 presents the algorithm for concurrent hash trie operations and describes it in detail. Section 4 presents the outline of the correctness proof – a complete proof is given in the appendix. Section 5 contains experimental results and their interpretation. Section 6 presents related work and section 7 concludes.

2 Discussion

Hash array mapped tries (from now on hash tries) described previously by Bagwell [1] are trees that have 2 types of nodes – internal nodes and leaves. Leaves store key-value bindings. Internal nodes have a 2^W-way branching factor. In a straightforward implementation, each internal node is a 2^W-element array. Finding a key proceeds in the following manner. If the internal node is at the root, the initial W bits of the key hashcode are used as an index in the array. If the internal node is at the level l, then W bits of the hashcode starting from the position $W * l$ are used. This is repeated until a leaf or an empty entry is found. Insertion and removal are similar.

Such an implementation is space-inefficient – most entries in the internal nodes are never used. To ensure space efficiency, each internal node contains a bitmap of length 2^W. If a bit is set, then its corresponding array entry contains an element. The corresponding entry for a bit on position i in the bitmap bmp is calculated as $\#((i-1) \odot bmp)$, where $\#$ is the bitcount and $\odot$ is a logical AND operation. The W bits of the hashcode relevant at some level l are used to compute the index i as before. At all times an invariant is preserved that the bitmap bitcount is equal to the array length. Typically, W is 5 since that ensures that 32-bit integers can be used as bitmaps. An example hash trie is shown in Fig. 1A. The expected depth is logarithmic in the number of elements added – this drives the running time of operations.

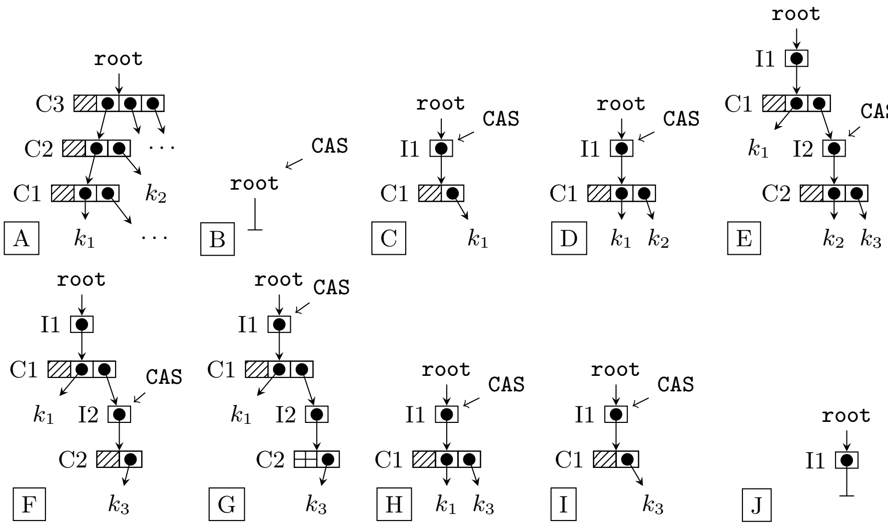


Fig. 1. Hash trie and Ctrie examples

We want to preserve the nice properties of hash tries – space-efficiency, cache-awareness and the expected depth of $O(\log_{2^W}(n))$, where n is the number of elements stored in the trie and 2^W is the bitmap length. We also want to make hash tries a concurrent data structure that can be accessed by multiple threads. In doing so, we avoid locks and rely solely on CAS instructions. Furthermore, we ensure that the new data structure has the lock-freedom property. We call the data structure a *Ctrie*. In the remainder of this chapter we give several examples.

Assume that we have a hash trie from Fig. 1A and that a thread T_1 decides to insert a new key below the node `C1`. One way to do this is to do a CAS on the bitmap in `C1` to set the bit that corresponds to the new entry in the array, and then CAS the entry in the array to point to the new key. This requires all the arrays to have additional empty entries, leading to inefficiencies. A possible solution is to keep a pointer to the array inside `C1` and do a CAS on that pointer with the updated copy of the array. The fundamental problem that still remains is that such an insertion does not happen atomically. It is possible that some other thread T_2 also tries to insert below `C1` after its bitmap is updated, but before the array pointer is updated. Lock-freedom is not ensured if T_2 were to wait for T_1 to complete.

Another solution is for T_1 to create an updated version of `C1` called `C1'` with the updated bitmap and the new key entry in the array, and then do a CAS in the entry within the `C2` array that points to `C1`. The change is then done atomically. However, this approach does not work. Assume that another thread T_2 decides to insert a key below the node `C2` at the time when T_1 is creating `C1'`. To do this, it has to read `C2` and create its updated copy `C2'`. Assume that after that, T_1 does the CAS in `C2`. The copy `C2'` will not reflect the changes by T_1. Once T_2 does a CAS in the `C3` array, the key inserted by T_1 is lost.

To solve this problem we define a new type of a node that we call an *indirection node*. This node remains present within the Ctrie even if nodes above and below it change. We now show an example of a sequence of Ctrie operations.

Every Ctrie is defined by the `root` reference (Fig. 1B). Initially, the `root` is set to a special value called `null`. In this state the Ctrie corresponds to an empty set, so all lookups fail to find a value for any given key and all remove operations fail to remove a binding.

Assume that a key k_1 has to be inserted. First, a new node `C1` of type `CNode` is created, so that it contains a single key `k1` according to hash trie invariants. After that, a new node `I1` of type `INode` is created. The node `I1` has a single field *main* (Fig. 2) that is initialized to `C1`. A CAS instruction is then performed at the `root` reference (Fig. 1B), with the expected value `null` and the new value `I1`. If a CAS is successful, the insertion is completed and the Ctrie is in a state shown in Fig. 1C. Otherwise, the insertion must be repeated.

Assume next that a key k_2 is inserted such that its hashcode prefix is different from that of k_1. By the hash trie invariants, k_2 should be next to k_1 in `C1`. The thread that does the insertion first creates an updated version of `C1` and then does a CAS at the `I1.main` (Fig. 1C) with the expected value of `C1` and the updated node as the new value. Again, if the CAS is not successful, the insertion process is repeated. The Ctrie is now in the state shown in Fig. 1D.

If some thread inserts a key k_3 with the same initial bits as k_2, the hash trie has to be extended with an additional level. The thread starts by creating a new node `C2` of type `CNode` containing both k_2 and k_3. It then creates a new node `I2` and sets `I2.main` to `C2`. Finally, it creates a new updated version of `C1` such that it points to the node `I2` instead of the key k_2 and does a CAS at `I1.main` (Fig. 1D). We obtain a Ctrie shown in Fig. 1E.

Assume now that a thread T_1 decides to remove k_2 from the Ctrie. It creates a new node `C2'` from `C2` that omits the key k_2. It then does a CAS on `I2.main` to set it to `C2'` (Fig. 1E). As before, if the CAS is not successful, the operation is restarted. Otherwise, k_2 will no longer be in the trie – concurrent operations will only see k_1 and k_3 in the trie, as shown in Fig. 1F. However, the key k_3 could be moved further to the root - instead of being below the node `C2`, it could be directly below the node `C1`. In general, we want to ensure that the path from the root to a key is as short as possible. If we do not do this, we may end up with a lot of wasted space and an increased depth of the Ctrie.

For this reason, after having removed a key, a thread will attempt to contract the trie as much as possible. The thread T_1 that removed the key has to check whether or not there are less than 2 keys remaining within `C2`. There is only a single key, so it can create a copy of `C1` such that the key k_3 appears in place of the node `I2` and then do a CAS at `I1.main` (Fig. 1F). However, this approach does not work. Assume there was another thread T_2 that decides to insert a new key below the node `I2` just before T_1 does the CAS at `I1.main`. The key inserted by T_2 is lost as soon as the CAS at `I1.main` occurs.

To solve this, we relax the invariants of the data structure. We introduce a new type of a node - a tomb node. A tomb node is simply a node that holds a single key. No thread may modify a node of type `INode` if it contains a tomb node. In our example, instead of directly modifying `I1`, thread T_1 must first create a tomb node that contains the key k_3. It then does a CAS at `I2.main` to set it to the tomb node. After having done this (Fig. 1G), T_1 may create a contracted version of `C1` and do a CAS at `I1.main`, at that point we end up with a trie of an optimal size (Fig. 1H). If some other thread T_2 attempts to modify `I2` after it has been *tombed*, then it must first do the same thing T_1 is attempting to do - move the key k_3 back below `C2`, and only then proceed with its original operation. We call an `INode` that points to a tomb node a *tomb-I-node*. We say that a tomb-I-node in the example above is *resurrected*.

If some thread decides to remove k_1, it proceeds as before. However, even though k_3 now remains the only key in `C1` (Fig. 1I), it does not get tombed. The reason for this is that we treat nodes directly below the root differently. If k_3 were next removed, the trie would end up in a state shown in Fig. 1J, with the `I1.main` set to `null`. We call this type of an `INode` a *null-I-node*.

```
                        MainNode: CNode | SNode
root: INode                                            structure SNode {
                        structure CNode {                k: KeyType
structure INode {         bmp: integer                   v: ValueType
  main: MainNode          array: Array[2^W]              tomb: boolean
}                       }                              }
```

Fig. 2. Types and data structures

3 Algorithm

We present the pseudocode of the algorithm in figures 3, 4 and 5. The pseudocode assumes C-like semantics of conditions in *if* statements – if the first condition in a conjunction fails, the second one is never evaluated. We use logical symbols for boolean expressions. The pseudocode also contains pattern matching constructs that are used to match a node against its type. All occurences of pattern matching can be trivially replaced with a sequence of *if-then-else* statements – we use pattern matching for conciseness. The colon (`:`) in the pattern matching cases should be understood as *has type*. The keyword `def` denotes a procedure definition. Reads and compare-and-set instructions written in capitals are atomic – they occur at

```
 1 def insert(k, v)
 2   r = READ(root)
 3   if r = null ∨ isNullInode(r) {
 4     scn = CNode(SNode(k, v, ⊥))
 5     nr = INode(scn)
 6     if !CAS(root, r, nr) insert(k, v)
 7   } else if ¬iinsert(r, k, v, 0, null)
 8     insert(k, v)
 9
10 def remove(k)
11   r = READ(root)
12   if r = null return NOTFOUND
13   else if isNullInode(r) {
14     CAS(root, r, null)
15     return remove(k)
16   } else {
17     res = iremove(r, k, 0, null)
18     if res ≠ RESTART return res
19     else remove(k)
20   }
21
22 def lookup(k)
23   r = READ(root)
24   if r = null return NOTFOUND
25   else if isNullInode(r) {
26     CAS(root, r, null)
27     return lookup(k)
28   } else {
29     res = ilookup(r, k, 0, null)
30     if res ≠ RESTART return res
31     else return lookup(k)
32   }
33
34 def ilookup(i, k, lev, parent)
35   READ(i.main) match {
36   case cn: CNode =>
37     flag, pos = flagpos(k.hc, lev, cn.bmp)
38     if cn.bmp ⊙ flag = 0 return NOTFOUND
39     cn.array(pos) match {
40     case sin: INode =>
41       return ilookup(sin, k, lev + W, i)
42     case sn: SNode ∧ ¬sn.tomb =>
43       if sn.k = k return sn.v
44       else return NOTFOUND
45     }
46   case (sn: SNode ∧ sn.tomb) ∨ null =>
47     if parent ≠ null clean(parent)
48     return RESTART
49   }
```

Fig. 3. Basic operations I

one point in time. This is a high level pseudocode and might not be optimal in all cases – the source code contains a more efficient implementation.[1]

Operations start by reading the `root` (lines 2, 11 and 23). If the `root` is `null` then the trie is empty, so neither removal nor lookup finds a key. If the `root` points to an `INode` that is set to `null` (as in Fig. 1J), then the root is set back to just `null` before repeating. In both the previous cases, an insertion will replace the `root` reference with a new `CNode` with the appropriate key.

If the `root` is neither `null` nor a null-I-node then the node below the root I-node is read (lines 35, 51 and 79), and we proceed casewise. If the node pointed at by the I-node is a `CNode`, an appropriate entry in its array must be found. The method `flagpos` computes the values `flag` and `pos` from the hashcode hc of the key, bitmap bmp of the cnode and the current level lev. The relevant `flag` in the bitmap is defined as $(hc >> (k \cdot lev)) \odot ((1 << k) - 1)$, where 2^k is the length of the bitmap. The position `pos` within the array is given by the expression $\#((flag - 1) \odot bmp)$, where $\#$ is the bitcount. The `flag` is used to check if the appropriate branch is in the `CNode` (lines 38, 54, 82). If it is not, lookups and removes end, since the desired key is not in the Ctrie. An insert creates an updated copy of the current `CNode` with the new key. If the branch is in the trie, `pos` is used as an index into the array. If an I-node is found, we repeat the operation recursively. If a key-value binding (an `SNode`) is found, then a lookup compares the keys and returns the binding if they are the same. An insert operation will either replace the old binding if the keys are the same, or otherwise extend the trie below the `CNode`. A remove compares the keys – if they are the same it replaces the `CNode` with its updated version without the key.

[1] See: `http://github.com/axel22/Ctries`

```
50 def iinsert(i, k, v, lev, parent)
51   READ(i.main) match {
52   case cn: CNode =>
53     flag, pos = flagpos(k.hc, lev, cn.bmp)
54     if cn.bmp ⊙ flag = 0 {
55       nsn = SNode(k, v, ⊥)
56       narr = cn.array.inserted(pos, nsn)
57       ncn = CNode(narr, bmp | flag)
58       return CAS(i.main, cn, ncn)
59     }
60     cn.array(pos) match {
61     case sin: INode =>
62       return iinsert(sin, k, v, lev + W, i)
63     case sn: SNode ∧ ¬sn.tomb =>
64       nsn = SNode(k, v, ⊥)
65       if sn.k = k {
66         ncn = cn.updated(pos, nsn)
67         return CAS(i.main, cn, ncn)
68       } else {
69         nin = INode(CNode(sn, nsn, lev + W))
70         ncn = cn.updated(pos, nin)
71         return CAS(i.main, cn, ncn)
72       }
73     }
74   case (sn: SNode ∧ sn.tomb) ∨ null =>
75     if parent ≠ null clean(parent)
76     return ⊥
77   }
```

```
78 def iremove(i, k, lev, parent)
79   READ(i.main) match {
80   case cn: CNode =>
81     flag, pos = flagpos(k.hc, lev, cn.bmp)
82     if cn.bmp ⊙ flag = 0 return NOTFOUND
83     res = cn.array(pos) match {
84     case sin: INode =>
85       return iremove(sin, k, lev + W, i)
86     case sn: SNode ∧ ¬sn.tomb =>
87       if sn.k = k {
88         narr = cn.array.removed(pos)
89         ncn = CNode(narr, bmp ^ flag)
90         if cn.array.length = 1 ncn = null
91         if CAS(i.main, cn, ncn) return sn.v
92         else return RESTART
93       } else return NOTFOUND
94     }
95     if res = NOTFOUND ∨ res = RESTART return res
96     if parent ne null ∧ tombCompress()
97       contractParent(parent, in, k.hc, lev - W)
98   case (sn: SNode ∧ sn.tomb) ∨ null =>
99     if parent ≠ null clean(parent)
100    return RESTART
101  }
```

Fig. 4. Basic operations II

After a key was removed, the trie has to be contracted. A remove operation first attempts to create a tomb from the current `CNode`. It first reads the node below the current I-node to check if it is still a `CNode`. It then calls `toWeakTombed` that creates a *weak tomb* from the given `CNode`. A weak tomb is defined as follows. If the number of nodes below the `CNode` that are not null-I-nodes is greater than 1, then it is the `CNode` itself – in this case we say that there is nothing to entomb. If the number of such nodes is 0, then the weak tomb is `null`. Otherwise, if the single branch below the `CNode` is a key-value binding or a tomb-I-node (also called a *singleton*), the weak tomb is the tomb node with that binding. If the single branch below is another `CNode`, a weak tomb is a copy of the current `CNode` with the null-I-nodes removed.

The procedure `tombCompress` continually tries to entomb the current `CNode` until it finds out that there is nothing to entomb or it succeeds. The CAS in line 132 corresponds to the one in Fig. 1F. If it succeeds and the weak tomb was either a `null` or a tomb node, it will return `true`, meaning that the parent node should be contracted. The contraction is done in `contractParent`, that checks if the I-node is still reachable from its parent and then contracts the `CNode` below the parent - it removes the null-I-node (line 148) or resurrects a tomb-I-node into an `SNode` (line 152). The latter corresponds to the CAS in Fig. 1G.

If any operation encounters a `null` or a tomb node, it attempts to fix the Ctrie before proceeding, since the Ctrie is in a *relaxed* state. A tomb node may have originated from a remove operation that will attempt to contract the tomb node at some time in the future. Rather than waiting for that remove operation

```
def toCompressed(cn)
  num = bit#(cn.bmp)
  if num = 1 ∧ isTombInode(cn.array(0))
    return cn.array(0).main
  ncn = cn.filtered(_.main ≠ null)
  rarr = ncn.array.map(resurrect(_))
  if bit#(ncn.bmp) > 0
    return CNode(rarr, ncn.bmp)
  else return null

def toWeakTombed(cn)
  farr = cn.array.filtered(_.main ≠ null)
  nbmp = cn.bmp.filtered(_.main ≠ null)
  if farr.length > 1 return cn
  if farr.length = 1
    if isSingleton(farr(0))
      return farr(0).tombed
    else CNode(farr, nbmp)
  return null

def clean(i)
  m = READ(i.main)
  if m ∈ CNode
    CAS(i.main, m, toCompressed(m))

def tombCompress(i)
  m = READ(i.main)
  if m ∉ CNode return ⊥
  mwt = toWeakTombed(m)
  if m = mwt return ⊥
  if CAS(i.main, m, mwt) mwt match {
    case null ∨ (sn: SNode ∧ sn.tomb) =>
      return ⊤
    case _ => return ⊥
  } else return tombCompress()

def contractParent(parent, i, hc, lev)
  m, pm = READ(i.main), READ(parent.main)
  pm match {
  case cn: CNode =>
    flag, pos = flagpos(k.hc, lev, cn.bmp)
    if bmp ⊙ flag = 0 return
    sub = cn.array(pos)
    if sub ≠ i return
    if m = null {
      ncn = cn.removed(pos)
      if !CAS(parent.main, cn, ncn)
        contractParent(parent, i, hc, lev)
    } else if isSingleton(m) {
      ncn = cn.updated(pos, m.untombed)
      if !CAS(parent.main, cn, ncn)
        contractParent(parent, i, hc, lev)
    }
  case _ => return
  }
```

Fig. 5. Compression operations

to do its work, the current operation should do the work of contracting the tomb itself, so it will invoke the `clean` operation on the parent I-node. The `clean` operation will attempt to exchange the `CNode` below the parent I-node with its compression. A `CNode` compression is defined as follows – if the `CNode` has a single tomb node directly beneath, then it is that tomb node. Otherwise, the compression is the copy of the `CNode` without the null-I-nodes (this is what the `filtered` call in the `toCompressed` procedure does) and with all the tomb-I-nodes resurrected to regular key nodes (this is what the `map` and `resurrect` calls do). Going back to our previous example, if in Fig. 1G some other thread were to attempt to write to `I2`, it would first do a `clean` operation on the parent `I1` of `I2` – it would contract the trie in the same way as the removal would have.

4 Correctness

As illustrated by the examples in the previous section, designing a correct lock-free algorithm is not straightforward. One of the reasons for this is that all possible interleavings of steps of different threads executing the operations have to be considered. For brevity, this section gives only the outline of the correctness proof. There are three main criteria for correctness. *Safety* means that the Ctrie corresponds to some abstract set of keys and that all operations change the corresponding abstract set of keys consistently. An operation is *linearizable* if any external observer can only observe the operation as if it took place instantaneously at some point between its invocation and completion [9] [11]. *Lock-freedom* means that if some number of threads execute operations concurrently, then after a finite number of steps some operation must complete [9].

We assume that the Ctrie has a branching factor 2^W. Each node in the Ctrie is identified by its type, level in the Ctrie l and the hashcode prefix p. The hashcode prefix is the sequence of branch indices that have to be followed from the root in order to reach the node. For a C-node $cn_{l,p}$ and a key k with the hashcode $h = r_0 \cdot r_1 \cdots r_n$, we denote $cn.sub(k)$ as the branch with the index r_l or *null* if such a branch does not exist. We define the following invariants:

INV1 For every I-node $in_{l,p}$, $in_{l,p}.main$ is a C-node $cn_{l,p}$, a tombed S-node $sn\dagger$ or *null*.
INV2 For every C-node the length of the array is equal to the bitcount in the bitmap.
INV3 If a flag i in the bitmap of $cn_{l,p}$ is set, then corresponding array entry contains an I-node $in_{l+W,p \cdot r}$ or an S-node.
INV4 If an entry in the array in $cn_{l,p}$ contains an S-node sn, then p is the prefix of the hashcode $sn.k$.
INV5 If an I-node $in_{l,p}$ contains an S-node sn, then p is the prefix of the hashcode $sn.k$.

We say that the Ctrie is *valid* if and only if the invariants hold. The relation $hasKey(node, x)$ holds if and only if the key x is within an S-node reachable from *node*. A valid Ctrie is *consistent* with an abstract set $\mathbb{A}$ if and only if $\forall k \in \mathbb{A}$ the relation $hasKey(root, k)$ holds and $\forall k \notin \mathbb{A}$ it does not. A Ctrie lookup is *consistent* with the abstract set semantics if and only if it finds the keys in the abstract set and does not find other keys. A Ctrie insertion or removal is *consistent* with the abstract set semantics if and only if it produces a new Ctrie consistent with a new abstract set with or without the given key, respectively.

Lemma 1. *If an I-node in is either a null-I-node or a tomb-I-node at some time* t_0 *then* $\forall t > t_0$ *in.main is never written to. We refer to such I-nodes as nonlive.*

Lemma 2. *C-nodes and S-nodes are immutable – once created, they do not change the value of their fields.*

Lemma 3. *Invariants INV1-3 always hold.*

Lemma 4. *If a CAS instruction makes an I-node in unreachable from its parent at some time* t_0*, then in is nonlive at* t_0*.*

Lemma 5. *Reading a cn such that* $cn.sub(k) = sn$ *and* $k = sn.k$ *at some time* t_0 *means that* $hasKey(root, k)$ *holds at* t_0*.*

For a given Ctrie, we say that the longest path for a hashcode $h = r_0 \cdot r_1 \cdots r_n$, $length(r_i) = W$, is the path from the root to a leaf such that at each C-node $cn_{i,p}$ the branch with the index r_i is taken.

Lemma 6. *Assume that the Ctrie is an valid state. Then every longest path ends with an S-node, C-node or null.*

Lemma 7. *Assume that a C-node cn is read from $in_{l,p}$.main at some time t_0 while searching for a key k. If cn.sub(k) = null then hasKey(root, k) is not in the Ctrie at t_0.*

Lemma 8. *Assume that the algorithm is searching for a key k and that an S-node sn is read from cn.array(i) at some time t_0 such that $sn.k \neq k$. Then the relation hasKey(root, k) does not hold at t_0.*

Lemma 9. *1. Assume that one of the CAS in lines 58 and 71 succeeds at time t_1 after in.main was read in line 51 at time t_0. Then $\forall t, t_0 \leq t < t_1$, relation hasKey(root, k) does not hold.*

2. Assume that the CAS in lines 67 succeeds at time t_1 after in.main was read in line 51 at time t_0. Then $\forall t, t_0 \leq t < t_1$, relation hasKey(root, k) holds.

3. Assume that the CAS in line 91 succeeds at time t_1 after in.main was read in line 79 at time t_0. Then $\forall t, t_0 \leq t < t_1$, relation hasKey(root, k) holds.

Lemma 10. *Assume that the Ctrie is valid and consistent with some abstract set $\mathbb{A}$ $\forall t, t_1 - \delta < t < t_1$. CAS instructions from lemma 9 induce a change into a valid state which is consistent with the abstract set semantics.*

Lemma 11. *Assume that the Ctrie is valid and consistent with some abstract set $\mathbb{A}$ $\forall t, t_1 - \delta < t < t_1$. If one of the operations clean, tombCompress or contractParent succeeds with a CAS at t_1, the Ctrie will remain valid and consistent with the abstract set $\mathbb{A}$ at t_1.*

Corollary 1. *Invariants INV4,5 always hold due to lemmas 10 and 11.*

Theorem 1 (Safety). *At all times t, a Ctrie is in a valid state $\mathbb{S}$, consistent with some abstract set $\mathbb{A}$. All Ctrie operations are consistent with the semantics of the abstract set $\mathbb{A}$.*

Theorem 2 (Linearizability). *Ctrie operations are linearizable.*

Lemma 12. *If a CAS that does not cause a consistency change in one of the lines 58, 67, 71, 125, 132, 148 or 152 fails at some time t_1, then there has been a state (configuration) change since the time t_0 when a respective read in one of the lines 51, 51, 51, 123, 128, 139 or 139 occured.*

Lemma 13. *In each operation there is a finite number of execution steps between consecutive CAS instructions.*

Corollary 2. *There is a finite number of execution steps between two state changes. This does not imply that there is a finite number of execution steps between two operations. A state change is not necessarily a consistency change.*

We define the **total path length** d as the sum of the lengths of all the paths from the root to some leaf. Assume the Ctrie is in a valid state. Let n be the number of reachable null-I-nodes in this state, t the number of reachable tomb-I-nodes, l the number of live I-nodes, r the number of single tips of any length and d the total path length. We denote the state of the Ctrie as $\mathbb{S}_{n,t,l,r,d}$. We call the state $\mathbb{S}_{0,0,l,r,d}$ the **clean** state.

Lemma 14. *Observe all CAS instructions which never cause a consistency change and assume they are successful. Assuming there was no state change since reading in prior to calling clean, the CAS in line 125 changes the state of the Ctrie from the state* $\mathbb{S}_{n,t,l,r,d}$ *to either* $\mathbb{S}_{n+j,t,l,r-1,d-1}$ *where* $r > 0$, $j \in \{0, 1\}$ *and* $d \geq 1$, *or to* $\mathbb{S}_{n-k,t-j,l,r,d' \leq d}$ *where* $k \geq 0$, $j \geq 0$, $k + j > 0$, $n \geq k$ *and* $t \geq j$. *Furthermore, the CAS in line 14 changes the state of the Ctrie from* $\mathbb{S}_{1,0,0,0,1}$ *to* $\mathbb{S}_{0,0,0,0,0}$. *The CAS in line 26 changes the state from* $\mathbb{S}_{1,0,0,0,1}$ *to* $\mathbb{S}_{0,0,0,0,0}$. *The CAS in line 132 changes the state from* $\mathbb{S}_{n,t,l,r,d}$ *to either* $\mathbb{S}_{n+j,t,l,r-1,d-j}$ *where* $r > 0$, $j \in \{0, 1\}$ *and* $d \geq j$, *or to* $\mathbb{S}_{n-k,t,l,r,d' \leq d}$ *where* $k > 0$ *and* $n \geq k$. *The CAS in line 148 changes the state from* $\mathbb{S}_{n,t,l,r,d}$ *to* $\mathbb{S}_{n-1,t,l,r+j,d-1}$ *where* $n > 0$ *and* $j \geq 0$. *The CAS in line 152 changes the state from* $\mathbb{S}_{n,t,l,r}$ *to* $\mathbb{S}_{n,t-1,l,r+j,d-1}$ *where* $j \geq 0$.

Lemma 15. *If the Ctrie is in a clean state and n threads are executing operations on it, then some thread will execute a successful CAS resulting in a consistency change after a finite number of execution steps.*

Theorem 3 (Lock-freedom). *Ctrie operations are lock-free.*

5 Experiments

We show benchmark results in Fig. 6. All the measurements were performed on a quad-core 2.67 GHz i7 processor with hyperthreading. We followed established performance measurement methodologies [2]. We compare the performance of Ctries against that of `ConcurrentHashMap` and `ConcurrentSkipListMap` [3] [4] data structures from the Java standard library.

In the first experiment, we insert a total of N elements into the data structures. The insertion is divided equally among P threads, where P ranges from 1 to 8. The results are shown in Fig. 6A-D. Ctries outperform concurrent skip lists for $P = 1$ (Fig. 6A). We argue that this is due to a fewer number of indirections in the Ctrie data structure. A concurrent skip list roughly corresponds to a balanced binary tree which has a branching factor 2. Ctries normally have a branching factor 32, thus having a much lower depth. A lower depth means less indirections and consequently fewer cache misses when searching the Ctrie.

We can also see that the Ctrie sometimes outperforms a concurrent hash table for $P = 1$. The reason is that the hash table has a fixed size and is resized once the load factor is reached – roughly speaking, a new table has to be allocated and all the elements from the previous hash table have to be copied into the new hash table. To do this, parts of the hash table have to be locked – other threads adding new elements into the table have to wait until the resize completes. This problem is much more apparent in Fig. 6B where $P = 8$. Fig. 6C,D show how the insertion scales for the number of elements $N = 200k$ and $N = 1M$, respectively. Due to the use of hyperthreading on the i7, we do not get significant speedups when $P > 4$ for these data structures. We next measure the performance for the remove operation (Fig. 6E-H). Each data structure starts with N elements and then emptied concurrently by P threads. The keys being removed are divided equally among the threads. For $P = 1$ Ctries are clearly outperformed by both other data structures. However, it should be noted that concurrent hash table does not shrink once the number of keys becomes much lower than the table size. This is space-inefficient – a hash table contains many elements at some point during the runtime of the application will continue to use the memory it does not need until the application ends. The slower Ctrie performance seen in Fig. 6E for $P = 1$ is attributed to the additional work the remove operation does to keep the Ctrie compact. However, Fig. 6F shows that the Ctrie remove operation scales well for $P = 8$, as it outperforms both skip list and hash table removals. This is also apparent in Fig. 6G,H. In the next experiment, we populate all the data structures with N elements and then do a lookup for every element once. The set of elements to be looked up is divided equally among P threads. From Fig. 6I-L it is apparent that concurrent hash tables have a much more efficient lookups than other data structures. This is not surprising since they are a flat data structure – a lookup typically consists of a single read in the table, possibly followed by traversing the collision chain within the bucket. Although a Ctrie lookup outperforms a concurrent skip list when $P = 8$, it still has to traverse more indirections than a hash table. Finally, we do a series of benchmarks with both lookups and insertions to determine the percentage of lookups for which the concurrent hash table performance equals that of concurrent tries. Our test inserts new elements into the data structures using P threads. A total of N elements are inserted. After each insert, a lookup for a random element is performed r times, where r is the ratio of lookups per insertion. Concurrent skip lists scaled well in these tests but had low absolute performance, so they are excluded from the graphs for clarity. When using $P = 2$ threads, the ratio where the running time is equal for both concurrent hash tables and concurrent tries is $r = 2$. When using $P = 4$ threads this ratio is $r = 5$ and for $P = 8$ the ratio is $r = 9$. As the number of threads increases, more opportunity for parallelism is lost during the resize phase in concurrent hash tables, hence the ratio increases. This is shown in Fig. 6M-O. In the last benchmark (Fig. 6P) we preallocate the array for the concurrent hash table to avoid resize phases – in this case the hash table outperforms the concurrent trie. The performance gap decreases as the number of threads approaches $P = 8$. The downside is that a large amount of memory has to be used for the hash table and the size needs to be known in advance.

6 Related Work

Concurrent programming techniques and important results in the area are covered by Shavit and Herlihy [9]. An overview of concurrent data structures is given by Moir and Shavit [10]. There is a body of research available focusing on concurrent lists, queues and concurrent priority queues [5] [10]. While linked lists are inefficient as sets or maps because they do not scale well, the latter two do not support the basic operations on sets and maps, so we exclude these from the further discussion and focus on more suitable data structures.

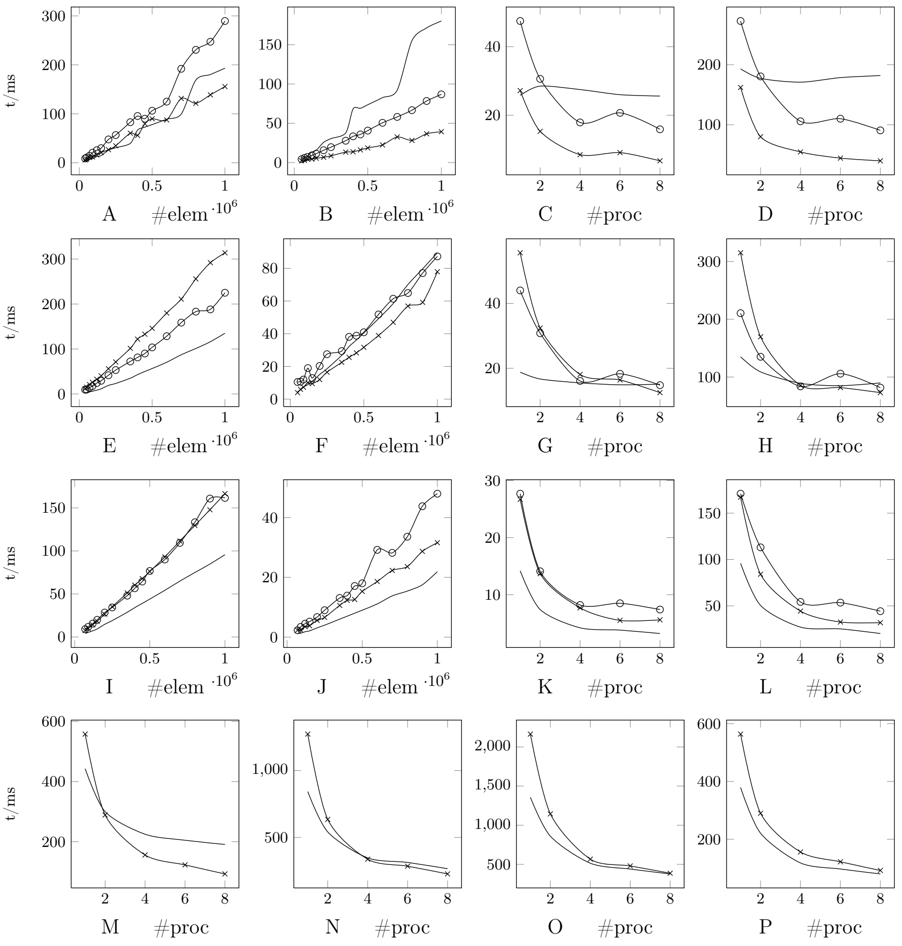


Fig. 6. Quad-core i7 microbenchmarks – *ConcurrentHashMap*(−), *ConcurrentSkipList*(∘), *Ctrie*(×): A) *insert*, P=1; B) *insert*, P=8; C) *insert*, N=200k; D) *insert*, N=1M; E) *remove*, P=1; F) *remove*, P=8; G) *remove*, N=200k, H) *remove*, N=1M; I) *lookup*, P=1; J) *lookup*, P=8; K) *lookup*, N=200k; L) *lookup*, N=1M; M) *insert/lookup*, ratio=1/2, N=1M; N) *insert/lookup*, ratio=1/5, N=1M; O) *insert/lookup*, ratio=1/9, N=1M; P) *insert/lookup* with preallocated tables, ratio=1/2, N=1M

Hash tables are typically resizeable arrays of buckets. Each bucket holds some number of elements which is expected to be constant. The constant number of elements per bucket necessitates resizing the data structure. Sequential hash tables amortize the cost of resizing the table over other operations [14]. While the individual concurrent hash table operations such as insertion or removal can be performed in a lock-free manner as shown by Maged [4], resizing is typically implemented with a global lock. Although the cost of resize is amortized against operations by one thread, this approach does not guarantee horizontal scalability. Lea developed an extensible hash algorithm which allows concurrent searches during the resizing phase, but not concurrent insertions and removals [3]. Shalev and Shavit propose split-ordered lists which keep a table of hints into a linked list in a way that does not require rearranging the elements of the linked list when resizing [15]. This approach is quite innovative, but it is unclear how to shrink the hint table if most of the keys are removed, while preserving lock-freedom.

Skip lists are a data structure which stores elements in a linked list. There are multiple levels of linked lists which allow efficient insertions, removals and lookups. Skip lists were originally invented by Pugh [16]. Pugh proposed concurrent skip lists which achieve synchronization through the use of locks [17]. Concurrent non-blocking skip lists were later implemented by Lev, Herlihy, Luchangco and Shavit [18] and Lea [3]. Concurrent binary search trees were proposed by Kung and Lehman [19] – their implementation uses a constant number of locks at a time which exclude other insertion and removal operations, while lookups can proceed concurrently. Bronson et al. presented a scalable concurrent implementation of an AVL tree based on transactional memory mechanisms which require a fixed number of locks to perform deletions [20]. Recently, the first non-blocking implementation of a binary search tree was proposed [21].

Tries were originally proposed by Brandais [6] and Fredkin [7]. Trie hashing was applied to accessing files stored on the disk by Litwin [12]. Litwin, Sagiv and Vidyasankar implemented trie hashing in a concurrent setting [13], however, they did so by using mutual exclusion locks. Hash array mapped trees, or hash tries, are tries for shared-memory proposed by Bagwell [1]. To our knowledge, there is no nonblocking concurrent implementation of hash tries prior our work.

7 Conclusion

We described a lock-free concurrent implementation of the hash trie data structure. Our implementation supports insertion, remove and lookup operations. It is space-efficient in the sense that it keeps a minimal amount of information in the internal nodes. It is compact in the sense that after all removal operations complete, all paths from the root to a leaf containing a key are as short as possible. Operations are worst-case logarithmic with a low constant factor ($O(\log_{32} n)$). Its performance is comparable to that of the similar concurrent data structures. The data structure grows dynamically – it uses no locks and there is no resizing phase. We proved that it is linearizable and lock-free.

In the future we plan to extend the algorithm with operations like *move key*, which reassigns a value from one key to another atomically. One research direction is supporting efficient aggregation operations on the keys stored in the Ctrie. One example of such an operation is the size of the Ctrie. Finally, we plan to develop an efficient lock-free snapshot operation for the concurrent trie which allows traversal of all the keys present in the data structure at the time at which the snapshot was created. One possible approach to doing so is to, roughly speaking, keep a partial history in the indirection nodes.

References

1. Bagwell, P.: Ideal Hash Trees (2002)
2. Georges, A., Buytaert, D., Eeckhout, L.: Statistically Rigorous Java Performance Evaluation. In: OOPSLA (2007)
3. Doug Lea's Home Page, `http://gee.cs.oswego.edu/`
4. Michael, M.M.: High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In: SPAA (2002)
5. Harris, T.L.: A Pragmatic Implementation of Non-Blocking Linked-Lists. In: IEEE Symposium on Distributed Computing (2001)
6. Brandais, R.: File searching using variable length keys. In: Proceedings of Western Joint Computer Conference (1959)
7. Fredkin, E.: Trie memory. Communications of the ACM (1960)
8. Silverstein, A.: Judy IV Shop Manual (2002)
9. Shavit, N., Herlihy, M.: The Art of Multiprocessor Programming. Morgan Kaufmann (2008)
10. Moir, M., Shavit, N.: Concurrent data structures. In: Handbook of Data Structures and Applications. Chapman and Hall (2004)
11. Herlihy, M., Wing, J.: Linearizability: A Correctness Condition for Concurrent Objects. ACM Transactions on Programming Languages and Systems (1990)
12. Litwin, W.: Trie Hashing. ACM (1981)
13. Litwin, W., Sagiv, Y., Vidyasankar, K.: Concurrency and Trie Hashing. ACM (1981)
14. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. The MIT Press (2001)
15. Shalev, O., Shavit, N.: Split-Ordered Lists: Lock-Free Extensible Hash Tables. Journal of the ACM 53(3) (2006)
16. Pugh, W.: Skip Lists: A Probabilistic Alternative to Balanced Trees. Communications ACM 33 (1990)
17. Pugh, W.: Concurrent Maintenance of Skip Lists (1990)
18. Herlihy, M., Lev, Y., Luchangco, V., Shavit, N.: A Provably Correct Scalable Concurrent Skip List. In: OPODIS (2006)
19. Kung, H., Lehman, P.: Concurrent manipulation of binary search trees. ACM (1980)
20. Bronson, N.G., Casper, J., Chafi, H., Olukotun, K.: A Practical Concurrent Binary Search Tree. ACM (2009)
21. Ellen, F., Fatourou, P., Ruppert, E., van Breugel, F.: Non-blocking binary search trees. In: PODC (2010)

Fine-Grained Treatment to Synchronizations in GPU-to-CPU Translation

Ziyu Guo and Xipeng Shen

College of William and Mary, Williamsburg VA 23187, USA
{guoziyu,xshen}@cs.wm.edu

Abstract. GPU-to-CPU translation may extend Graphics Processing Units (GPU) programs executions to multi-/many-core CPUs, and hence enable cross-device task migration and promote whole-system synergy. This paper describes some of our findings in treatment to GPU synchronizations during the translation process. We show that careful dependence analysis may allow a fine-grained treatment to synchronizations and reveal redundant computation at the instruction-instance level. Based on thread-level dependence graphs, we present a method to enable such fine-grained treatment automatically. Experiments demonstrate that compared to existing translations, the new approach can yield speedup of a factor of integers.

1 Introduction

For their advantages on computing power, cost, and energy efficiency, Graphic Processing Units (GPU) have become a type of mainstream co-processors in modern computing systems. Recent years have seen a rapid adoption of GPU-specific programming models, such as NVIDIA CUDA.

GPU-to-CPU translation aims at compiling code written in GPU programming models to (multi-core) CPU code. It has drawn some recent research interest from both academy and industry [3,8,10,12,14,16], for three reasons. First, it extends the range of applicable architecture and hence the impact of GPU programming models. An application developed in CUDA, for instance, can be automatically converted to a form suitable for multicore CPU. Such a capability is becoming increasingly important, given that the number of applications written in GPU programming models has increased continuously. Second, the translation enables smooth CPU-GPU collaborations. Given the trends towards heterogeneous systems, an essential requirement for maximizing computing efficiency is the synergistic cooperation among various types of processors. Automatic GPU-to-CPU translation facilitates seamless migration of jobs among GPU and CPU, hence helping promote a whole system synergy. Finally, a viable and widely accepted methodology for programming heterogeneous Chip Multiprocessors (CMP) systems is yet to be developed. As a programming model that exposes multi-level parallelism of an application to an extreme extent, CUDA-like GPU programming models are potential contenders if translation from them to a conventional commodity microprocessor works well.

Recent studies have produced some GPU-to-CPU translation systems, at both the source code level (e.g., MCUDA [9,10]) and below (e.g., Ocelot [8]). However, none of them has systematically explored the different implications of synchronizations on GPU and CPU.

On GPU, it is typical that some explicit device-specific intrinsics are introduced to enable synchronizations among threads. These intrinsics are used widely for being essential for the implementation of various parallel operations on GPU. In a CUDA application, for example, the GPU threads are organized in a number of *thread blocks*, and each thread block contains a number of *thread warps*. An intrinsic, __syncthreads(), serves as a thread-block–level barrier, ensuring that none of the threads in a block passes over the synchronization point before all threads in the block reaches that point.

In many GPU applications, the synchronization intrinsics are used even though a synchronization between just a subset of threads in a block is necessary. By doing that, the programmer introduces unnecessarily strong constraints, but avoids thread-specific checks and obtains programming easiness. It causes almost no issue to the performance of GPU applications because of the low overhead and high parallelism of hardware. However, a literal translation of such __syncthreads() calls to CPU, as existing GPU-to-CPU translation systems all do, often leads to considerable inefficiency.

The problem becomes even more serious when implicit synchronizations are taken into consideration. Due to the hardware implementation of GPU, synchronizations are sometimes realized in an implicit manner. In CUDA, every thread warp (32 threads) proceeds in lockstep. In another word, none of the threads can proceed to the next instruction until all threads in the warp have finished the current instruction. This default SIMD execution model is equivalent to that there is an implicit warp-level barrier after every instruction. Due to the prevalence of such implicit synchronizations, a literal translation of GPU synchronizations to CPU would cause serious efficiency issues. Existing GPU-to-CPU translation systems typically ignore such implicit synchronizations during the translation, the consequence of which is even more serious than the efficiency issue: The produced CPU code may be erroneous for violating some data dependences originally maintained by the implicit synchronizations (illustrated in Section 3).

In a recent study [12], we analyzed the correctness issue caused by the neglection of implicit synchronizations, proposed a dependence theory to identify critical implicit synchronizations (i.e., those that should not be ignored), and developed a state-level code generation algorithm to fix the issue.

In this paper, we investigate the potential of fine-grained treatment to the synchronizations. Unlike the prior study, this exploration deals with both implicit and explicit synchronizations. More importantly, it analyzes dependences among threads at the level of the dynamic instances of GPU statements. By lowering the granularity from statement to statement instances, it achieves a more detailed understanding of inter-thread data dependences. The understanding then leads to more efficient treatments to GPU synchronizations and the revelation of fine-grained redundant computations.

In the following parts of this paper, we first discuss the origin, forms, and performance implications of GPU synchronization intrinsics, both the explicit (Section 2) and implicit (Section 3). We then present the use of thread-level dependence graphs (TLDG) for representing fine-grained data and control dependences among dynamic instances of GPU kernel instructions. We report two uses of the fine-grained analysis. The first is to generate CPU code with unnecessary synchronization constraints relaxed. The second is to prune instruction-instance–level redundant computations (Section 4). We evaluate the effectiveness of the techniques on three GPU programs. By comparing with the codes produced by prior techniques, we demonstrate that the fine-grained treatment to synchronizations has some clear performance benefits (Section 5).

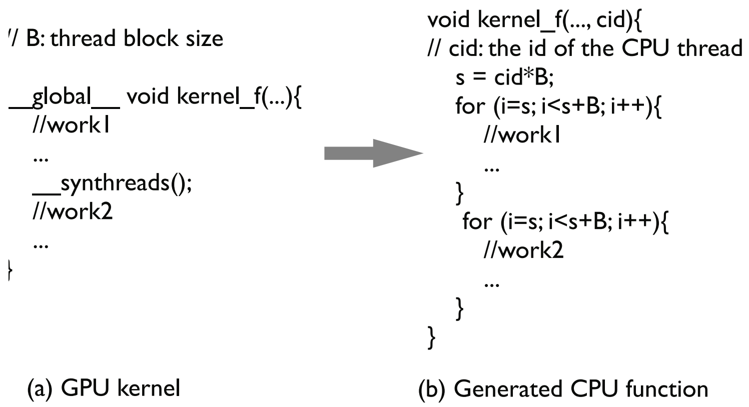
2 Impact of Explicit Synchronizations

One of the major device-specific features of the CUDA programming model is the intra-block synchronization. An invocation to the "__syncthreads()" will stall any run-ahead threads in the block until all threads have reached that point. As an essential and efficient barrier primitive, "__syncthreads()" exists prevalently in CUDA programs.

In existing GPU-to-CPU translation, the common approach to transforming GPU __syncthreads() into equivalent CPU code is to imitate the strict intra-block barrier via loop splitting. Figure 1 illustrates the translation scheme implemented in MCUDA [10], a typical GPU-to-CPU translator. For the purpose of explanation, the figure shows only the produced sequential code corresponding to the execution of the tasks done by one thread block. The GPU kernel function, orginially executed by every thread in the block, turns into two loops, which each has B iterations corresponding to the tasks executed by B GPU threads in the thread block. The two loops are for *work1* and *work2* respectively. Putting them into two separate loops ensures that the order constraints imposed by __syncthreads() are satisfied.

The loop splitting approach is subject to several drawbacks. First, it introduces extra loop overhead.

Second, it imposes strong constraints on the scheduling of instructions. In principle, the instructions in the second loop cannot be executed before the first loop finishes (unless the compiler finds that the two loops can be fused together). Although that constraint is the same as what the GPU synchronization intrinsics implies, it is often unnecessarily strong. The efficient synchronization and uniform SIMD execution model on GPU makes it attempting for programmers to skip fine-grained data dependence analysis during the coding of GPU kernels, and insert __syncthreads() wherever it might be needed. Often, synchronizations are only needed between a subset of threads. But this strategy is fine for GPU programming because the synchronization intrinsic is lightweight and usually there are no better alternatives—inserting conditional statements often



```
'/ B: thread block size

__global__ void kernel_f(...){
   //work1
   ...
   __synthreads();
   //work2
   ...
}
```

(a) GPU kernel

```
void kernel_f(..., cid){
// cid: the id of the CPU thread
   s = cid*B;
   for (i=s; i<s+B; i++){
      //work1
      ...
   }
    for (i=s; i<s+B; i++){
      //work2
      ...
   }
}
```

(b) Generated CPU function

Fig. 1. Illustration of MCUDA compilation. For illustration purpose, it shows the CPU code that corresponds to the execution of only one GPU thread block.

introduces even higher overhead than synchronizations. But on CPU, the constraints may hurt instruction scheduling and hence computing efficiency substantially.

The severity of the two issues depend on the density of the invocation of synchronization intrinsics. Unfortunately, due to the simplicity brought by invocations of the intrinsics, the density can be quite high in real GPU applications. In one of our benchmarks, CG_CUDA, for instance, 23 __syncthreads() invocations appear in a kernel with only 170 lines of code.

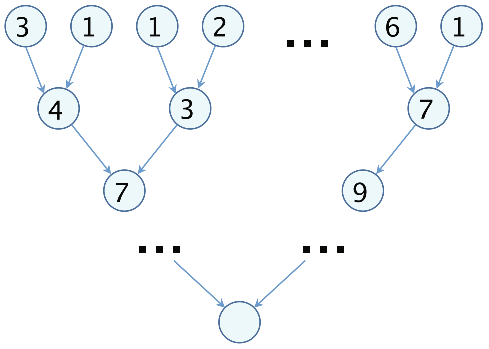
3 Impact of Implicit Synchronizations

GPU threads in a warp proceed in locksteps, which is equivalent to having an warp-level synchronization after every instruction. We call such order constraints implicit synchronizations.

Most previous GPU-to-CPU translations give no considerations to implicit synchronizations. Consequently, there is no guarantee that the generated code will retain the dependences in the original program; the produced code can be hence incorrect.

The parallel reduction program in Figure 2 illustrates the correctness pitfall. On the left is an illustration of the standard parallel reduction algorithm. It proceeds level by level; data dependences exist between every two levels. The bottom six lines of code in the kernel shown on the right of the figure correspond to the six bottom levels of the parallel reduction algorithm. They contain no explicit invocation of synchronization intrinsics, but the data dependences among the levels are well preserved, thanks to the implicit synchronizations among GPU instructions[1]. A translation of the kernel by MCUDA will violate the inter-level data dependences because of the negliction of the implicit synchronizations.

[1] We use source code rather than assembly instructions for illustration purpose.



(a) Algorithm

```
// s[]: contains input data
for (i=blockSize/2; i>32;
i>>=1){
        if (tid < 1)        s[tid]
+= s[tid+1];
        __syncthreads();
}
if (tid<32){
        s[tid] += s[tid+32];
        s[tid] += s[tid+16];
        s[tid] += s[tid+8];
        s[tid] += s[tid+4];
        s[tid] += s[tid+2];
        s[tid] += s[tid+1];
}
```

(b) Kernel function

Fig. 2. Algorithm and excerpted code of CUDA SDK reduction (kernel 5)

Such an exploitation of implicit synchronizations saves invocations of explicit synchronization intrinsics, and has served as a trick adopted by many GPU applications for achieving high performance. It is important to find an approach to handling them correctly and efficiently.

4 Instance-Level Dependence Analysis and Code Generation

We propose a fine-grained dependence analysis and code generation approach to address the limitations of the prior treatments to both explicit and implicit synchronizations in GPU-to-CPU translations. The approach is based on thread-level dependence graphs (TLDGs), a kind of representation of the dynamic instances of the instructions in a GPU kernel with intra- and inter-thread dependences captured.

In this section, we first introduce the concept of TLDG, and then describe its usage for enabling efficient and correct code generation in GPU-to-CPU translations.

Without loss of generality, we assume that the target code region for our following analysis meets the following two conditions: (1) It contains no loops; (2) the execution patterns of all thread blocks on that region are identical or the region is executed by only one thread block. These assumptions help focus our discussions on the handling of the synchronization issues; the elided complexities can be handled by existing compiler techniques without major adjustments to our analysis framework.

4.1 TLDG

A TLDG is a directed graph constructed based on the data and control dependences in the GPU code, with awareness of the semantics of the warp/block logical hierarchy and synchronizations. It captures the important dependences that will appear in the execution of a GPU thread block. Depending on the number of threads a TLDG models, a TLDG can be for a thread block or a thread warp. The former is useful for dealing with explicit synchronizations, while the latter is for implicit synchronizations.

The generation of the node set of TLDG focuses on only those statements that access data (e.g. arrays) shared by different threads in the warp or block. The first step is to break the statements into load/store references, as illustrated in Figure 3 (b). This step yields a set of data reference units (DRU), each containing exactly one reference to shared data. DRUs are the basic scheduling units in the follow-up optimizations.

The second step is to build up a set of static nodes, as illustrated by the nodes in Figure 3 (c). Each DRU maps to one static node. Each node is marked by the array reference in its corresponding DRU. An instruction that accesses no shared data are attached to a node whose corresponding DRU follows that instruction.

The third step creates the nodes in the TLDG by duplicating the entire set of static nodes N times (N is the number of threads to model). Each node corresponds to the dynamic instance of a static node executed by a GPU thread.

The final step connects the nodes via directed edges, as shown in Figure 3 (d). Each edge $n_1 \rightarrow n_2$ represents one of the following two possibilities:

- There is a control dependence from n_1 to n_2, when both nodes come from the same thread, or
- There is a data dependence from n_1 to n_2 coming from either the same or different threads, where the type of data dependence could be either true, anti- or output dependence.

Dependence analysis is done between every pair of DRU. Due to the regular data-level parallelism of many GPU kernels, compiler-based static analysis is often sufficient.

Note that nodes constructed from the same DRU are executed simultaneously on GPU. And because there are no loops in the code, dependence edges can not point from a later DRU to an earlier DRU. Thus if we organize the nodes in a matrix, with the thread id increasing along the horizontal direction and the time stamp of each DRU execution increasing along the vertical direction, then no edges can point upwards, and no cycles exist in a TLDG.

The edges in an TLDG essentially compose the set of order constraints for the execution of the tasks in the GPU kernel. As long as a GPU-to-CPU translation of the kernel observes these constraints, the produced CPU code meets the order requirement. Next, we describe how we select a good instruction order among many legal schedules.

```
if (blockSize >=  8)  sdata[tid] += sdata[tid +  4];
if (blockSize >=  4)  sdata[tid] += sdata[tid +  2];
if (blockSize >=  2)  sdata[tid] += sdata[tid +  1];
```

(a)

```
//node 0
if (blockSize >=  8)  {tempBuf[tid] =  sdata[tid+4];}
//node 1
if (blockSize >=  8)  {sdata[tid] =+  tempBuf[tid];}
//node 2
if (blockSize >=  4)  {tempBuf[tid] =  sdata[tid+2];}
//node 3
if (blockSize >=  4)  {sdata[tid] +=  tempBuf[tid];}
//node 4
if (blockSize >=  2)  {tempBuf[tid] =  sdata[tid+1];}
//node 5
if (blockSize >= 2)  {sdata[tid] +=  tempBuf[tid];}
```

(b)

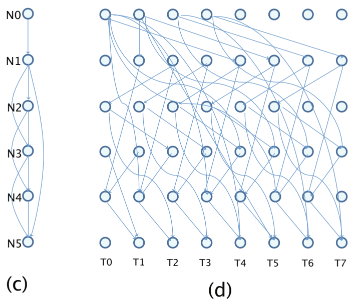


Fig. 3. (a). The original statements in CUDA SDK source code. (b). Statements broken into references, each forming a DRU. (c). The static nodes and dependences. (d) The intra-thread and inter-thread edges of the TLDG.

4.2 Code Generation

To generate a piece of CPU code from the TLDG, we need to introduce additional ordering between the DRUs without changing any source-sink relationships of the original GPU code. As the graph is acyclic, a simple breadth-first traversal of the graphs will yield a correct sequence. The rest part of the problem is comparing the quality of all the legal sequences and picking an optimal one. The produced code will be a sequence of all DRU instances, and no loops are needed to express the operations of a thread block.

Our code generation algorithm is a round-based scheduling algorithm as shown in Figure 4. The key idea is to partition the nodes into different groups and impose strict order among groups while maintaining full concurrency within each group. The DRUs in each group forms one round. In each round, the algorithm puts all nodes in the current TLDG that have no incoming edges into the round group (roundQueue in Figure 4), and then removes them along with their outgoing edges from the TLDG before proceeding to the next round. The process continues until the graph is empty.

A simple output of the instructions contained in each round in the round order will produce a correct execution sequence of DRUs. It is easy to see that the produced code maintains the source and sink relationships of all the dependences in the original TLDG. The successful detection and preservation of instance-level dependences effectively eliminate the need for a whole block

```
while G not empty:
    for each node N :
        if N.inDeg == 0
                    roundQueue.push(N);
           for each edge E outgoing from N:
            delete E from G
           delete N from G
    roundQueue.sort(N)
    outputCode.append(roundQueue.codeGeneration())
```

Fig. 4. Pseudo code for round-based code generation

synchronization. Such relaxation introduces additional freedom in the optimization space for GPU-to-CPU code generation. Figure 5 (a) shows the content of a piece of generated code.

This round-based code scheduling shares some commonality with traditional list-based instruction scheduling [7]. A key difference is that the round-based algorithm works on dynamic instances of instructions (TLDG) rather than static instructions (as in traditional instruction dependence graphs).

A simple optimization is to compute the values of thread ID-specific conditional statements during the code generation process to save runtime execution of those branch statements.

Another by-product of the above code generation process is the change of memory-access pattern in the original GPU program. Since memory coalescing and layout transformation are often explicitly maintained by GPU programmers, we would normally expect the memory referencing code of the GPU program to produce relatively regular memory accesses. Therefore the unrolling of the original loop into CPU code might impair the sequentiality and locality of memory accesses. To alleviate this problem, we simply add a sorting process within each round to heuristically reduce the spatial distance between two adjacent references in the generated CPU code. A more detailed study is part of our future studies.

4.3 Instance-Level Redundancy Removal

Due to the massive parallelism of GPU and the sensitivity of its efficiency to conditional branches, it is common for a GPU application to contain some useless calculations. Such redundancy differs from the redundancy in traditional CPU code in that the corresponding statements are not completely redundant. It is common that the executions of them by some but not all threads are useful. An example is the bottom line in Figure 2 (b). Even though only the calculation by the first thread is useful, all threads in the thread block runs that statement. The execution results of these threads are ignored automatically, causing no harm to the GPU computing efficiency.

But they may impair the CPU efficiency substantially. Take parallel reduction as an example. As a fundamental parallel algorithm that produces relatively small amount of data from large number of input entries, parallel reductions are often implemented under the rationale of reducing the length of the critical path as much as possible rather than the utilization of the processor. A typical parallel reduction code taken from CUDA SDK shows that no iterations after the first one actually utilizes more than half of all threads, but the redundant threads perform calculations just like the small portion of useful threads, creating large waste of processor time that can not be effectively hidden on CPU.

With instance-level computations fully exposed, the TLDG described earlier offers the basis for finding such redundant calculations. As shown in figure 5, the TLDG-based redundancy removal starts from a list of "useful" nodes and traces upwards to the top of the graph, marking all the nodes encountered in the process as useful. The initial set of useful nodes are those containing returning values of the kernel.

After redundancy removal, the number of lines of code generated for the reduction kernel (when block size is 256) is reduced from more than 3000 to around 500, significantly cutting the number of instructions CPU needs to execute.

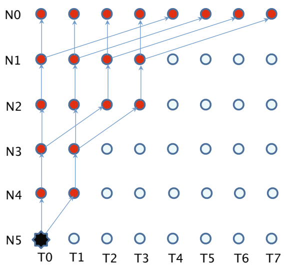
Similar to the optimization in code generation, redundancy removal can also be integrated into the code generation framework. With the large proportion of unnecessary memory references and conditional checks removed, the pruned code may outperform the original code substantially.

```
   1 {
   2 T*data,
   3 hash<int, T> tempBuf)
   4 if(tid[128]<256) tempBuf.insert(<0>, data[128]);
   5 if(tid[129]<256) tempBuf.insert(<1>, data[129]);
......
3458 if(tid[239]<32) data[239] += tempBuf.pop(239);
3459 if(tid[255]<32) data[255] += tempBuf.pop(255);
3460}
```

(a)

```
  1 {
  2 T*data,
  3 hash<int, T> tempBuf)
  4 tempBuf.insert(<0>, data[128]);
  5 tempBuf.insert(<1>, data[129]);
......
511  data[1]+=tempBuf.pop(1);
512  tempBuf.insert(<0>, data[1]);
513  data[0]+=tempBuf.pop(0);
514 }
```

(b)



(c)

Fig. 5. (a). The original generated code before redundancy removal. (b). Pruned code where all useless computations are removed. (c). Bottom-up redundancy removal, starting with the initial useful nodes, marked black.

Our discussions so far concentrate on block-level TLDG for handling explicit synchronizations and block-level redundant calculations. It is easy to see that the same approach applies to warp-level implicit synchronizations and redundancies. The only change needed is to replace the block-level TLDG with a warp-level TLDG.

5 Evaluation

In this section, we present experiment result using the TLDG framework on 3 benchmarks: `reduction` and `sortingNetwork` from the CUDA SDK examples, and the CUDA version of the `NPB CG` benchmark, a conjugate gradient application [1]. All three benchmarks demonstrate both explicit and implicit synchronizations.

To test the performance of our framework, our experiment was carried out on a quad-core Intel Xeon E5460 machine, with Linux 2.6.33 and GCC 4.1.2 installed. The compilations always use the highest level of optimization.

5.1 Versions

For each benchmark, we create five versions for comparison.

– *Baseline:* This version is generated by MCUDA, a typical existing GPU-to-CPU translator available to the public. Note that due to the negliction of implicit synchronizations, the translation results from this version may not be correct.
– *Merged Version:* This version is a simple extension of MCUDA. It addresses the implicit synchronization issue by treating them as explicit synchronizations—that is, the statements between every pair of implicit synchronizations becomes a separate loop. We then employ the loop fusion in existing C compilers to reduce the incurred loop overhead.
– *Split Version:* This version is produced by a statement-level dependence analysis proposed in a recent study [12]. It applies a dependence theory to identify critical implicit synchronization points in a kernel and conducts loop splitting at those points to handle implicit synchronizations. This approach may avoid the drawbacks of the merged version in creating too many small loops.
– *TLDG-basic Version:* This version is from our basic approach. It is based on instance-level dependence analysis as described in earlier sections. Besides handling implicit synchronizations correctly, it relaxes the order constraints imposed by both implicit and explicit synchronizations through the round-based scheduling.
– *TLDG-opt Version:* This version is the same as the TLDG-basic, but with the TLDG-based redundancy removal applied.

5.2 Experiment Results

In our experiment, the timing results correspond to the entire-kernel execution for `reduction` and `sortingNetworks`, while for `CG-CUDA`, the it corresponds to the time spent in the two reduction bodies on the *common* array.

In Table 1, we compare the performance of the five versions. The merged version lags behind all other versions with considerable slowdown. It is due to the high loop overhead introduced by the transformation and the limited capability of compilers in loop fusion. The split version always demonstrates similar performance as the baseline, proving the effectiveness of statement-level dependence analysis and the moderate overhead of the synchronizations inserted upon the analysis.

The TLDG-based version outperforms both merged and split versions significantly in two of the three benchmarks, even without the redundancy removal. The main reason for such an advantage is its fully unrolled instruction sequence, which has almost no loop overhead, and meanwhile, provides a chunk of linear code for compiler to optimize. The locality produced by the intra-round sorting may also contribute to the speedup to a certain degree.

Table 1. Relative Speedup over the (incorrect) baseline version

Versions	Reduction	Sorting	CG
Baseline	1	1	1
Merged	0.38	0.72	0.93
Split	0.63	1.01	0.98
TLDG-basic	1.34	1.58	3.16
TLDG-opt	1.61	1.58	12.47

One exception is `reduction`, where TLDG-based version shows the worst performance among all versions. The reason lies in the implementation details. Since the original loop structure is broken and then fused into a bigger function body, adjacent DRUs from the same GPU thread might be separated by a large number of instructions from other threads. To avoid introducing unnecessary variables renaming, we used a temporary buffer in the generated code to store the intermediate results of each DRU, as well as its own thread id to cope with the frequent condition calculations in the `reduction` kernel. As shown in figure 5 (a) and (b), this buffer is implemented as a hash table to enable rapid loop-up for the latest stored value of a particular GPU thread. Such a design introduces some additional memory accesses. With only two explicit synchronizations per kernel invocation in the `reduction` benchmark, the time saving from enlarged basic block in the CPU code is not sufficient to outweigh this overhead. In CG, the synchronizations are repeated in a loop, while in `sortingNetworks`, there are a large number of memory loads and stores from the swapping process. Both provide sufficient opportunities for the compiler to take advantage of.

The TLDG-opt version gives the best performance of all. The speedup comes from the downsized CPU code with all useless operations and condition calculations removed. On the CG benchmark, thanks to is large kernel size and block size, the speedup is the most prominent. `Reduction` also shows a significant speedup. The code size of these two benchmarks are reduced by a factor of 8 and 6.8 respectively, compared with their TLDG-basic versions. The program, `sorting`, shows no extra speedup as it contains no redundancies.

6 Related Work

A number of previous works have aimed at automatic compilation of GPU program onto CPU. MCUDA [9,10] and Ocelot [8]) both use an iterative execution framework based on the original GPU code structure to take advantage of its data and logical regularities. However, neither of them addresses the implicit synchronization pitfall, nor relaxes the constraints imposed by GPU synchronizations.

A recent study [12] analyzes the correctness issue caused by the negliction of implicit synchronizations. It proposes a state-level dependence theory to identify critical implicit synchronizations and generate correct CPU code. This current study concentrates on instance-level analysis and transformations.

NVIDIA provides a native emulation tool for running CUDA programs on CPU focuses on easing the debugging on GPU rather than improving performance [2]. Under the emulation mode, the programmer needs to manually insert macros to adapt to the current device at runtime. Although CUDA emulator provides the capability to run GPU program on CPU, it is not suitable for GPU-to-CPU automatic compilation. A similar case lies in OpenCL. While it allows the use of implicit synchronizations, it does not specify how they should be treated differently on different platforms, and the programmer again has to manually ensure the correctness of the cross-platform compilation [3]. Neither of them has attempted to relax order constraints or remove redundancies.

There have been many studies trying to ease GPU programming. A common approach is pragma-guided translation (e.g., from OpenMP to CUDA) [4,14]. Others have proposed extensions to CUDA or OpenCL (e.g. [16]). Dynamical optimization of GPU executions through either software (e.g. [5,6,13,17,19–21]) or hardware (e.g. [11,15,18]) techniques have shown large benefits recently. This current study is unique in focusing on the translation of synchronizations across devices.

7 Conclusion

In this paper, we examine the impact of explicit and implicit synchronizations on the compilation of GPU code to CPU. We propose an instance-level dependence analysis to help produce correct CPU code with efficiency optimized. The new approach employs TLDG to capture the dependences among dynamic instances of instructions of all threads in a thread block or warp. Assisted with round-based code generation and redundancy removal, it not only addresses the correctness issue in the treatment to implicit synchronizations by existing techniques, but also leads to significant speedups on three benchmarks.

Acknowledgment. We owe the anonymous reviewers our gratitude for their helpful suggestions on the paper. This material is based upon work supported by the National Science Foundation under Grant No. 0811791 and 0954015. 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. Hpcgpu project, http://hpcgpu.codeplex.com/
2. NVIDIA CUDA Programming Guide, http://developer.download.nvidia.com
3. OpenCL, http://www.khronos.org/opencl/
4. Ayguade, E., Badia, R.M., Cabrera, D., Duran, A., Gonzalez, M., Igual, F., Jimenez, D., Labarta, J., Martorell, X., Mayo, R., Perez, J.M., Quintana-Ortí, E.S.: A Proposal to Extend the OpenMP Tasking Model for Heterogeneous Architectures. In: Müller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP 2009. LNCS, vol. 5568, pp. 154–167. Springer, Heidelberg (2009)
5. Baskaran, M.M., Bondhugula, U., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P.: A compiler framework for optimization of affine loop nests for GPGPUs. In: ICS 2008: Proceedings of the 22nd Annual International Conference on Supercomputing, pp. 225–234 (2008)
6. Carrillo, S., Siegel, J., Li, X.: A control-structure splitting optimization for GPGPU. In: Proceedings of ACM Computing Frontiers (2009)
7. Cooper, K., Torczon, L.: Engineering a Compiler. Morgan Kaufmann (2003)
8. Diamos, G., Kerr, A., Yalamanchili, S., Clark, N.: Ocelot: A dynamic compiler for bulk-synchronous applications in heterogeneous systems. In: Proceedings of the Nineteenth International Conference on Parallel Architectures and Compilation Techniques. ACM (2010)
9. Stratton, J.A., Stone, S.S., Hwu, W.-M.W.: MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs. In: Amaral, J.N. (ed.) LCPC 2008. LNCS, vol. 5335, pp. 16–30. Springer, Heidelberg (2008)
10. Stratton, J.A., et al.: Efficient compilation of fine-grained SPMD-threadedprograms for multicore CPUs. In: CGO 2010 (2010)
11. Fung, W., Sham, I., Yuan, G., Aamodt, T.: Dynamic warp formation and scheduling for efficient GPU control flow. In: MICRO 2007: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 407–420. IEEE Computer Society, Washington, DC (2007)
12. Guo, Z., Zhang, E., Shen, X.: Correctly treating synchronizations in compiling fine-grained SPMD-threaded programs for CPU. In: Proceedings of International Conference on Parallel Architectures and Compilation Techniques (2011)
13. Hormati, A., Samadi, M., Woh, M., Mudge, T., Mahlke, S.: Sponge: Portable stream programming on graphics engines. In: ASPLOS 2011 (2011)
14. Lee, S., Min, S.-J., Eigenmann, R.: Openmp to GPGPU: a compiler framework for automatic translation and optimization. In: PPOPP 2009, pp. 101–110 (2009)

15. Meng, J., Tarjan, D., Skadron, K.: Dynamic warp subdivision for integrated branch and memory divergence tolerance. In: ISCA 2010 (2010)
16. Michel, S., Philipp, K., Sergei, G.: Skelcl - a portable skeleton library for high-level GPU programming. In: IPDPS 2011 (2011)
17. 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 2008: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 73–82 (2008)
18. Tarjan, D., Meng, J., Skadron, K.: Increasing memory latency tolerance for SIMD cores. In: SC 2009 (2009)
19. Yang, Y., Xiang, P., Kong, J., Zhou, H.: A GPGPU compiler for memory optimization and parallelism management. In: PLDI (2010)
20. Zhang, E.Z., Jiang, Y., Guo, Z., Shen, X.: Streamlining GPU applications on the fly. In: Proceedings of the ACM International Conference on Supercomputing, ICS, pp. 115–125 (2010)
21. Zhang, E.Z., Jiang, Y., Guo, Z., Tian, K., Shen, X.: On-the-fly elimination of dynamic irregularities for GPU computing. In: ASPLOS 2011 (2011)

A Mutable Hardware Abstraction to Replace Threads

Sean Halle[1,2,3] and Albert Cohen[1]

[1] INRIA and École Normale Supérieure, France
[2] University of California at Santa Cruz, USA
[3] Technical University Berlin, Germany

Abstract. We propose an abstraction to alleviate the difficulty of programming with threads. This abstraction is not directly usable by application programmers. Instead, application-visible behavior is defined through a semantical plugin, and invoked via a language or library that uses the plugin. The main benefit is that parallel language runtimes become simpler to implement, because they use sequential algorithms for the parallel semantics. This is possible because the abstraction makes available a virtual time in which events in different program time-lines are sequentialized. The parallel semantics relate events in different time-lines via relating the sequentialized versions within the virtual time-line.

We have implemented the abstraction in user-space and demonstrate its low overhead and quickness to implement a runtime on three sets of parallelism constructs: rendez-vous style `send` and `receive`; Cilk style `spawn` and `sync`, which have similar performance to Cilk 5.4; and `mutex` and `condition variable` constructs from pthreads, which have 80x lower overhead than Linux thread operations. Development time averaged around two days per set, versus an expected duration of weeks to modify a thread-based runtime system.

1 Motivation

Thread parallelism constructs have been well documented to be difficult to program with. They directly expose low-level concurrency to the programmer. Arbitrary non-deterministic behavior and deadlocks can arise from improperly synchronized code. Efficient execution requires non-blocking algorithms whose correctness require deep understanding of weakly consistent memory models. In addition, the operating system abstraction for threads comes with a very high context-switching and synchronization overhead.

A Partial Solution. To deal with the last problem, a parallel language's runtime turns off operating system threads by pinning one to each physical core. This way, the custom runtime is assured that the software thread is one-to-one with a physical core. It then implements a user-level thread package that lets it control which OS thread a computational task is assigned to. Finally, the runtime then implements the language's parallel semantics in terms of those user threads.

The user-level threading approach addresses the system overhead issue, but it still hides important events such as input-output or node-to-node communications in a cluster. Scalable runtimes need to coordinate task assignment to cores with application access of input and output, memory allocation over non-uniform cache and memory hierarchies, offloading to hardware accelerators, power management, and inter-node communication in a cluster. The user-level threading approach is hampered in addressing these needs, and further makes the parallel runtime implementation cumbersome, error-prone and complex, because it is still written in terms of threads.

Ideally, the OS would be in terms of a mutable hardware abstraction, and export mutations as new behavior. We define a mutable hardware abstraction to be an interface to hardware-level behaviors that are normally inside the OS or below it. Examples include communication between cores, allotting time-slots to applications, and establishing ordering of events among cores (which is what atomic memory operations and equivalent patterns of instructions do). The kernel itself would be implemented in terms of such an abstraction, and would accept mutations the same way it accepts device-drivers. It would then export the mutated behaviors for the language to trigger.

A language runtime would be in the form of a mutation. Being inside the OS, it has secure access to kernel-only hardware mechanisms. It could directly negotiate with the kernel to manage physical resources, in a low-overhead way. The arrangement enables the runtime to control which task is assigned to which processing element at what time. Both high performance and low-energy depend on this for implementing data affinity techniques. For example, the runtime could track data within the memory hierarchy and assign tasks to locations close to their consumed data.

Contribution. We show in this paper the definition and implementation of such a mutable hardware abstraction, albeit at user-level rather than in the kernel. The abstraction lets a language's runtime be implemented as a mutation, which we call a plugin. The plugin implements parallelism constructs and assignment of tasks to cores.

We focus in this introductory paper on the definition of the abstraction and its support for parallelism constructs, postponing exploration of assignment of tasks onto cores and other performance optimizations to following papers. This paper defines multiple time-lines in a program, and a virtual timeline that globally orders events among them. It demonstrates three sets of parallelism constructs: synchronous `send-receive` motivated by process calculi; `spawn` and `sync` from Cilk [7,10]; and `mutex` and `condition variable` from pthreads. The assignment policy we implemented with them is simply first-come first-served.

We call the abstraction Virtualized Master-Slave, or VMS. It exports facilities to create virtual processors and control how their timelines relate to each other, and relate to physical time. It also exports facilities to suspend a virtual processor and for an executable to interact with the plugin. The plugin embodies most of a language's runtime. A wrapper-library or keyword is what appears in application code, and is what triggers the runtime.

Organization of Paper. Section 3 provide the original concepts and definitions of VMS. Section 4 focuses on the implementation, describing the elements and how they interact, then relating them back to the theoretical definition. Section 5 takes the point of view of the application code, studying the usage and implementation of parallel language constructs as a VMS plugin. To wrap up, measurements of effectiveness appear in Section 6 and conclusions in Section 7.

2 Background and Related Work

User-level thread packages and most parallel language runtimes have to side-step OS threads, by pinning one to each core, which effectively gives the user-level package control over the core. Our VMS implementation also does this. We are not claiming in this paper to have the OS level implementation of VMS that is possible – but just the user-space version.

Related Work. The most primitive methods for establishing ordering among cores or distributed processors are atomic instructions and clock-synchronization techniques [16,4].

Meanwhile, the most closely related work is Scheduler Activations [2], which also allows modifying concurrency constructs and controlling assignment of virtual processors onto cores. However it has no virtual time to guarantee globally consistent sequentialization, and no interface for plugins.

BOM [6], which is used in Manticore to express scheduling policies and synchronization, also bears resemblances to VMS, but at a higher level of abstraction. BOM is a functional language, rather than a primitive abstraction meant to sit at the hardware-software boundary as VMS is.

Coroutines is a high-performance means of switching between tasks. Coroutine scheduling and stack handling techniques were well suited to the user-space implementation of VMS.

Other related work either provides an abstraction of the thread model, or is a full language with specific parallelism constructs. As a protypic example of user-level threads, Cilk [7,10] provides a simplified abstraction with an efficient scheduling and load balancing algorithm, but limited to fork-join concurrency. OpenMP [18] is a typical example of a parallel extension of sequential languages; it allows creating tasks and controlling their execution order. We claim that both Cilk and OpenMP, as well as most thread abstractions or parallel languages may be implemented via plugins to VMS, with similar performance.

VMS is unique in that it doesn't impose its own concurrency semantics as a programming model, but rather takes preferred ones as plugins. This makes it only a *support* mechanism to implement language runtimes – VMS is hidden from the application, underneath the language. Parallelism constructs may be implemented as VMS plugins, easily, quickly, and with high performance as indicated in Section 6.

This work presents a first incarnation of VMS. We plan to explore the embedding into VMS of a variety of parallel languages, with a special interest for coordination languages [8,15,5]. We will also explore VMS's compatibility with different concurrent semantics [14,13,17,12,1]. One particularly important application would be to use VMS to facilitate the design and implementation of the emerging hybrid programming models, such as MPI+OpenMP, or OpenMP+OpenCL [3,9].

3 Abstract Definition of VMS

We start with an intuitive overview, then precise the definitions and properties in the following sub-sections.

Definitions: 1) We want to avoid the confusion associated with the various interpretations for the terms "thread" and "task" so will use the term *Virtual Processor* (VP), which we define as state in combination with the ability to animate code or an additional level of virtual processors. The state consists of a program counter, a stack with its contents, a pointer to top of stack, and a pointer to the current stack frame. 2) A *physical processor* executes a sequential stream of instructions. 3) A program-timeline is the sequence of instructions animated by a Slave VP, which is in turn animated by a physical processor.

Intuitive Overview. VMS can be understood via an analogy with atomic instructions, such as Compare and Swap (CAS). These are used to establish an ordering among the timelines of cores. They consist of two parts: 1) the semantics of what is done to the memory location, 2) a mechanism that establishes an ordering among the cores. For CAS, the semantics are: "compare value in this register to value at the address, and if same, then put value in second register into the address." Multiple kinds of atomic instructions share the same order-establishing mechanism, they simply provide different semantics as a front-end.

VMS can be viewed as virtualizing the order-establishing mechanism. It allows the semantics to be plugged-in to it. This breaks concurrency constructs into two parts: the VMS mechanism, which establishes an ordering between events in different timelines; and the plugin, which supplies the semantics.

Below the interface, hardware mechanisms are employed to order specific points in one physical processor's timeline relative to specific points in another's timeline. Above the interface, a plugin provides the semantics that an application uses to invoke creation of the ordering.

Together, VMS plus the plugin form a parallelism construct, by which an application controls how the time-lines of its virtual processors relate. Such constructs also guarantee relations of VP time-lines to hardware events.

As an example, consider a program where one VP writes into a data structure then calls a `send` construct. Meanwhile, a different VP calls the `receive` construct then reads the data structure. The semantics of the `send` and `receive` constructs are that all data written before the `send` is readable in the other time-line after the `receive`. To implement these constructs, VMS provides the mechanism to enforce the ordering, and to include the writes and reads in that ordering. The plugin directs that mechanism to order the `send` event before the `receive` event.

What the VMS Interface Provides: The interface provides primitive operations to create and suspend VPs; a way for plugins to control when and where each VP is (re)started; a way for application code to send requests to the plugin; and a way to order a specific point in one VP time-line relative to a specific point in another VP time-line. All implementations of the VMS interface provide these, whether it is on shared memory or distributed, with strong memory consistency or weak.

Specification in Three Parts. We specify the observable behavior of a VMS system *with plugins present*. Hence, the specified behaviors remain valid with any parallelism construct implementable with VMS. First we give the specification of a computation system that VMS is compatible with; then specify a notion of time and the key VMS guarantee; and lastly specify virtual processor scheduling states and transitions between them.

3.1 The Specifications for a VMS-compatible Computation System

- An application creates multiple VPs, which are Slaves, each with an independent time-line.
- A schedule of Slaves is generated by a Master entity, from within a hidden time-line(s).
- A schedule is the set of physical locations and time-points at which Slaves are (re)animated.
- All semantic parallelism behavior is invoked by Slaves communicating with the Master.
- A Slave communicates with the Master by using a VMS primitive, which suspends the Slave.

Where We Define: Semantic Parallelism Behavior is the actions taken by a parallelism construct, which establishes an ordering among events in different Slave timelines.

Discussion: The key point is that *scheduling is separated from the application code.* This is enforced by the schedule being generated in a time-line hidden from the application. The rest of the requirements are consequences of that separation.

The Master entity appears as a single entity, to the slaves. However it may be implemented with multiple (hidden) timelines. This is the approach taken in our initial implementation, which has several Master VPs hidden inside the VMS implementation.

3.2 The Time-Related Specifications of VMS

To prepare for the time-related specifications, we give an advance peek of the following section, 3.3. There, Slave VPs are specified to have three scheduling states: Animated, Blocked, and Ready. When a parallelism construct starts execution, the Slave transitions from Animated to Blocked. When it ends execution, the Slave transitions from Blocked to Ready. VMS provides a way to control the order of these state-transitions, which is equivalent to controlling the order of the parallelism-constructs. Controlling the state transitions is how the ordering among constructs in different timelines is established.

With that background, here are time-related specifications for VMS:

– VMS provides a Virtual timeline that globally orders changes of scheduling state of Slave VPs.
– Ordering is created among construct-invocations by controlling the order of Blocked to Ready transitions in the Virtual timeline.
– Causally tied construct-invocations are tied-together inside the Master.
– VMS enforces ordering of *observations* of physical events in Slave timelines to be consistent with the Virtual time ordering.
– Virtual time defines only ordering, but not spans, nor widths.

Discussion: Most importantly here, Virtual time defines a global ordering among Slave state-transitions. To make this useful for parallelism, VMS must be implemented so that observations of physical events, like reads and writes to shared memory, are consistent with that ordering.

The Virtual timeline plays the same role as the mechanism added to memory systems to support atomic instructions. All atomic instructions require hardware that establishes an ordering among the timelines of physical cores. That hardware sequentializes execution of atomic memory accesses to the same address. VMS virtualizes this mechanism. It provides the same ordering function.

An important point is that the Virtual timeline is generated inside the Master. When a Slave uses the VMS primitive to send a parallelism-construct request, it suspends. However, that Slave doesn't actually transition state from Animated to Blocked until the Master *acknowledges* the suspension. It is the acknowledgement that adds the Slave transition into the Virtual timeline.

The essential value of VMS is using it to control the order of observing events. It has to be able to causally link the execution of a parallelism construct in one timeline to the execution of a construct in a different timeline. Establishing such a causal link is called *tying together* two construct executions. It is specific executions from different timelines that are causally linked with such a tie.

The Key VMS Guarantee: the order of observing physical events is consistent with the order of tied together parallelism constructs.

To explain this, take as given: two Slaves both execute parallelism constructs, those are tied together by the Master, establishing a causal ordering from one to the other. So, one construct is the *before*-construct, the other is the *after*-construct. Now, the guarantee means that any events triggered before the before-construct, in its timeline, are guaranteed to be detected in the other timeline as also preceding the after-construct. In addition, events triggered after the after-construct are guaranteed not visible before the before-construct in its timeline. This two-part guarantee is the result of the above specifications of VMS's time-related behavior.

Definitions: Some more definitions, to prepare for the next explanation. 1) An *ordering-point* exists in a Slave VP's timeline as a zero-width event that can be tied to ordering points in other timelines. It is initiated by a Slave VP executing the suspend primitive, and ended by the Master transitioning the Slave back to Animated. 2) A trace-segment is a portion of a Slave VP's stream of instructions bounded by ordering-points.

Hence, the timeline of a Slave is a sequence of trace-segments. Each trace-segment is animated by a single physical processor, but not necessarily the same as animated the Slave's other trace segments.

Relating Time-Lines to Each Other. Figure 1 shows two ordering points being tied together. A trace segment starts, at the same time an ordering-point ends, by its Slave transitioning to Animated. Because the transition to Animated exists as a point in Virtual time, the start of a trace-segment has a well-defined position within Virtual time. Likewise, a trace-segment is ended by its Slave executing the suspend primitive of VMS. Although this does not have a well-defined point in Virtual time, every execution of suspend is acknowledged by the Master, which transitions the Slave to Blocked. That transition does have a well-defined position in Virtual time. Hence, the end of every trace-segment is associated with a well-defined position in Virtual time.

As a result, trace segments can be ordered relative to each other, by checking their start and end points in Virtual time. If they have no overlap in virtual time then they have a total ordering. However, if any portion of them overlaps in Virtual time, then they are considered concurrent trace-segments, and their Slaves are considered to be executing in parallel between those points of Virtual time.

Note that this is conservative because it doesn't take into account the physical wait time between a Slave suspending and the Master acknowledging. The Slave may stop executing at a physical time-point that would map onto an earlier point in Virtual time. In some cases, ending the Slave's trace-segment at the earlier point would eliminate the overlap with a particular other trace-segment. But VMS's set of specifications doesn't allow such mapping of suspend-execution onto Virtual time (for implementation-related reasons, which require downloading the

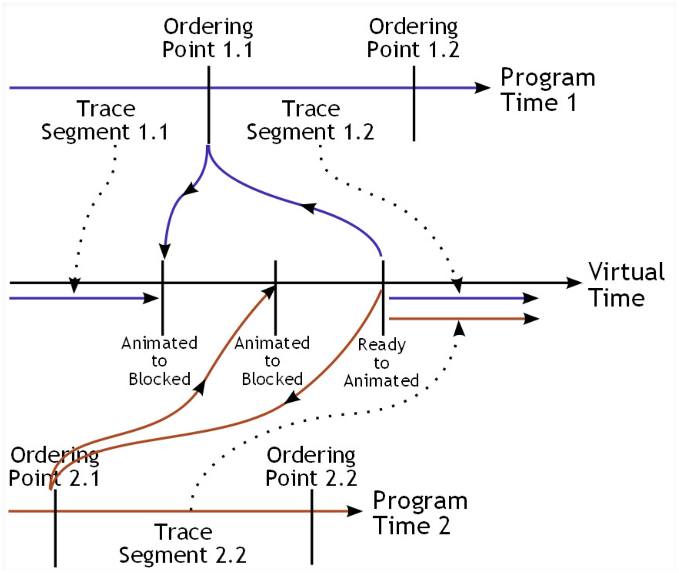


Fig. 1. Time Behaviors: Shows Ordering Point 1.1 being tied to Ordering Point 2.1. As a result, VMS guarantees that events triggered in Trace Segment 1.1 are seen as having taken place in the past in Trace Segment 2.2. Also shows that there is no common tied ordering point between segments 1.2 and 2.2, so VMS provides no guarantees about what order segment 2.2 sees events triggered in segment 1.2

code and gaining experience with it, to establish a common language, for an explanation to be understood).

A subtlety is that events triggered before one tied ordering-point, *might* be visible in the other timeline before the other tied ordering-point. In the figure, segment 2.1 might be able to see events from segment 1.1 if it looked. The VMS guarantee doesn't cover overlapped trace-segments. Physical events triggered before are only guaranteed visible *after* the tie point, and events after are only guaranteed *not* visible *before* the tie point.

We call this bounded non-determinism, because events within overlapped trace-segments have non-deterministic ordering, but the region of non-determinism can be bounded by tied ordering-points. This allows a program to specify non-determinism, but control the region of non-deterministic behavior. For example, a reduction construct could be created that non-deterministically assigns portions of the reduction work to overlapped Slave segments. It would tie together ordering points from all the Slaves that mark the end of reduction. Hence, the outcome is deterministic, but the path to get there is not.

Sequential Algorithms for Parallel Constructs. The globally-consistent sequential order in Virtual time enables one of VMS's main benefits: sequential algorithms for parallel constructs. An implementation to tie ordering points together equals an implementation of parallel constructs. A plugin has an ordering of state transitions available, and chooses from those. Sequential algorithms rely on an ordering existing, while concurrent algorithms must include operations that establish an ordering. Plugins have Virtual time ordering available, so they can use sequential algorithms.

3.3 Specification of Scheduling State

Scheduling state is used in VMS to organize internal activity, for enforcing the guarantees.

- VPs have three scheduling states: *Animated*, *Blocked*, *Ready*; see Figure 2.
- VPs in Animated are *allowed* to advance Program time with (core-local) physical time.
- VPs in Blocked and Ready do not advance their Program time.
- Animated has two physical states: *Progressing* and *Stalled*.
- VPs in Progressing advance Program time with (core-local) physical time, those in Stalled do not (allowing non-semantic suspend for hardware interrupts).
- Scheduling states are defined in Virtual time only.
- Progressing and Stalled are defined in (core-local) physical time only; the distinction is invisible in Virtual time.

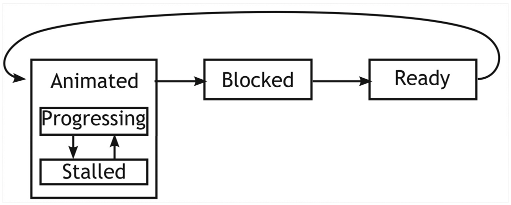


Fig. 2. Scheduling states of a slave VP in the VMS model. Animated, Blocked, and Ready are only defined in Virtual Time and only visible in the Master. Progressing and Stalled are only visible in physical-processor local time, not visible in Virtual time.

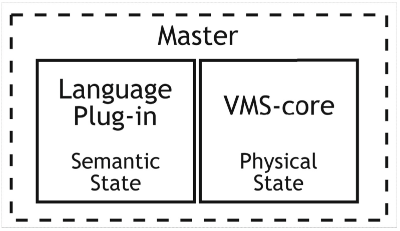


Fig. 3. The Master, split into a generic core and a language-specific plugin. The core encapsulates the hardware and remains the same across applications. The plugin implements the semantics of the parallelism-constructs.

Some important points: (1) only VPs Animated can trigger physical events that are seen in other program time-lines; (2) the distinction between Blocked vs Stalled is that a Slave has to explicitly execute a VMS primitive operation to enter Blocked. In contrast, Stalled happens invisibly, with no effect on semantic behavior. It is due to hardware events hidden inside VMS, such as interrupts.

The Ready state is used to separate the parallelism-construct behavior from the scheduling behavior. It acts as a "staging area" for scheduling. VPs placed into this state are *ready* to be animated, but the scheduler decides when and where.

An interesting point is that in VMS, the causal tie between timelines is created by actions *outside* program timelines. In contrast, memory-based lock algorithms place the concurrency-related behavior *inside* program timelines.

Transition Between Slave Scheduling States.

- VPs transition states as shown in Figure 2.
- Animated→Blocked is requested by a Slave executing suspend, but takes place in Virtual time at the point the Master acknowledges that request.
- Blocked→Ready is determined by the semantics implemented in the plugin.
- Ready→Animated is determined by the scheduler in the plugin.
- Transitions in scheduling state have a globally consistent order in Virtual time.

The parallelism primitives executed by a program do not directly control change in scheduling states. Rather they communicate messages to the Master, via a VMS supplied primitive. If it suspended when sending the request, then the act of the Master acknowledging the request places the Animated→Blocked transition into Virtual time. Inside the Master, the plugin then processes the message. Based on contents, it performs changes in state from Blocked→Ready, creates new VPs, and dissipates existing VPs. Most communication from Slave to Master requires the Slave to suspend when it sends the message. A few messages, like creating new Slave may be sent without suspending.

The suspend primitive decouples local physical time from Virtual time. Execution causes immediate transition to Stalled in physical time, later the Master performs Animated→Blocked, fixing that transition in Virtual time. The only relationship is causality. This weak relation is what allows suspension-points to be serialized in Virtual time, which in turn is what allows using sequential algorithms to implement parallelism constructs.

3.4 Plugins

The Master entity has two parts, a generic core part and a plugin (Figure 3). The core part of the Master is implemented as part of VMS itself. The plugin supplies two functions: the communication-handler and the scheduler, both having a standard prototype. The communication-handler implements the parallelism constructs, while scheduler assigns VPs to cores.

An *instance* of a plugin is created as part of initializing an application, and the instance holds the semantic and scheduling state for that run of the application. This state, combined with the virtual processor states of the slaves created during that application run, represents progress of the work of the application. For example, multi-tasking is performed simply by the Master switching among plugin instances when it has a resource to offer to a scheduler. The parallelism-semantic state holds all information needed to resume (hardware state, such as TLB and cache-tags is inside VMS).

4 Internal Workings of Our Implementation

For our example implementation, we name the elements and describe their logical function, then relate those to the abstract model. We then step through the operation of the elements.

Elements and Their Logical Function. As illustrated in Figure 4, our VMS implementation is organized around physical cores. Each core has its own *master virtual-processor*, `masterVP`, and a *physical-core controller*, which communicate via a set of scheduling slots, `schedSlot`. The Master in the abstract definition is implemented by the multiple `masterVP`s plus a particular plugin instance with its shared parallelism-semantic state (seen at the top).

On a given core, only one of: the core-controller, `masterVP`, or a slave VP, is animated at any point in local physical time. Each `masterVP` animates the same function, called `master_loop`, and each slave VP animates a function from the application, starting with the top-level function the slave is created with, and following its call sequence. The core controller is implemented here as a Linux pthread that runs the `core_loop` function.

Switching between VPs is done by executing a VMS primitive that suspends the VP. This switches the physical core over to the controller, by jumping to the start of the `core_loop` function, which chooses the next VP and switches to that (switching is detailed in Section 5 Figure 7).

Relation to Abstract Model. We chose to implement the Master entity of the model by a set of `masterVP`s, plus plugin functions and shared parallelism-semantic state. VMS consists of this implementation of the Master, plus the core-controllers, plus the VMS primitive libraries, for creating new VPs and dissipating existing VPs, suspending VPs, and communicating from slave VP to Master. In Figure 4, everything in green is part of VMS, while the plugin is in red, and application code appears as blue, inside the slave VP.

Virtual time in the model is implemented via a combination of four things: a `masterLock` (not shown) that guarantees non-overlap of `masterVP` trace-segments; the `master_loop` which performs transition Animated→Blocked; the `comm_handler_fn` which performs Blocked→Ready and the `scheduler_fn` which performs Ready→Animated. Each state transition is one step of Virtual time; is guaranteed sequential by the non-overlap of `masterVP` trace segments; and is global due to being in parallelism-semantic state that is shared (top of Figure 4).

Transitions Progressing⇄Stalled within the Animated state are invisible to the parallelism semantics, the Master, and Virtual time, and so have no effect on the elements seen.

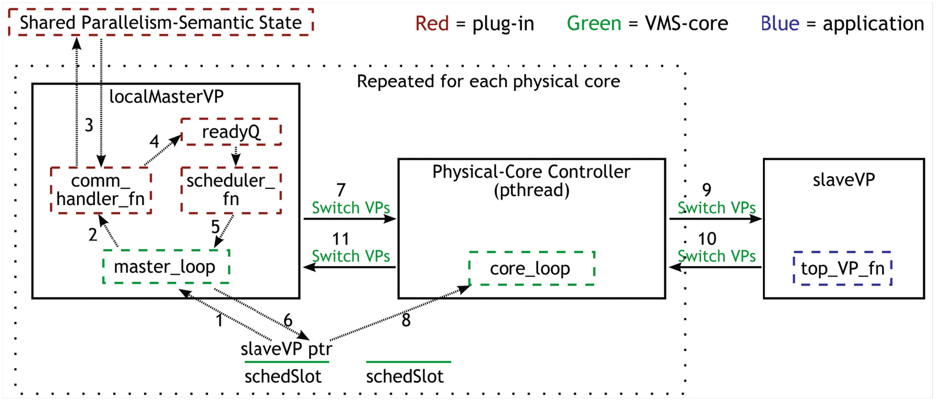


Fig. 4. Internal elements of our example VMS implementation

Steps of Operation. The steps of operation are numbered, in Figure 4. Taking them in order:

1. `master_loop` scans the scheduling slots to see which ones' slaves have suspended since the previous scan.
2. It hands these to the `comm_handler_fn` plugged in (which equals transition Animated→Blocked).
3. The VP has a request attached, and data in it causes the `comm_handler_fn` to manipulate data structures in the shared parallelism-semantic state. These structures hold all the slaves in the blocked state (code-level detail and example will come in Figure 8, Section 5).
4. Some requests cause slaves to be moved to a `readyQ` on one of the cores (Blocked→Ready). Which core's `readyQ` receives the slave is under plugin control, determined by a combination of request contents, semantic state and physical machine state.
5. During the scan, the `master_loop` also looks for empty slots, and for each calls the `scheduler_fn` plugged in. It chooses a slave from the `readyQ` on the core animating `master_loop`.
6. The `master_loop` then places the slave VP's pointer into the scheduling slot (Ready→Animated), making it available to the `core_loop`.
7. When done with the scan, `masterVP` suspends, switching animation back to the `core_loop`.
8. `core_loop` takes slave VPs out of the slots.
9. Then `core_loop` switches animation to these slave VPs.
10. When a slave self-suspends, animation returns to the `core_loop` (detail in code in Figure 9), which picks another.
11. Until all slots are empty and the `core_loop` switches animation to the `masterVP`.

Enabling Sequential Implementation of Parallelism Semantics. All these steps happen on each core separately, but we use a central `masterLock` to ensure that only one core's `masterVP` can be active at any time. This guarantees non-overlap of trace-segments from different `masterVPs`, allowing the plugins to use sequential algorithms, without a performance penalty, as verified in Section 6.

Relating this to the abstract model: the parallelism-semantic behavior of the Master is implemented by the communication handler, in the plugin. It thus runs in the Master time referred to, in the model, in Section 3. Requests are sent to the Master by self-suspension of the slaves, but sit idle until the other slaves in the scheduling slots have also run. This is the passive behavior of requests that was noted in Section 3, which allows the `masterVPs` to remain suspended until needed. This in turn enables the `masterVPs` from different cores to be non-overlapped. It is the non-overlap that enables the algorithms for the parallelism semantics to be sequential.

5 Code Example

To relate the abstract model and the internal elements to application code and parallelism-library code, we give code snippets that illustrate key features. We start with the application then work down through the sequence of calls, to the plugin, using our SSR [11] parallelism-library as an example.

In general, applications are either written in terms of a parallel language that has its own syntax, or a base language with a parallelism library, which is often called an *embedded language*. Our demonstrators, VCilk [11], Vthread, and SSR, are all parallelism libraries. A parallel language would follow the standard practice of performing source-to-source transform, from custom syntax into C plus parallelism-library calls.

SSR. SSR stands for Synchronous Send-Receive, and details of its calls and internal implementation will be given throughout this section. It has two types of construct. The first, called *from-to* has two calls: `SSR_send_from_to` and `SSR_receive_from_to`, both of which specify the sending VP as well as the receiving VP. The other, called *of-type* also has two calls: `SSR__send_of_type_to` and `SSR__receive_of_type`, which allow a receiver to accept from anonymous senders, but select according to type of message.

Application View. Figure 5 shows snippets of application code, which use the SSR parallelism library. The most important feature is that all calls take a pointer to the VP that is animating the call. This is seen at the top of the figure where slave VP creation takes a pointer to the VP asking for creation. Below that is the standard prototype for top level functions, showing that the function receives a pointer to the VP it is the top level function for.

The pointer is placed on the stack by VMS when it creates the VP, and is the means by which the application comes into possession of the pointer. This animating VP is passed to all library calls made from there. For example, the

Creating a new processor:

```
newProcessor = SSR__create_procr( &top_VP_fn, paramsPtr, animatingVP );
```

prototype for the top level function:

```
top_VP_fn( void *parameterStrucPtr, VirtProcr *animatingVP );
```

handing animating VP to parallelism constructs:

```
SSR__send_from_to( messagePtr, animatingVP, receivingVP );
messagePtr = SSR__receive_from_to( sendingVP, animatingVP );
```

Fig. 5. Application code snippets showing that all calls to the parallelism library take the VP animating that call as a parameter

bottom shows a pointer to the animating VP placed in the position of sender in the `send` construct call. Correspondingly, for the `receive` construct, the position of receiving VP is filled by the VP animating the call.

Relating these to the internal elements of our implementation, the `animatingVP` suspends inside each of these calls, passing a request (generated in the library) to one of the `masterVPs`. The `masterVP` then calls the `comm-handler` plugin, and so on, as described in Section 4.

For the `SSR__create_processor` call, the comm-handler in turn calls a VMS primitive to perform the creation. The primitive places a pointer to the newly created VP onto its stack, so that when `top_VP_fn` is later animated, it sees the VP-pointer as a parameter passed to it. All application code is either such a top-level function, or has one at the root of the call-stack.

The send and receive calls both suspend their animating VP. When both have been called, the communication handler pairs them up and resumes both. This ties time-lines together, invoking the VMS guarantee. Both application-functions know, because of the VMS guarantee (Section 3), that writes to shared variables made before the send call by the sender are visible to the receiver after the receive call. This is the programmer's view of tying together the local time-lines of two different VPs, as defined in Section 3.

Concurrency-Library View. A parallelism library is a wrapper. Each call, in general, only creates a request, sends it, and returns, as seen below. To send a request, it uses the combined request-and-suspend VMS primitive that attaches the request then suspends the VP. The primitive requires the pointer to the VP, to attach the request and to suspend it.

In Figure 6, notice that the request's data is on the stack of the virtual processor that's animating the call, which is the `receiveVP`. The `VMS__send_sem_request` suspends this VP, which changes the physical core's stack pointer to a different stack. So the request data is guaranteed to remain undisturbed while the VP is suspended.

Figure 7 shows the implementation of the VMS suspend primitive. As seen in Figure 4, suspending the `receiveVP` involves switching to the `core_loop`. In our implementation, this is done by switching to the stack of the pthread pinned to the physical core and then jumping to the start-point of `core_loop`.

This code uses standard techniques commonly employed in co-routine implementations. Tuning effort spent in `core_loop` is inherited by all applications.

Plugin View. SSR's communication handler dispatches on the `reqType` field of the request data, as set by the `SSR__receive_from_to` code. It calls the handler code in Figure 8. This constructs a hash-key, by concatenating the from-VP's pointer with the to-VP's pointer. Then it looks-up that key in the hash-table that SSR uses to match sends with receives, which is in the shared semantic state seen at the top of Figure 4 in Section 4.

The most important feature in Figure 8 is that both send and receive will construct the same key, so will find the same hash entry. Whichever request is handled first in Virtual time will see the hash entry empty, and save itself in that entry. The second to arrive sees the waiting request and then resumes both VPs, by putting them into their `readyQ`s.

Access to the shared hash table can be considered private, as in a sequential algorithm. This is because our VMS-core implementation ensures that only one handler on one core is executing at a time.

6 Results

We implemented blocked dense matrix multiplication with right sub-matrices copied to transposed form, and ran it on a 4-core Core2Quad 2.4Ghz processor.

Implementation-Time. As shown in Table 1, time to implement the three parallel libraries averages 2 days each. As an example of productivity, adding nested transactions, parallel singleton, and atomic function-execution to SSR required a single afternoon, totaling less than 100 lines of C code.

Execution Performance. Performance of VMS is seen in Table 2. The code is not optimized, but rather written to be easy to understand and modify. The majority of the plugin time is lost to cache misses because the shared parallelism-semantic state moves between cores on a majority of accesses. Acquisition of the master lock is slow due to the hardware implementing the CAS instruction.

Existing techniques will likely improve performance, such as localizing semantic data to cores, splitting malloc across the cores, pre-allocating slabs that are recycled, and pre-fetching. However, in many cases, several hundred nanoseconds per task is as optimal as the applications can benefit from.

Head to Head. We compare our implementation of the `spawn` and `sync` constructs against Cilk 5.4, on the top in Table 3, which shows that the same application code has similar performance. For large matrices, Cilk 5.4's better

```
void * SSR__receive_from_to( VirtProcr *sendVP, VirtProcr *receiveVP )
 { SSRSemReq  reqData;
   reqData.receiveVP = receiveVP;
   reqData.sendVP    = sendVP;
   reqData.reqType   = receive_from_to;
   VMS__send_sem_request( &reqData, receiveVP );
   return receiveVP->dataReturnedFromRequest;
 }
```

Fig. 6. Implementation of SSR's receive_from_to library function

```
VMS__suspend_procr( VirtProcr *animatingVP )
 { animatingVP->resumeInstrAddr = &&ResumePt; //GCC takes addr of label
   animatingVP->schedSlotAssignedTo->isNewlySuspended = TRUE; //for master_loop to see
   <assembly code stores current physical core's stack reg into animatingVP struct>
   <assembly code loads stack reg with core_loop stackPtr, which was saved into animatingVP>
   <assembly code jmps to core_loop start instr addr, which was also saved into animatingVP>
 ResumePt:
   return;
 }
```

Fig. 7. Implementation of VMS suspend processor. Re-animating the virtual processor reverses this sequence. It saves the `core_loop`'s resume instr-addr and stack ptr into the VP structure, then loads the VP's stack ptr and jmps to its `resumeInstrAddr`.

```
handle_receive_from_to( VirtProcr *requestingVP, SSRSemReq *reqData, SSRSemEnv *semEnv )
 { commHashTbl = semEnv->communicatingVPHashTable;
   key[0] = reqData->receiveVP;   key[1] = reqData->sendVP; //send uses same key
   waitingReqData = lookup_and_remove( key, commHashTbl );  //get waiting request
   if( waitingReqData != NULL )
    { resume_virt_procr( waitingReqData->sendVP );
      resume_virt_procr( waitingReqData->receiveVP );
    }
   else
      insert( key, reqData, commHashTbl ); //receive is first to arrive, make it wait
 }
```

Fig. 8. Pseudo-code of communication-handler for `receive_from_to` request type. The `semEnv` is a pointer to the shared parallel semantic state seen at the top of Figure 4.

use of the memory hierarchy (the workstealing algorithm) achieves 23% better performance. However, for small matrices, VCilk is better, with a factor 2 lower overhead. Cilk 5.4 does not allow controlling the number of spawn events it actually executes, and chooses to run smaller matrices sequentially, limiting our comparison.

When comparing to pthreads, our VMS based implementation has more than an order of magnitude better overhead per invocation of mutex or condition variable functionality, as seen on the bottom of Table 3. Applications that inherently have short trace segments will synchronize often and benefit the most from Vthread.

Table 1. Person-days to design, code, and test each parallelism library. L.O.C. is lines of (original) C code, excluding libraries and comments.

	SSR	Vthread	VCilk
Design	4	1	0.5
Code	2	0.5	0.5
Test	1	0.5	0.5
L.O.C.	470	290	310

Table 2. Cycles of overhead, per scheduled slave. "comp only" is perfect memory, "comp + mem" is actual cycles. "Plugin-concur" only concurrency requests, "plugin-all" includes create and malloc requests. Two significant digits due to variability.

		comp only	comp + mem
VMS Only	`master_loop`	91	110
	switch VPs	77	130
	(malloc)	160	2300
	(create VP)	540	3800
Language:			
SSR	plugin – concur	190	540
	plugin – all	530	2200
	lock		250
Vthread	plugin – concur	66	710
	plugin – all	180	1500
	lock		250
VCilk	plugin – concur	65	260
	plugin – all	330	1800
	lock		250

Table 3. On left, exe time in seconds for MM. To the right, overhead for pthread vs. Vthread. First column is cycles for perfect memory and second is total measured cycles. pthread cycles are deduced from round-trip experiments.

Matrix size	Lang.	sec.
81×81	Cilk	0.017
	VCilk	0.008
324×324	Cilk	0.13
	VCilk	0.13
648×648	Cilk	0.71
	VCilk	0.85
1296×1296	Cilk	4.8
	VCilk	6.2

operation	Vthread		pthread	ratio
	comp only	total		
mutex_lock	85	1050	50,000	48:1
mutex_unlock	85	610	45,000	74:1
cond_wait	85	850	60,000	71:1
cond_signal	90	650	60,000	92:1

7 Conclusion

We have shown an alternative to the thread model that enables easier-to-use parallelism constructs by splitting the scheduler open, to accept new parallelism constructs in the form of plugins. This gives the language control over assigning virtual processors to physical cores, for performance, debugging, and flexibility benefits. Parallelism constructs of programming languages can be implemented using sequential algorithms, within a matter of days, while maintaining low runtime overhead.

References

1. Agha, G., Mason, I., Smith, S., Talcott, C.: A foundation for actor computation. Journal of Functional Programming 7(01), 1–72 (1997)
2. Anderson, T.E., Bershad, B.N., Lazowska, E.D., Levy, H.M.: Scheduler activations: effective kernel support for the user-level management of parallelism. ACM Trans. Comput. Syst. 10, 53–79 (1992)
3. Carribault, P., Pérache, M., Jourdren, H.: Enabling Low-Overhead Hybrid MPI/OpenMP Parallelism with MPC. In: Sato, M., Hanawa, T., Müller, M.S., Chapman, B.M., de Supinski, B.R. (eds.) IWOMP 2010. LNCS, vol. 6132, pp. 1–14. Springer, Heidelberg (2010)
4. Christian, F.: Probabilistic clock synchronization. Distributed Computing 3, 146–158 (1989)
5. Intel Corp. CnC homepage, http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc/
6. Fluet, M., Rainey, M., Reppy, J., Shaw, A., Xiao, Y.: Manticore: a heterogeneous parallel language. In: Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming, DAMP 2007, pp. 37–44 (2007)
7. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the cilk-5 multithreaded language. In: PLDI 1998: Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation, Montreal, Quebec, pp. 212–223 (June 1998)
8. Gelernter, D.: Generative communication in Linda. ACM Transactions on Programming Languages and Systems (TOPLAS) 7(1), 80–112 (1985)
9. Kronos Group. Open Compute Language homepage, http://www.khronos.org/opencl
10. Cilk group at MIT. CILK homepage, http://supertech.csail.mit.edu/cilk/
11. Halle, S.: VMS home page, http://vmsexemodel.sourceforge.net
12. Hewitt, C.: Actor model of computation (2010), http://arxiv.org/abs/1008.1459
13. Hoare, C.A.R.: Communicating sequential processes. Communications of the ACM 21(8), 666–677 (1978)
14. Kahn, G.: The semantics of a simple language for parallel programming. In: Rosenfeld, J.L. (ed.) Information Processing, Stockholm, Sweden, pp. 471–475. North Holland, Amsterdam (1974)
15. Knobe, K.: Ease of use with concurrent collections (CnC). In: HOTPAR 2009: USENIX Workshop on Hot Topics in Parallelism (March 2009)
16. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21, 558–565 (1978)
17. Milner, R.: Communicating and Mobile Systems: The pi-Calculus. Cambridge University Press (1999)
18. OpenMP organization. OpenMP home page, http://www.openmp.org

Dynamic Task Parallelism with a GPU Work-Stealing Runtime System

Sanjay Chatterjee, Max Grossman, Alina Sbîrlea, and Vivek Sarkar

Department of Computer Science, Rice University, Houston, Texas, USA
{cs20,jmg3,alina,vsarkar}@rice.edu

Abstract. NVIDIA's Compute Unified Device Architecture (CUDA) enabled GPUs become accessible to mainstream programming. Abundance of simple computational cores and high memory bandwidth make GPUs ideal candidates for data parallel applications. However, its potential for executing applications that combine task and data parallelism has not been explored in detail. CUDA does not provide a viable interface for creating dynamic tasks and handling load balancing issues. Any support for such has to be orchestrated entirely by the CUDA programmer today.

In this work, we introduce a *finish-async* style API to GPU device programming as first step towards task parallelism. We present the design and implementation details of our new intra-device inter-SM work-stealing runtime system. We compare performance results using our runtime to direct execution on the device as well as past work on GPU runtimes. Finally, we show how this runtime can be targeted by extensions to the high-level CnC-CUDA programming model.

Keywords: gpu, work-stealing, finish, async, task, runtime.

1 Introduction

Graphics Processing Units (GPUs) contain hundreds of lightweight cores with large bandwidth access to on-chip memory. The massive parallelism and memory bandwidth of a GPU makes it very useful for computationally heavy applications operating on large data sets. Add to this the relative energy efficiency and low cost of its hundreds of simple cores compared to a CPU, with fewer cores, and we begin to understand its growing applicability both on the supercomputer scale [1] and in desktop computing for application areas that include life sciences, medical imaging and finance [2].

NVIDIA provides a C/C++ based API for programming their GPUs called the Compute Unified Device Architecture (CUDA) programming model. CUDA includes explicit memory management functions and generally requires considerable knowledge and understanding of the GPU hardware to achieve the significant performance gains that GPUs are capable of delivering, representing a steep learning curve to many programmers. The CUDA programming model was developed to primarily benefit regular data parallel applications, that are aligned to the needs of heavy graphics processing. In a nutshell, CUDA allows the programmer to launch large batches of SIMD threads. These collections of threads are decomposed into blocks of threads, containing anywhere from 32 to 1024 threads. The threads within a block are all mapped to the same streaming multiprocessor on the device. Threads on the same streaming multiprocessor (SM), and therefore in the same block, execute the same kernel in lock-step, but threads in different SMs may run completely separate kernels with no performance penalty.

Executing irregular applications that may involve dynamic task parallelism is not trivial using the current CUDA API. With this work, we aim to help improve the adoption of CUDA-like models for irregular applications by handling many of the lower level runtime details for the programmer. We have designed and implemented a runtime abstraction for dynamic task parallelism using the *finish-async* style API [3] on top of the regular data parallel model of execution. The programmer is freed from the responsibility of load balancing dynamically created tasks with the aid of our CUDA work stealing scheduler that operates across multiple SMs in the same device. The runtime helps reduce data transfer overhead by overlapping it with kernel execution, manage multiple devices, and distribute tasks on the device with the goal of balancing the workload across all SMs on a GPU.

The rest of the paper is organized as follows. Section 2 introduces our new GPU runtime for dynamic task parallelism. Section 3 provides implementation details of the work-stealing runtime deployed using the CUDA programming model. Section 4 shows how the runtime can be targeted by extensions to the high-level CnC-CUDA programming model introduced in past work [4]. Section 5 presents the results of experiments on our runtime system. Finally, Section 6 discusses related work and Section 7 contains our conclusions.

2 The GPU Work-Stealing Runtime System

GPUs' restriction to primarily data parallel applications means its potential for executing irregular applications that combine dynamic task parallelism with data parallelism has not yet been explored in detail. With CUDA currently providing no viable interface for dynamically creating tasks or handling load balancing issues, it may be some time before any official support is provided, if ever. In applications that perform recursive decomposition (say), each step produces tasks that may execute in parallel. This requirement for dynamic task creation is not trivially solved using the current CUDA API. Another factor that inhibits execution of irregular applications on the GPU is the need for task synchronization. CUDA allows synchronization only among threads that belong to the same block, which can run on only one SM. As a result, the only way that parallel blocks of threads can synchronize with each other in CUDA is via multiple kernel launches. This enforces a severe restriction on running irregular applications, that may require combining results from parallel work in each step before moving onto the next step.

Both of these disadvantages, dynamic task creation and parallel task synchronization, can be addressed by the *finish-async* style of programming [3]. In this model, an *async* creates or spawns a task that can potentially execute in parallel with the continuation of the spawning task. The *finish* provides a scope for all spawned tasks, both direct and nested, to complete before execution of the continuation of a finish. The *finish* and *async* constructs provide a natural way for programmers to express task parallelism using dynamic task creation and synchronization.

In this work, we have developed a runtime system on the GPU that provides a task-based implementation for abstractions such as finish and async. The goal of our *finish-async* model on the GPU is to be faster than the current divergent execution model for irregular applications without sacrificing the performance of intra-SM regular computations. We aim to provide the users with a simpler programming model for task parallelism, while handling the problem of load balancing which all systems supporting dynamic task creation must deal with. At the moment, the finish-async functionality on the device has only been thoroughly tested with a flat finish wrapping all device asyncs.

3 Implementation Details of GPU Runtime

The goal of this runtime is to balance work across the threads of a GPU better and with less effort for the user than a hand written application, while supplying a simpler and easier to use API. To achieve this goal we use a hybrid task distribution model that uses both work stealing and work sharing queues to provide load balancing between SMs on a CUDA device, and across different devices.

Our runtime starts by launching N blocks of CUDA threads on each device, where N is the number of SMs on that device. Conceptually, our runtime treats these blocks as worker blocks, analogous to worker threads in CPU-based work-stealing runtimes. Each of these worker blocks executes the runtime kernel. Shown in Figure 1, each worker block maintains its own work stealing deque. A worker can also steal from other workers' deques that reside on the same device. A separate FIFO shared queue is maintained to place tasks from the host onto the device. Only the host can push tasks onto this queue while the workers on the device compete to pop these tasks. At the moment, tasks are distributed evenly and naively among devices from the host, but more intelligent device selection for task placement would be an interesting direction for future work.

3.1 Task Representation

As mentioned earlier, CUDA has two levels of parallelism: SIMD threads within a block of threads on the same SM, and threads executing potentially different kernels on different SMs. When we talk about tasks in this paper, these represent tasks which are run by a block of threads, not individual threads, though tasks can be created by any thread in a block. To dynamically create tasks at the fine

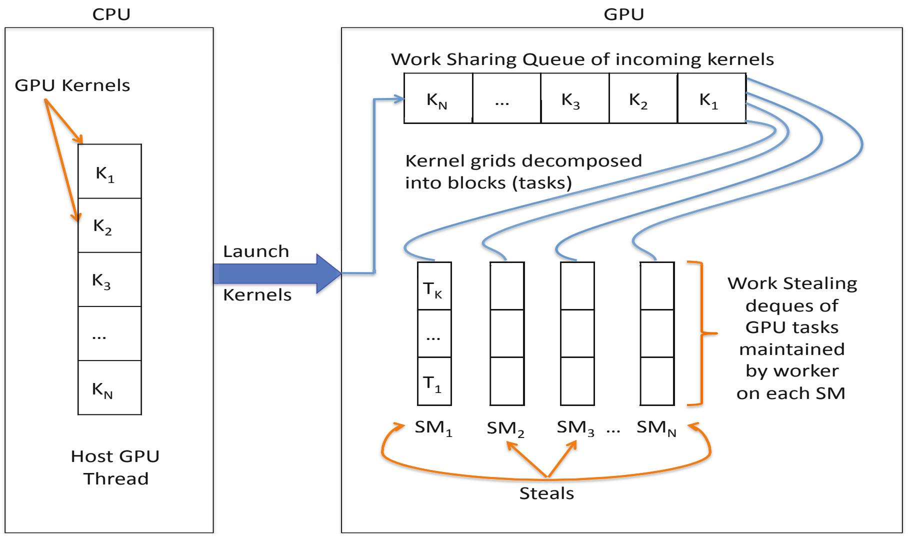

CPU
GPU Kernels
K_1
K_2
K_3
...
K_N
Host GPU Thread
Launch Kernels
GPU
Work Sharing Queue of incoming kernels
K_N ... K_3 K_2 K_1
Kernel grids decomposed into blocks (tasks)
T_K
...
T_1
Work Stealing deques of GPU tasks maintained by worker on each SM
SM_1 SM_2 SM_3 ... SM_N
Steals

Fig. 1. GPU runtime for dynamic task parallelism

grain level of individual threads would be prohibitively expensive for GPUs, in most cases leading to major divergence and serialization of execution.

Tasks designated for execution on the device are represented by a task structure. This task structure uses a param structure to represent inputs and outputs of the device task. The work sharing and stealing deques contain pointers to these task structures, shown below.

```
typedef struct {
        void *ptr; // address of the data
        size_t size; // number of bytes in this parameter
        volatile int *done_flag; // Indicates readiness of data
    unsigned char type; // Indicates input and/or output from device
} param;

typedef struct task_ {
        int type; // what code to run for this task
        param *p; // list of parameters to this task
        int *ready_flag; // indicates if this task has completed
    int num_params; // number of parameters
    ...
} task;
```

The task structure represents a task on the device with certain global inputs and outputs. The type field of a task structure uses an integer id to identify the code to be executed by this task, while the param pointer points to a list of parameters containing information on the inputs and outputs to the device. Tagging a parameter as being input or output only has significance in context of the device, and does not describe its relation to a task. The ready_flag field is used to test when a continuation task is ready to be executed in our async-finish model for on-device dynamic task creation, which we will discuss in more depth later in this section.

3.2 Task Creation

A task is launched by building a task structure with the appropriate parameters and type on the host or on the device and then placing it on the work sharing queue (if launching from the host) or on the worker block's work stealing queue (if launching from the device). If launched from the host, this process includes asynchronously allocating and copying the input and output data.

Once this task is placed in the appropriate queue it will then be fetched by a worker block. The worker block will then call the appropriate function based on the type field of the task. All queues on the device use locks to protect against concurrent conflicting accesses using CUDA's atomic CAS instruction.

A large obstacle to launching tasks from the device lies in CUDA's lack of dynamic memory allocation. Without dynamic memory allocation and with a potentially dynamic number of tasks it is impossible to estimate the amount of device memory that will be necessary to preallocate for each application. At the moment, this problem is being solved by a mock memory manager for the device, implemented as a linked list of preallocated task structures for each type of task on each device. Each of these empty tasks is allocated with memory for its parameters. When a device on the task launches a nested task it simply allocates a task from one of these linked lists, which can then be pushed onto the work stealing queues in the same manner as any task. When a task has completed on the device, it can be freed for future use by placing it back onto these linked lists.

Once all tasks have been placed on the device, the worker blocks are told that there is no remaining work by placing a special value into the work sharing queue. Upon finding this value, each worker block can be sure that there are no incoming tasks from the host to be run and instead begins waiting for a global counter of tasks to reach zero, while continuing to attempt to pop from its own work queue and steal from others' work queues.

3.3 Communication between Device and Host

One of the largest burdens placed on a CUDA programmer trying to achieve optimal execution on the device is device memory management. While the most basic memory management functions are easy to work with (cudaMalloc and cudaMemcpy being analogous to malloc and memcpy on the host), they also generally result in inefficient executions with lots of blocking function calls. Therefore, managing device memory for the user is a problem that any CUDA runtime system must solve. Ours hides all device allocation and communication from the user, instead using the contents of each task structure to know what the memory requirements of a task are.

One of the keys to good performance in any multicore system is overlapping communication with computation in order to hide the added overhead that isn't a concern in sequential code. In this runtime system, we leverage CUDA streams in an attempt to manage this overlapping for the user. Each time a new task is placed onto the device from the host, the associated input is copied with it. Using asynchronous memory copies in a CUDA stream allows this communication to occur in parallel with the runtime kernel and any programmer-written code executing on the device. In the future, a more advanced task pushing mechanism on the host could also allow copying through multiple streams at once to ensure maximum utilization of the bandwidth to each device.

In this implementation, a list of address mappings conceptually sits between the host and device memory. These mappings allow the runtime on the host to keep track of what host memory locations have already been copied to the device, what device location they were copied to, and how many bytes were transferred. This information allows us to be certain of where to copy to/from and how much to copy.

4 Extensions to CnC-CUDA

The GPU work-stealing runtime is a standalone tool which can be integrated with a programming model in order to provide a friendly user-interface. We made the integration with a new C-based implementation of the Concurrent Collections (CnC) programming model, being motivated by the results in previous work on CnC-CUDA [4]. The details of CnC-C are beyond the scope of this paper. Current work is being done on the integration with the Habanero C language [15] which uses the async-finish model, making integration more straight forward. On the other hand the CnC model is more general than the async-finish model; that is to say more graphs/programs can be expressed using CnC than with finish-async. In our work we will be using a subset of CnC's synchronization pattern. We reserve for future work extending the current implementation to support data dependences between CPU and GPU tasks.

CnC offers a easy way for the programmer to specify the dependences within his program with the aid of an intuitive graph language. The main components of any graph are data collections, control collections and steps. Data collections can be viewed as a tagged data storage (analogous to key-value pairs). A tag's role once it is put into a control collection is that of starting (prescribing) the steps assigned to it. Steps represent units of computation and are prescribed by a tag. They also read (get) items from data collections and can put items and/or tags into their data and control collections. A step may request an item with a certain tag from a data collection without having the knowledge whether the item exists or not. The CnC runtime will ensure the steps that have been prescribed will execute when the data they need is available.

Let us take one of the benchmarks - Crypt - and show the transformation of a CnC graph to its analogous CUDA code, assuming we already have a kernel written in CUDA for Crypt. First, we will write a simple CnC graph. The notation '::' in a CnC graph indicates the prescription of a computation step with a tag. The notation '− >' in a CnC graph indicates a computation step which either consumes or produces an item. For example, in the CnC graph below representing the crypt application, decrypt_tag is a tag collection which prescribes (launches) the computation step gpu_decrypt. The data collections used are "original" the original text, "z" the encryption key, "crypt" the encrypted text, "dk" the decryption key and "original_decrypted" the original text after decryption.The gpu_encrypt computation step consumes items from data collections original and z, and produces items into crypt and decrypt_tag output collection, where decrypt_tag is a control collection.

```
<decrypt_tag>::gpu_decrypt;
<encrypt_tag>::gpu_encrypt;
[ original ], [ z ] -> { gpu_encrypt } -> [ crypt ], < decrypt_tag >;
[ crypt ], [ dk ] -> { gpu_decrypt } -> [ original_decrypted ];
```

Using features offered by CnC much of the code needed to link the user's inputs with the kernel will be auto-generated.

A CnC program would then be written to work with the code generated by this graph file. The C code will look as follows:

```
CnCGraph graph;
graph.original.Put(tag, orig);
graph.z.Put(tag, z);
graph.encrypt_tag.Put(tag);
```

The CnC-C runtime will then manage the data dependencies, control dependences, and computation step invocation using the Habanero-C parallel programming language. Integrating the GPU work-stealing runtime with the CnC-C programming model would allow computation analogous to the below CUDA code to be generated for the user from the CnC graph specified above:

```
cudaMalloc(&d_original); cudaMalloc(&d_crypt);
cudaMalloc(&d_original_decrypted);
cudaMalloc(&d_z); cudaMalloc(&d_dk);
cudaMemcpy(d_original, original);
cudaMemcpy(d_z, z); cudaMemcpy(d_dk, dk);
encrypt_kernel<<<blocks_per_grid,threads_per_block,0,stream>>>(d_original,
    d_z, d_crypt);
decrypt_kernel<<<blocks_per_grid,threads_per_block,0,stream>>>(d_crypt,
    d_dk, d_original_decrypted);
cudaMemcpy(original_decrypted, d_original_decrypted);
```

The eventual goal is to auto-generate all code related to the CUDA runtime for the CnC user, only requiring a) an initialization call in the CnC Main function, b) a terminating call to signal the runtime kernel on the device to exit, and c) CUDA kernels to be used as computation steps. Reaching complete auto-generation is still a work in progress, but a manual proof of concept has already successfully demonstrated the integration of the GPU runtime and CnC.

5 Experiments

To test the performance of our GPU work stealing runtime, we tried to find examples of applications that are challenging to implement efficiently on graphics hardware and data parallel applications that are already well suited for CUDA. We use *n-queens* from BOTS [5] for our first benchmark. The BOTS implementation of nqueens on a CUDA device is difficult because it can result in unbalanced computation trees and requires a lot of dynamic task creation and load balancing. For our second benchmark, we will use an implementation of the *quicksort* sorting algorithm based on [6]. Our third benchmark, the *crypt* benchmark from the Java Grande Forum Benchmark Suite [7], is regular and recognized to be a good candidate for GPU execution. Our fourth benchmark will be a shortest path computation based on the implementation of *Dijkstra's* algorithm in [8]. This benchmark starts with a single task and must then spread the load across all SMs as evenly as possible. The fifth benchmark is an implementation of an unbalanced tree search based on code from OSU, which again tests the load balancing capabilities of the runtime. Our final benchmark will be the Series benchmark from the Java Grande Forum Benchmark Suite, another data parallel benchmark which demonstrates the low overhead of our runtime system.

In our tests, we compare runs with different numbers of devices as well as different data sizes to see how this impacts execution time. Additionally, we use diagnostic data from our runtime to measure how effectively we are load balancing the application's work.

Benchmark tests were performed with 1, 2, or 3 NVIDIA Tesla C2050 GPUs. The Tesla GPUs were tested with direct calls to runtime functions from C or CUDA code. Each Tesla C2050 has 14 multiprocessors, 2.8 GB of global memory, 1.15 GHz clock cycle and are using CUDA Driver and Runtime version 3.20. The host machine of the Tesla GPUs has 4 Quad Core AMD CPUs (2.5 GHz).

5.1 N-Queens

For the n-queens benchmark we ported the BOTS implementation of n-queens to the device using our runtime. The BOTS benchmark suite is intentionally designed to test the effect that irregular parallelism has on a multicore system. Irregular and recursive based parallelism results in less predictable numbers and distributions of tasks that are more difficult to allocate between worker threads. Because of this, n-queens is a challenge to port to the CUDA programming model without a loss in performance. Our runtime facilitates this irregular parallelism on graphics hardware. In Figure 2 we can see that our implementation of the n-queens benchmark scales well across multiple devices. From experience in developing it, we can also say that building an n-queens benchmark for the CUDA was much simpler with our runtime than it would have been without.

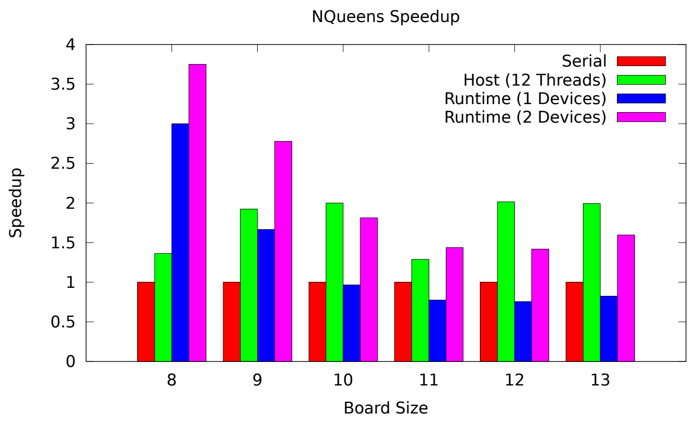


Fig. 2. Speedup normalized to single-threaded execution of the n-queens benchmark using our work stealing GPU runtime on 1 or 2 devices, or using 12 threads on the host

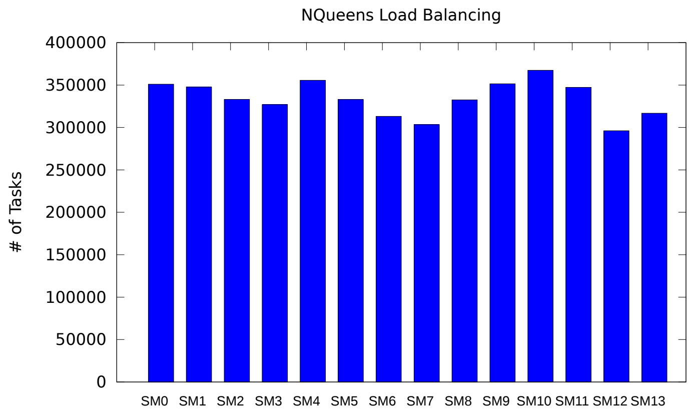


Fig. 3. Tasks executed by each SM on a single device running the n-queens benchmark

5.2 Crypt

The crypt benchmark from the Java Grande Forum Benchmark Suite (JGF) performs the IDEA cryptographic algorithm on a block of bytes, encrypting and then decrypting and validating the results. This algorithm is already well suited for execution in a data parallel programming model like CUDA. The encryption and decryption of every 8 bytes can be run independent of the rest of the data set with no conflicting accesses to shared variables. We include crypt in these experiments to demonstrate that using this runtime to run an application which is already well suited for CUDA will not result in significant degradation of performance.

The hand coded CUDA version of crypt which we tested against does not try to take advantage of any overlapping of communication and computation. The reason for this is that in our initial implementation we did not predict any advantages in splitting the copying of the original data to the device. Later tests showed that better performance was achievable with hand coded CUDA code, but required considerable more experience and effort on the part of the CUDA programmer.

Though not shown in Figure 4 an unexpected benefit of our runtime automatically providing device management for more than one device was the ability to handle larger data sets when using our runtime.

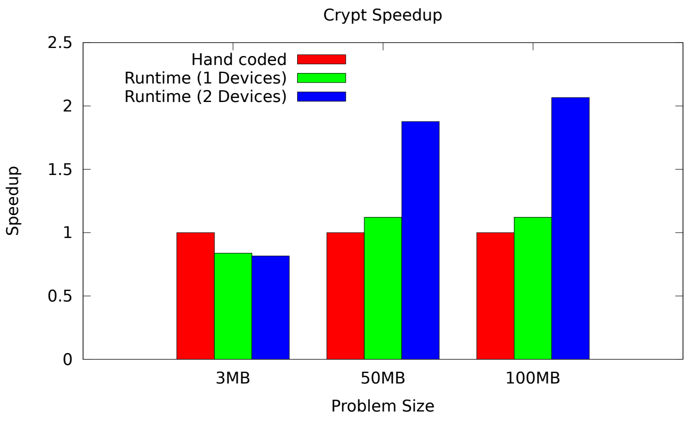


Fig. 4. Speedup of Crypt benchmark using our work stealing GPU runtime on 1 or 2 devices and using hand coded CUDA on a single device. Speedup is normalized to single device execution.

5.3 Dijkstra's Shortest Path Algorithm

We implemented Dijkstra's shortest path algorithm using our runtime based on the algorithm used in "Dynamic Work Scheduling for GPU Systems" [8] for the same reason as they did: it is an application which effectively tests the load balancing abilities of a runtime. Figure 5 shows how many tasks each worker executes while finding the distance from each node to a destination node in a 10,000 node bidirectional weighted graph. Initially a single task is placed on a single worker. From there, our runtime is able to distribute tasks to all SMs on the device, indicated by the level top surface of Figure 5.

5.4 Unbalanced Tree Search

Unbalanced Tree Search (UTS) makes a pass over a tree with a randomized number of children at each node. Because of the imbalance in the tree, static

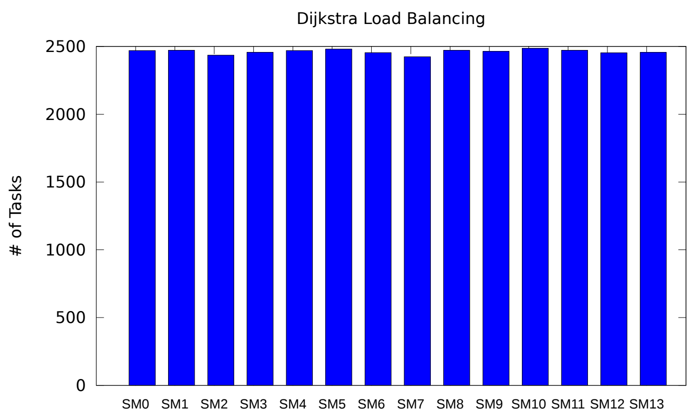


Fig. 5. Tasks executed by each SM on a single device running the Dijkstra benchmark

work assignment is very detrimental to performance. We implemented UTS using our GPU runtime, based on a multi-threaded implementation from OSU. Figure 7 shows that, in general, our implementation was able to maintain performance parity with a 12 threaded host system.

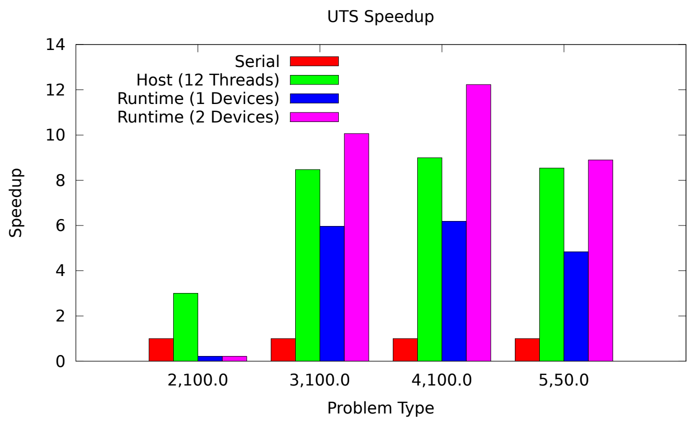


Fig. 6. Speedup of the UTS benchmark using our work stealing GPU runtime on 1, 2, or 3 devices, using 12 threads on a 12 core host system, and running in single threaded mode. Speedup is normalized to the single threaded implementation.

5.5 Series

The Series benchmark from the Java Grande Forum Benchmark Suite is extremely data parallel, and well optimized to run on the GPU. An interesting result of this test is that our runtime with one device severely underperforms compared to the hand coded implementation. The cause of this is the small input data sizes to series. Because there is very little input to the series benchmark, the small copies necessary to launch tasks on the device add enough overhead to degrade performance. This introduces a small subset of applications which our runtime may not perform as well on. However, with 2 devices we outstrip single device hand-coded without any additional effort on behalf of the programmer.

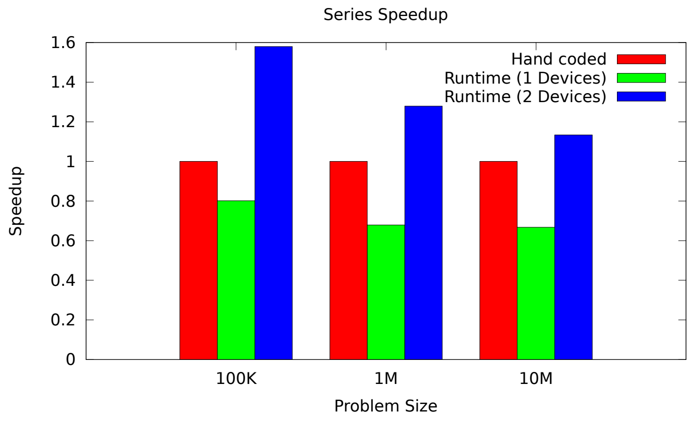


Fig. 7. Speedup of the Series benchmark using our work stealing GPU runtime on 1 or 2 devices, compared against hand-coded CUDA on 1 device. Speedup is normalized to the single device implementation.

5.6 Multi-GPU Performance

For many of the above benchmarks, we see less than expected acceleration from using multiple devices, a counterintuitive result which requires some explanation. The primary cause of this is redundant memory copies. Take the crypt benchmark for example. In crypt, there are encryption and decryption keys being used which must be accessible from each device. This means that by increasing the number of devices, you are actually increasing the amount of necessary communication that is necessary. Other factors may be causing this discrepancy as well, but further investigation would be necessary to understand what they may be. Two potential solutions to this problem could be: 1) more intelligent task placement of tasks which share data on the same device, or 2) use CUDA 4.0's unified virtual device address space to access a single copy from any device. The usefulness of more intelligent task placement when benchmarking would be minimal as it would mean even when 3 devices are initialized not all would be necessarily used. Additionally, while early experience with the unified virtual address space shows that it is more useful in the easing porting, its use can actually result in performance degradation.

6 Related Work

There have been some recent experiments at either including GPU execution into current programming models and languages, or implementing a task-parallel runtime on the GPU. Lastras-Montano et al. [8] implemented a work stealing runtime on the device. In their paper, a worker is a single warp of threads (32 threads). Each of these warps is assigned a q-structure, which is a collection of queues in shared and global memory to place tasks in and steal tasks from. Since multiple warps in a block always reside on the same SM, this runtime can attempt to steal from those other warps via faster on-chip shared memory before looking to steal from global memory (i.e. from workers in other SMs). Their use of queues in shared memory would yield considerably less latency than our global memory queues, and could be included in future work. However, designating each individual warp as a worker would lead to increased contention for steals. While our runtime is designed for continuous use throughout an application, their runtime starts with a kernel launch and ends when a certain number of steals have failed. This termination condition could be harmful to performance critical applications.

The X10 programming language recently began supporting the execution of tasks on CUDA devices [9]. They do not provide an actual on-GPU work stealing runtime, but instead integrate GPU tasks into their host work stealing runtime. They provide the user with a simpler API for allocating device memory and copying asynchronously from host memory to device memory than the CUDA API does and expose the block and thread CUDA memory model to the programmer in what might be a more intuitive way: as nested loops iterating over X10 points. However, this is not a device runtime and even though the appearance of the code may be more familiar to non-CUDA programmers and they hide some of the memory management from the programmer, the programmer must still be very aware of the CUDA programming model and its challenges and nuances.

The work in [10] presented a variety of potential GPU load balancing schemes ranging in complexity from a static array of tasks to a work-stealing task distribution technique similar to the one used in our runtime system. In order to compare the relative performance of the different systems proposed in their paper they used an octree creation application. They demonstrated that the task distribution system most similar to our own methods for distributing tasks between SMs on the same device achieved the best performance of those tested.

StarPU [11] [12] is another runtime system for hybrid CPU and GPU execution. Similar to X10, this system's role is to dispatch tasks to different processing unit for which it makes complex scheduling decisions based on different hardware. StarSs and its GPU extension GPUSs [13] is building on the OpenMP model and offers simplicity by using pragmas to define tasks that can be executed on the GPU. However each task annotated for GPU execution will run as an independent kernel without any control on how the computation is distributed on the device. Both StarPU and StarSs will use the CPU, GPU as well as other resources like the Cell to achieve system-wide load balancing, but neither of them addresses the problem of load balancing inside the GPU, but base their assumption on the fact that work inside the GPU kernel will be uniform.

This work has the potential of being supported on OpenCL [14]. In such a scenario, slight modifications of the runtime API will needed to be done for conformance to OpenCL standards, and possibly a reimplementation of the runtime kernel.

7 Conclusions and Future Work

In this paper, we have presented a GPU work stealing runtime that support dynamic task parallelism at thread block granularity. We demonstrated the effectiveness of our combination of work stealing and work sharing queues in distributing tasks across the device using the nqueens and Dijkstra benchmarks. Each of these benchmarks starts with a single task on a single SM, requiring low overhead task distribution to achieve good performance. We demonstrated that even applications for which CUDA is well suited using this runtime incurs little overhead and may even result in better performance, a result of automatically managing multiple devices for the user as well as overlapping data transfer with kernel execution. We gave a brief overview of other simplified interfaces to the device that are currently available or in development and compared them to our own approach. While support for CUDA calls in X10 provide simpler access to the device and the previous work by Angels et al provided fine grain load balancing at the level of a warp of CUDA threads, our runtime demonstrates parts of both of these features with a persistent state on the device that supports more of a streaming and data driven programming model than the launch-wait-relaunch model normally used with CUDA.

Some topics for future work are as follows. At the moment, our runtime handles device memory allocation and transfer for the programmer, but freeing of device memory cannot occur while a kernel is running on the device, and therefore cannot happen while our runtime is being used. Therefore, in order to limit waste of device memory and of page locked host memory our runtime system should include some more advanced memory reuse mechanisms. This may include the implementation of a concurrent memory manager on the device.Some optimization may be possible on our device work stealing code, with a focus on minimizing the use of atomic instructions and memory fences. While these are necessary to ensure each worker has a consistent view of other workers' deques we may be over-using these instructions. We would also like to consider the results of decreasing the number of threads in each worker on the device. Angels et al. used warps as task execution units. Investigating the effect that a change in worker granularity would have on our system could be very beneficial (or damaging) to overall performance. Finally, we also aim to integrate this work into the larger Habanero-C parallel programming language project [15] at Rice University. The plan is to create an unified runtime for integrated CPU and GPU scheduling of tasks. This work will enable a system to automatically decide at runtime when it is more beneficial to run a task on the GPU instead of the CPU and vice versa.

References

1. National Supercomputing Center in Tianjin, Tianhe-1A (November 2010), http://www.top500.org/system/details/10587
2. Nvidia, CUDA (2011), http://developer.nvidia.com/cuda-action-research-apps
3. Charles, P., et al.: X10: an object-oriented approach to non-uniform cluster computing. In: OOPSLA, NY, USA, pp. 519–538 (2005)
4. Grossman, M., Simion Sbîrlea, A., Budimlić, Z., Sarkar, V.: CnC-CUDA: Declarative Programming for GPUs. In: Cooper, K., Mellor-Crummey, J., Sarkar, V. (eds.) LCPC 2010. LNCS, vol. 6548, pp. 230–245. Springer, Heidelberg (2011)
5. Duran, A., et al.: Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In: ICPP 2009, pp. 124–131 (2009)
6. Cederman, D., Tsigas, P.: GPU-quicksort: A practical quicksort algorithm for graphics processors. J. Exp. Algorithmics 14 (January 2010)
7. The Java Grande Forum benchmark suite, http://www.epcc.ed.ac.uk/javagrande/javag.html
8. Lastras-Montano, M.A., et al.: Dynamic work scheduling for GPU systems. In: International Workshop of GPUs and Scientific Applications, GPUScA 2010 (2010)
9. X10 2.1 CUDA, http://x10.codehaus.org/x10+2.1+cuda
10. Cederman, D., Tsigas, P.: On sorting and load balancing on GPUs. SIGARCH Comput. Archit. News 36 (June 2009)
11. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.-A.: StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 863–874. Springer, Heidelberg (2009)
12. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. Concurr. Comput.: Pract. Exper. 23, 187–198 (2011)
13. Ayguadé, E., Badia, R.M., Igual, F.D., Labarta, J., Mayo, R., Quintana-Ortí, E.S.: An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 851–862. Springer, Heidelberg (2009)
14. Opencl 1.1, http://www.khronos.org/opencl/
15. Habanero-C, https://wiki.rice.edu/confluence/display/habanero/habanero-c

A Code Merging Optimization Technique for GPU

Ryan Taylor and Xiaoming Li

ECE Department, University of Delaware, USA

Abstract. A GPU usually delivers the highest performance when it is fully utilized, that is, programs running on it are taking full advantage of all the GPU resources. Two main types of resources on the GPU are the compute engine, i.e., the ALU units, and the data mover, i.e., the memory units. This means that an ideal program will keep both the ALU units and the memory units busy for the duration of the runtime. The vast majority of GPU applications, however, either utilize ALU units but leave memory units idle, which is called ALU bound, or use the memory units but idle ALUs, which is called memory bound, and rarely attempt to take full advantage of both at the same time.

In this paper, we propose a novel code transformation technique at a coarse grain level to increase GPU utilization for both NVIDIA and AMD GPUs. Our technique merges code from heuristically selected GPU kernels to increase performance by improving overall GPU utilization and lowering API overhead. We look at the resource usage of the kernels and make a decision to merge kernels based on several key metrics such as ALU packing percentage, ALU busy percentage, Fetch busy percentages, Write busy percentages and local memory busy percentages. In particular, this technique is applied at source level and does not interfere with or exclude kernel code or memory hierarchy optimizations, which can still be applied to the merged kernel. Notably, the proposed transformation is not an attempt to replace concurrent kernel execution, where different kernels can be context-switched from one to another but never really run on the same core at the same time. Instead, our transformation allows for merged kernels to mix and run the instructions from multiple kernels in a really concurrent way. We provide several examples of inter-process merging describing both the advantages and limitations. Our results show that substantial speedup can be gained by merging kernels across processes compared to running those processes sequentially. For AMD's Radeon 5870 we obtained an average speedup of 1.28 and a maximum speedup of 1.53 and for NVIDIA's GTX280 we obtained an average speedup of 1.17 with a maximum speedup of 1.37.

1 Introduction

Today's GPUs consist of hundreds of processing cores and can run thousands of threads at the same time. Even though the sheer scale of parallelism may already generate good speedups for GPU programs, crucial resources on GPUs such as ALU processing units and memory units are frequently imbalancedly utilized, which prevents the realization of full GPU performance. The reason is that even though many threads can run on the GPU concurrently, the threads are identical copies of the same code. If a batch of threads saturate the ALU units but leave the memory units idling, the imbalance and under-utilization won't be remedied by context switching because the next batch of threads will use the resources in the same way. This observation is true for both NVIDIA's and AMD's GPUs. Clearly, GPUs can deliver better performance if the GPU resources are used in a more balance way.

Recently NVIDIA released a technology called concurrent kernel execution [1] that allows different kernels (i.e., different code) to be kept active on NVIDIA Fermi GPU and switches the execution between kernels when one kernel is stalled. While sounds like a perfect solution for resource idling, the concurrent kernel execution in fact does *not* improve the resource imbalance or resource under-utilization within a kernel because at any moment, only *one* kernel can run on a core of a GPU. While the resource idling due to long latency operations is averted by context switching, the imbalance of the kernels, utilization of the core resources is untouched. For example, assuming that we have two kernels, one only using ALU and one only doing memory operation. The concurrent-kernel-execution will switch back-and-forth between the two kernels because the memory operations are slow and will stall execution. Overall, the effective memory access latency is reduced. However, at any given moment, the GPU is either idling the ALU or idling the memory controller, because only one kernel runs at a time. In other words, the switching from an imbalanced kernel to another imbalanced kernel will not help either one to get more balanced. This sub-kernel level of resource imbalance and the consequent resource under-utilization is our key observation and is not addressed in current technology. We need a technique that works within a thread/kernel to help achieve full utilization of each of the given resource units in the GPU.

In this paper, we propose a novel code transformation technique that strategically merges kernels with different resource usage patterns to improve performance of GPU programs. This technique is applied at a coarse grain level. More specifically, this paper makes three contributions. The first contribution is the identification of a group of profiling metrics that defines how *fully* a resource is by a GPU kernel, i.e., the boundedness of resource utilization. The level of component use is a good metric for measuring the imbalance of resource utilization in a GPU program kernel and therefore provides guidelines of which kernels might be good candidates for merging.

The second contribution is the kernel merging transformation that provides multiple ways of merging GPU kernels to balance the resource usage on GPU. We also developed several heuristics to guide the transformation to maximize benefit. The third contribution is that the proposed transformation is, to our best knowledge, the first cross-platform code transformation technique that addresses the resource under-utilization problem for both AMD's and NVIDIA's GPGPU programming frameworks.

The rest of this paper is organized as follows. Section 2 introduces the architectural features of AMD and NVIDIA GPUs that are relevant to the code merging transformation. Section 3 explains from a GPU architecture point-of-view why code merging might improve program performance on GPU. Section 4 describes the transformation and heuristics of how and when to apply the transformation. Then, we comprehensively evaluate our approach on both AMD and NVIDIA GPUs and show the results in Section 5. Finally we conclude and suggest future directions of research in Section 6.

2 GPU Background

Modern GPUs are massively multi-threaded many-core architectures with several layers of memory in a complex memory hierarchy scheme. One key aspect in the evaluation of the performance of GPU programs is the bottleneck, which is the limiting factor of the application. There are essentially two types of bottlenecks on GPUs: ALU and memory. An ALU bound GPU kernel is a kernel in which the majority of the time executing the kernel is spent doing ALU operations. A memory bound GPU kernel is a kernel in which the majority of the time executing the kernel is spent doing memory operations. The GPU achieves the greatest utilization when both ALU and memory are fully used.

The two main brand name GPU architectures share similar features but are also quite different. The terminologies are also different on the two architectures even for features that work in similar ways. In the following we briefly introduce the architectural features on NVIDIA and AMD GPUs that are most relevant to the proposed code transformation. We will refer the detailed introduction of GPU programming on the two architectures to their respective programming guides.

2.1 AMD GPU

The AMD GPU consists of multiple compute units (SIMD engines) each of which has 16 stream cores (thread processors). Each stream core itself contains processing elements that make up a 5-wide VLIW processor. The memory hierarchy consists of special registers, global purpose registers, L1 and L2 cache, local memory, texture and vertex fetch and global memory. Threads are organized into work-items and then into wavefronts. For the high end AMD compute device, a wavefront consists of 64 threads organized into 16 2x2 quads, and wavefronts are then organized into work-groups. A compute device has two wavefront slots, odd and even, and executes two wavefronts simultaneously. More wavefronts can be in the work queue and can be switched with an executing wavefront when the executing wavefront stalls. The number of wavefronts that can exist in the queue is dependent on the number of resources available, such as registers and local memory. This is an important feature because this wavefront switching allows for greater GPU utilization and gives the GPU the ability to hide memory latencies. The instruction set architecture is organized into VLIW instructions or bundles. These bundles are then organized into clauses, which are a group of bundles of the same type. For example, ALU operations are grouped into an ALU clause while fetch operations might be grouped into a TEX clause. Each clause is executed by a wavefront until completion and cannot be switched out mid-clause.[2]

2.2 NVIDIA GPU

The NVIDIA GPU consists of multiple stream multiprocessors with 8 streaming processors per multiprocessor. The memory hierarchy consists of registers, shared memory, texture memory, constant memory and global memory. Threads are organized into groups of 32, called warps. Each warp is executed in SIMD fashion on a stream multiprocessor and is broken into half-warps, 16 threads, and quarter-warps, 8 threads.

Like the switching between clauses on AMD GPUs, NVIDIA GPUs also allow switching between warps when warps stall. Warps are organized into blocks which are then organized into grids. While NVIDIA exposes PTX (an intermediate representation of instructions) they do not currently expose their ISA directly. Since PTX is an intermediate representation and is not directly executed on GPU hardware, it is not able to give the same level of detail as an instruction set architecture.[1]

3 Motivation

The main motivation behind this code merging transformation is the idea to improve performance by balancing the utilization of GPU resources within kernels. In general, most current GPU applications are implemented with the goal to get maximum ALU utilization. While the ideal full ALU utilization may give the best performance for an application, some algorithms are naturally memory bound and simply don't allow their GPU implementations to become ALU bound. These memory bound GPU kernels leave many ALU resources idle. In the same notion, ALU bound GPU kernels leave many memory resources idle. We note that GPU is an intrinsically throughput-oriented architecture. Our proposed transformation attempts to achieve better overall GPU utilization by recognizing idleness, both ALU and memory bound, and combining GPU kernels that are on different ends of the ALU/memory usage spectrum so as to move the point of usage of the combined code closer to neutral which implies better utilization of GPU resources and better overall performance. Equally important is the criteria that determines which kernels exhibit good qualities of a merging candidate. This type of transformation would be particularly helpful for a multi-user GPU system, because the proposed transformation can improve the overall throughput by recouping idle resource capacity across different applications that run at the same time on such shared systems.

We want to address the relationship between the code merging transformation with other optimization techniques on the GPU. The proposed transformation is compatible with most other GPU optimizations. The transformation occurs at a coarser granularity than other known optimizations, most of which occur within the kernel itself such as memory and register optimizations[3][4] and divergence and workload balance optimizations[5][6]. This transformation does not interfere with these known optimizations because these optimizations can still be applied after the proposed transformation to the merged kernels. There also exist some optimizations which require the kernel code to be split into multiple kernels[7]. This transformation does not affect this type of optimization either since it can still be applied to the split kernels, if there are other kernels to merge with them. Furthermore, this transformation can also be applied with emerging GPU technology such as concurrent kernels [1] since several kernels can be merged into one or more kernels and can then be run concurrently under the same context.

The discussion of the level of use of resource utilization provides a high-level view of why the proposed code merging transformation works. Next we go into the design details of NVIDIA's and AMD's GPU to reveal an architectural explanation of how code merging, if done properly, leads to a better utilization of the ALU/memory units and improves program performance on GPU.

Furthermore, the second major motivation behind this transformation is to merge kernels from different processes on multi-user systems into one process. For example, if a developer is utilizing a system for executing an ALU intense algorithm and another developer is waiting to execute a memory intense algorithm, these two processes' kernels can be merged into one larger kernel resulting in better overall GPU utilization and kernel speedup. Thereby, this type of transformation would be particularly helpful for a multi-user system.

Next we discuss three architectural/program features that are not ideally handled in current technology and can be improved by the proposed transformations.

3.1 ALU Packing Percentage

Since NVIDIA GPUs do not utilize VLIW processors this metric applies only to AMD GPUs. The ALU packing percentage is the percent of cores in the VLIW processor that are being utilized by the GPU kernel. For example, if the ALU packing percentage is 20% then only 1 of the 5 cores are being used per VLIW instruction. There are two major factors that affect the ALU packing percentage. The first factor is that of data dependence. Instructions that have data dependence are not able to be scheduled together in one VLIW instruction. The second factor is that of control flow and scope. Instructions within a control flow statement or scope are scheduled, by the compiler, to run in separate ISA clauses. Instruction packing does not occur across clauses and so while there may not be any data dependence between an instruction outside an if statement and one inside an if statement, these two instructions cannot be packed. The ALU packing percentage can be increased through this transformation because separate GPU kernels have no data dependence and their instructions can be bundled together within VLIW instructions. Instructions within control flow statements can only be combined across GPU kernels if the control flow statements have the same conditionals or the instructions within the control flow statements can be software predicated or the two kernels have synchronization points which can be combined. Figure 1 and Figure 2 both have data dependence from instruction 13 to 14 in ALU clause 02 and are only utilizing 1 of the 5 cores in each VLIW instruction. If these two kernels are merged, the resulting code, as shown in Figure 3 now has 2 of the 5 cores in each VLIW instruction being utilized. Basically the merged code uses the same number of cycles as either of the pre-merge kernels, as if one is piggy-backing the other.

3.2 Idleness

There are three major GPU components which can execute in parallel: ALU components, global memory components and local memory components. Kernel code is generally laid out in a streaming fashion: read input, work on input and write output. This inherent layout causes dependence to occur between the components and while there can exist some overlap in resource usage due to context switching, there still leaves idleness in those components in which the kernel does not greatly use. For example, in an ALU bound kernel each wavefront uses the ALU units far more than the memory units and the time it takes to execute one wavefronts' ALU operations can be spent

```
02 ALU: ADDR(51) CNT(117)
13 w: ADD T0.w, T0.y, PV12.x
14 z: ADD T0.z, T0.x, PV13.w
```

Fig. 1. Kernel One ISA

```
02 ALU: ADDR(51) CNT(63)
13 y: ADD T0.y, T0.w, PV12.z
14 x: ADD T0.x, T0.z, PV13.y
```

Fig. 2. Kernel Two ISA

```
02 ALU: ADDR(56) CNT(121)
 13 x: ADD T0.x, PV12.w, R2.x
    z: ADD T0.z, R0.x, PV12.w
 14 y: ADD T0.y, T0.w, PV13.z
    w: ADD T0.w, T0.w, PV13.x
```

Fig. 3. Merged Kernels ISA

fetching inputs for multiple wavefronts from memory. This leaves a gap when executing the back half of the wavefronts such that all the inputs have been fetched for all the wavefronts but not all the ALU operations have been executed. In this case, there is an opportunity to use the memory units that are sitting idle. In contrast, in a memory bound kernel the memory units are used far more than the ALU operations and therefore, in the same sense, the ALU units are not being fed fast enough and are sitting idle. There also exists the case where local memory is being used but neither the ALU units nor the global memory units are being used. In both types of memory, global and local, stalling is considering as being busy since the unit is being tasked. Stalling is not the same as idling. In this paper, these are the three major cases that motivate this kernel merging technique.

3.3 API Overhead

Each process or application requires a certain amount of API overhead or setup time. This not only includes the individual kernel invocation time but also count the entire API overhead and setup time, such as the setting up of device and the creation of context. For individual processes each of these steps must be done at least once and while kernel invocation time can be reduced with a single application by queuing the kernels it cannot be reduced when looking at kernels across processes. When kernels are merged this overhead of setup is reduced since these calls only need to take place once. This overhead reduction scales with the number of merged kernels across programs since the device, context and platform setup times remains the same.

4 Transformation

Kernels from different processes are merged in two steps. The first step brings the two separate host codes into one program and the second step is to merge the kernel codes. The main work in the first step is to create a proper context for the kernel merging in the second step. The context preparation involves merging the data structures, functions and necessary code into one program. Furthermore, we also bring the two separate OpenCL

API calls under one ”umbrella” of OpenCL API calls, i.e., setting up the platform, device, context and kernel for OpenCL. In addition, all of the buffer creation, buffer reads, buffer writes and releasing objects need to be merged. The first step may appear to involve a lot of things, however, is actually very straightforward because it’s not important in which order these things get merged in context preparation, so long as they stay in the proper order within the code, i.e. creating buffers before writes and writes before kernel call and kernel call before reads and reads before releases. The performance of the merged kernel is not affected either as long as the context is prepared in a valid way.

Merging the kernel code in the second step is not as straightforward and the way in which the kernel code is merged can greatly impact performance. The simplest way to merge two kernels is to cascade the kernel codes as shown in Figure 4. However, simple solutions like cascading do not usually give optimal performance. Here we will discuss several key factors that are important to the decision of how to merge. Furthermore, we describe several heuristics for choosing which kernels to merge. The heuristics are very effective and give near optimal performance in most cases.

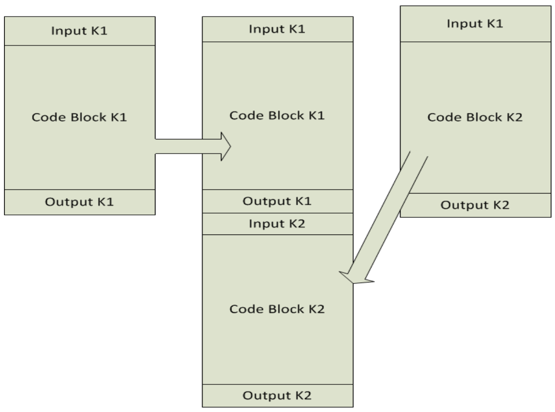


Fig. 4. Cascading Kernels

4.1 Merging Based on Resource Usage

Generally speaking, kernels that use resources in different ways can potentially benefit from merging. There are two values which impact this decision: actual resource busy percentages and theoretical resource busy percentages. Both of these values give an indication of the bottleneck, the profiler giving the real bottleneck and the theoretical values giving what ”should” be the bottleneck. The theoretical values can also be looked at as a best case scenario (components are totally parallelized, so the execution time is the greatest of the three components’ times) or a worst case scenario (components are totally serialized, so the execution time is the sum of the three components’ times). In reality, neither of the theoretical times are true since there is dependence and also some parallelization of work. However, these values can be used as a guide when compared

Table 1. AMD OpenCL Profiler Counters

Counters	Description
Time	Kernel execution time not including kernel setup
ALU Instr	Number of ALU instructions
Fetch Instr	Number of global read instructions
Write Instr	Number of global write instructions
LDS Fetch Instr	Number of LDS read instructions
LDS Write Instr	Number of LDS write instructions
ALU Busy	Percent of overall time ALU units are busy
Fetch Busy	Percent of overall time read units are busy (includes stalled)
Write Stalled	Percent of overall time Write units are stalled
LDS Bank Conflict	Percent of overall time LDS units are stalled by bank conflicts
Fast Path	Total KBs written using the fast path
Complete Path	Total KBs written using the complete path

to the profiler values as to how much serialization, parallelization or idleness is actually occurring in a kernel. In this paper, we use these two sets of values of kernels to decide the potential benefit for merging them.

For this transformation, the profiler counters used are described in Table 1. Some of the counters are used to calculate the actual run times of the components and some of the other counters are needed to calculate the theoretical values. Equations 1 and 2 refer to the equations to calculate the theoretical time spent on each component. Equations 3, 4, 5 and 6 are the equations to calculate the actual time spent on each component. The actual time spent on ALU and read operations is easily calculated since the profiler directly reports the percent time the kernel is busy performing these operations. The actual time spent using the LDS (local memory) and writing data to global memory is given as an estimation using some profiler counters together with the theoretical equations since the profiler does not report direct busy percents on these operations.

$$\text{Global Memory Time} = \frac{\text{Total Bits}}{\text{Bus Width} \times \text{Memory Clock}} \tag{1}$$

$$\text{LDS Time} = \frac{\text{\# Threads} \times \text{LDS Instr}}{\text{LDS per Clock} \times \text{Engine Clock}} \tag{2}$$

$$\text{Actual ALU Time} = \text{Kernel Time} \times \frac{\text{ALU Busy}}{100} \tag{3}$$

$$\text{Actual Read Time} = \text{Kernel Time} \times \frac{\text{Fetch Busy}}{100} \tag{4}$$

$$\text{Est. Actual Write Time} = (1) + \text{Kernel Time} \times \text{Write Stalled} \tag{5}$$

$$\text{Est. Actual LDS Time} = (2) + \text{Kernel Time} \times \text{LDS Conflict} \tag{6}$$

For example, given a kernel with 98% ALU Busy and 22% Fetch (read) Busy counters we can determine that there is 1) overlap in ALU workload and global memory read workload (since together they are greater than 100%) and 2) 78% of the time the memory read units are idle. Since this kernel is 98% ALU Busy this would indicate that this kernel would not be a good candidate for merging with another ALU bound kernel based on resource usage. This kernel is however only using the read units 22% of the time and therefore would make a good candidate for merging with a read bound kernel (since although adding ALU operations would increase the ALU execution time in an already ALU bound kernel, it would hide memory latency from the second kernel). A kernel with the opposite values, 98% Fetch Busy and 22% ALU Busy would be a good candidate for merging with an ALU bound kernel (since the first kernel could hide some ALU operations of the second kernel).

The resource usage of a kernel is represented as a vector of five values: an ALU value, a global read value, a global write value, an LDS value and an overall ”overlap” value. The vectors will be used as the decision factor for the purposes of choosing which kernels to merge and the way of merging. The first four values represent how much of the resource is being used by the kernel and the last value, the overlap value, represents how much of that work is being done in parallel by the components. The ALU value and global read value come directly from the counters ALU Busy and Fetch Busy, respectively. The global write value and LDS value are the percents of equation 5 and equation 6 in reference to the Time counter, respectively, since the profiler does not directly give the percent these units are busy. The overlap value is important because it gives detail as to how much resource availability within that time exist among the different components and how much room in each component is available for merging. In the above example, if the overlap is 0 then the ALU is idle during the entire fetch time and vice-versa. A kernel with a low overlap value makes for a better merge candidate than a kernel with a higher overlap value. If a kernel is 98% ALU busy and 50% fetch busy and has an overlap of 100% then there is only 50% of the total kernel time to do extra fetch instructions.

The individual actual times are calculated and summed and the percent difference is taken with regards to the Time counter. This calculation yields the overlap value equation below:

$$\text{Overlap} = 1 - \frac{\text{Kernel Time}}{(3)+(4)+(5)+(6)} \tag{7}$$

This equation is not a direct calculation for the actual overlap since it is not currently possible to directly extract the level of component parallelization from the profiler that has no such counter. Unfortunately, at this time, it is not possible to tell which components' runtimes overlap with which other components' runtimes, making it not possible to give an exact formula for estimating benefit based on resource usage. Instead, this heuristic is used to give a quantifiable estimation of the overall overlap.

Each kernel candidate's values, the five listed above, are summed together. For example, given two kernels with the above properties would result in an ALU value of .98 + .98 = 1.96 and a global read value of .22 + .22 = .44, so these would not be good merging candidates because both kernels are very ALU bound. For example, if we assume that these two kernels serialize the read and ALU operations (no overlap) then only a speedup of the time of one kernel's read operations can be hidden. However, given one kernel with the properties mentioned above and one with opposite properties the ALU value would be .98 + .22 = 1.20 and a global read value of .22 + .98 = 1.20, giving a much better balance and would hence be good merging candidates since the fetch bound kernel could hide memory latency in the ALu bound kernel. For example, if we again assume that these two kernels serialize the read and ALU operations then the speedup could be .98 - .22 = .68 of the second kernels' read time. What this means is that if the first kernel spends 22% of it's time reading data then 98% of it's time executing ALU operations, the second kernel can spend 68% of the first kernel's time reading data. Speedup depends on the level of overlap for each kernel, which in practice is neither 0% or 100%, even in the first example the ALU is not busy 100% of the time.

4.2 Merging Based on ALU Packing Percentage

Another attribute that contributes to the merging decision is the ALU packing percentages. This heuristic only applies to AMD's VLIW architecture and, unlike merging based on resource usage, would allow two ALU bound kernels to be merged and obtain speedup. When merging kernels, one kernel's ALU instructions might fit into the empty VLIW slots of the other kernel, thereby reducing the overall number of ALU cycles needed to execute the kernel. The compiler has control of how the VLIW instructions are packed and the size of the window the compiler uses when looking at which VLIW instructions to pack. The window for packing analysis is between the point in which two code blocks meet. In other words, the compiler's window for packing can only reach so far up and down and won't pack a merged kernels' instructions if they are outside of that window. Greater code motion and interleaving of code statements between the merged kernels could lead to better packing improvements but would, in most instances, not overcome the negative effects of the increased register usage. It is still possible to get speedup from this feature with those constraints. For example, given two ALU bound kernels with 500 ALU instructions and 440 instructions merged over 2048*2048 threads, a speedup of 1.13 is obtained on AMD's Radeon 5870. The decision to merge based on ALU packing percentage should be evaluated after the kernel candidates have failed the test based on resource usage. Specifically, the ALU packing heuristic is based on 1) the kernel's packing percentage (lower is better) and 2) the kernel's ALU usage (higher is better). One of the other advantages to ALU packing is that profile feedback is not needed, ALU packing can be determined statically through the analysis of the assembly code of the program.

4.3 General Techniques of Kernel Merging

When merging kernels there are some good practices that generally adhere to most scenarios. One key issue when merging kernels is the impact the extra code will have on register pressure. Register pressure is always a performance consideration when programming for the GPU and, generally, the lower register pressure the better since more threads can be queued which leads to more memory latency hiding. Ideally, since the kernels are independent pieces of code, there should be little to no added register pressure. The register count used in the merged code should be no more than the highest register count among all kernels. This is true when the kernels are simply cascaded because all the registers used by the top kernel can then be reused by the bottom kernel, so this technique does not significantly impact register pressure. There is no need to have both sets of registers reserved because registers can be reused from one kernel's code to the next when the kernels are merged. However, simply cascading the kernels is not always optimal for performance. Every kernel has some output and when merging kernels are simply cascaded the output of one kernel is put in the middle of the newly merged kernel code. In most instances this causes the compiler to produce a wait acknowledgment assembly instruction $(WAIT_ACK)$, which adds synchronization to the kernel and causes slowdown because it waits for acknowledgments back from either the read or write memory units. To remove this synchronization, the output of all the merged kernels are moved toward the end of the kernel as shown in Figure 5. In this way, the register pressure is not significantly affected. It's not always possible to put the outputs right next to each other for the kernels being merged, instead a best effort is made while keeping register pressure as low as possible (without reducing occupancy) and eliminating the $(WAIT_ACK)$ instruction. Similar to the principle of maximizing

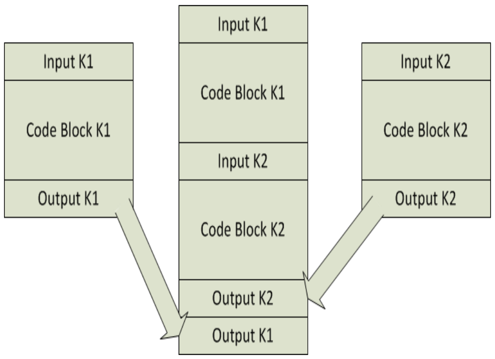


Fig. 5. Moving Output of Merged Kernels

the grouping of outputs, every attempt should also be made to group the inputs. However, this is not due to a wait acknowledgment after a memory read but is driven by the goal to group the memory reads in the same memory read clause, so as to reduce any delay caused by switching between two different such clauses or any delay caused by data dependence. Since the effect of this optimization might not be high, it is not performed if the register count needs to be increased to the point of reducing the number of wavefronts/warps. This is shown in Figure 6. For AMD GPUs, the equation to calculate the number of simultaneous wavefronts is:

$$\text{Simul WFs} = \frac{\text{Regs per Thread} - \text{Temp Regs used}}{\text{Regs Used}} \quad (8)$$

4.4 Fences and Barriers

There are many types of kernels which require synchronization and in these kernels fences and barriers are used to implement this synchronization. There are two cases to consider when merging kernels with synchronization primitives: both kernels have synchronization or one kernel has synchronization. In the case of both kernels having

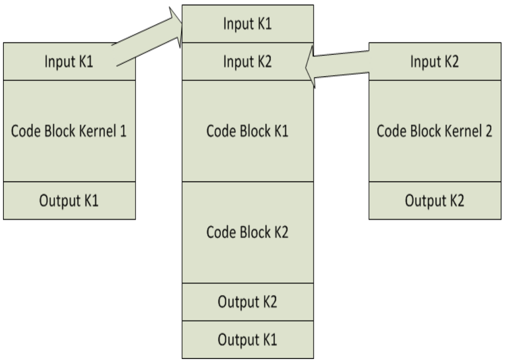


Fig. 6. Moving Input of Merged Kernels

synchronization each kernel code is broken into blocks separated by these fences or barriers. The blocks are then merged based on their sequence according to the blocks as shown in Figure 7. The case of merging kernels with synchronization primitives is a good example showing that a simple merging approach such as cascading can sometimes destroy performance. If the kernels were simply cascaded the total number of fences or barriers would be the sum of the synchronizations of all the kernels being merged. However, merging kernels at the granularity of code blocks that are synchronized allows the number of synchronizations to remain the same as the kernel with the highest number of synchronizations because the kernels now share the same barriers, thus eliminating the need for duplicate barriers. For example, if each kernel performs some operations and then synchronizes, in the merged kernel they can both perform these operations at the same time and then synchronize together.

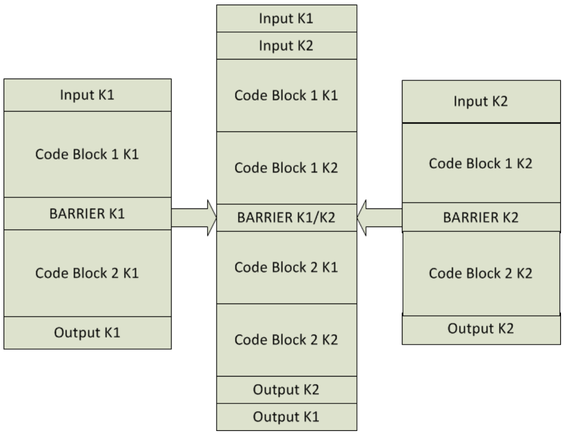


Fig. 7. Merging Kernels with Synchronization Points

If only one kernel has synchronization operations, the kernel that does not have synchronization is placed either before or after the synchronization operation depending on which option gives the lowest register usage, which can be determined statically. This technique does not exclude the other techniques mentioned above, meaning that both moving the output and inputs is taken in to account when merging kernels with synchronization.

4.5 Limiting Factors

The proposed kernel merging technique cannot be employed freely. There are a few other limiting factors that should be considered when deciding whether to apply this transformation or not. One limitation is caused by the possible changes of the dimensions of a GPU kernel when it is merged with another kernel, and the consequent changes in its memory access patterns. For example, changing the local dimension from 1D to 2D, or vice versa, could cause global memory conflicts or contention. Kernels work best when they execute over the dimensions, both global and local, for which they were coded. For some applications a specific local thread size and/or dimension are needed for correct results, particularly for problems that utilize a certain block size for local memory calculations. For example, matrix operations often fall into this category. There are also applications that may have their performance effected by the local dimension size when going from one dimension to larger dimensions due to a change in memory access size based on the algorithm. For example, image filters often look at the surrounding pixels and the size of the reads can change depending on the local dimension, whether they are just looking forward and backward or forward, backward, side to side and diagonally.

Our heuristic considers both local memory usage and local thread size. If a kernel has a two dimension local thread size (block size) it will not be reduced when merging. What this means is that if this kernel is merged with a one dimension local thread size kernel, that kernel's local thread size is changed to two dimensions, unless changing the local thread size effects the fetch size performance (architecture dependent). Our heuristic checks to make sure that the combined local memory usage does not exceed the local memory available for the architecture, if it does then those kernels are not merged.

5 Evaluation

We evaluate our code merging transformation technique on both AMD and NVIDIA GPUs. All of the results presented in this paper were taken with maximum global size and maximum memory usage allowed for the given problem, thereby imitating a large data inter-process environment. The host-to-device transfer will not be addressed in our evaluation since it does not impact the effectiveness of the proposed transformation in one way or the other. The AMD GPU used was the 5870 which has the RV870 chip and 1GB of memory. The NVIDIA GPU used was the GTX280 which has the GT200 chip and 1GB of memory. The specifications of both GPU architectures are mentioned in the GPU Background section of this paper. The latest drivers and SDKs were used at the time of this writing, which was the AMD Stream SDK 2.2 and NVIDIA's CUDA/OpenCL 3.1 Toolkit. All profiling information listed was obtained using the AMD OpenCL profiler.

The benchmarks used for evaluation were taken from the AMD Stream SDK 2.2 OpenCL samples. The code is compatible with but might not be tuned for NVIDIA hardware. However, since we use the same compiler options before and after our transformation, the speedup validly reflect the effectiveness of our approach. The runtimes are taken directly from these samples, unaltered for either GPU architecture. The merged kernels were merged without modifying any kernel code other than the transformation techniques outlined in the Transformation section of this paper. All of the speedups shown were compared to the combined single benchmark times using the same global size with an optimal local size. The benchmarks are one-dimension BlackScholes, Mersenne Twister, SobelFilter and SimpleConvolution, as well as two-dimension DCT and MatrixTranspose. The six benchmarks can be combined into 15 different pairs (C_6^2), and we tested all combinations that passed the heuristic tests.

The results for both the AMD 5870 and the NVIDIA GTX280 are listed in Figures 8 and 9, respectively. All the timing results were obtained using the respective profilers: AMD's Stream OpenCL profiler and NVIDIA's OpenCL profiler. The AMD GPU shows the most significant performance gain with an average speedup of 1.28 and a maximum speedup of 1.53. Good speedups were also obtained for the GTX280 (average speedup 1.17 and max 1.37).

5.1 Effects of Resource Usage

Barring any of the mentioned limiting factors, the best performance results come from the merging of ALU bound kernels with memory bound kernels. This conforms to our transformation heuristics and allows for the greatest increase in overlap. For our benchmarks, the DCT, MatrixTranspose and SimpleConvolution all had relatively low ALU values and overlap values while the BlackScholes, SobelFilter and MersenneTwister samples had relatively high ALU and overlap values, as can be seen in Table 2.

Table 2. Profiled Resource Utilization of Single Benchmark

Kernel	ALU	Read	Write	LDS	Overlap
DCT	0.381	0.224	0.016	0.429	0.049
Black Scholes	0.997	0.222	0.110	0	0.241
Mersenne Twister	0.956	0.086	0.130	0.280	0.311
MaxTrans	0.025	0.006	0.010	0.057	-9.06
Simple Conv	0.434	0.334	0.026	0	-0.25
Sobel Filter	0.960	0.231	0.035	0	0.184

In Table 2 the kernels with a high ALU value also have a high overlap value since these kernels are ALU bound and, combined with context switching, allows for a high amount of memory latency hiding. This means that these kernels will merge well. The DCT sample is LDS bound and therefore has little overlap and will merge well with a kernel that has either a high ALU value, a high read value or a high write value. Both the Matrix Transpose and the Simple Convolution benchmarks show a negative overlap, however. The profiler does not give the information about all aspects of execution that might impact performance, such as unreported bank conflicts, LDS stalls, synchronization stalls, and context switching latency. However, for the purpose of merging, since we know the negative overlap is not caused by the ALU or read values, the negative value itself indicates that there is heavy idleness.

The values are added to predict which kernels are good merge candidates. For example, if two kernels are to be merged the speedup gained is going to decrease as any value approaches 2. For example, for any combination of Black Scholes, Mersenne Twister or Sobel Filter on NVIDIA 280 GTX, the sum of their ALU values readily approaches the threshold and so the speedup is nominal. The ALU value for Matrix Transpose and Black Scholes is just over 1, meaning that most of the other operations can be hidden by ALU operations while being just slightly more ALU bound. This method is done for each of the first four values with the same concept. For example, if two kernels had read values of .99 then they wouldn't be good merge candidates. The best merge candidates are going to stress different components of the GPU, ideally a perfectly merged kernel with have values of 1. This can be extended to multiple kernels (more than two) since there are more than two components that work in parallel and most kernels won't use close to 100% of any of those components. For example, a high ALU value kernel could be very well merged with a high read value kernel and a high LDS value kernel. This can be extended to as many kernels as the heuristic will predict speedup.

Looking at Table 3, we can see that the Matrix Transpose kernel merges well with almost every other kernel since it's values are so small and it's overlap is deeply negative, there's plenty of opportunity for other kernels to do work during it's runtime. This is expressed in Table 3 by the improvement in the overlap value.

Table 3. Heuristics values for kernel combinations

Merged Kernel	ALU	Read	Write	LDS	Overlap
DCT+Twister	0.770	0.141	0.104	0.370	0.278
DCT+Scholes	0.803	0.258	0.085	0.240	0.278
Scholes+MaxTrans	0.810	0.185	0.114	0.071	0.153
Scholes+Twister	0.978	0.139	0.144	0.206	0.318
Twister+MaxTrans	0.765	0.072	0.125	0.297	0.206
DCT+MaxTrans	0.219	0.128	0.018	0.294	-0.514
Scholes+Simple	0.854	0.263	0.098	0	0.177
Twister+Simple	0.836	0.138	0.122	0.248	0.257
DCT+Simple	0.351	0.217	0.018	0.233	-0.217
Simple+MaxTrans	0.122	0.077	0.013	0.037	-2.98
Scholes+Sobel	0.998	0.223	0.109	0	0.249
Twister+Sobel	0.940	0.106	0.131	0.266	0.308
DCT+Sobel	0.442	0.182	0.021	0.279	-0.079
Simple+Sobel	0.683	0.297	0.042	0	0.0227
Sobel+MaxTrans	0.131	0.029	0.013	0.038	-3.68

5.2 The Effects of Barriers

Of the 6 benchmarks only the DCT and MatrixTranspose have barriers. Each of the two has one barrier. These two kernels are merged according to the Fences and Barriers subsection of the Transformation section. That is, the code segments that are separated by barriers are interleaved in the merged kernel. When merging the DCT and Matrix-Transpose, the code sections on each side of the barrier were put on the proper side of the barrier in the merged kernels, so no extra barriers were needed. Despite the need for synchronization, these samples give good performance on the AMD GPU when merged with BlackScholes and MersenneTwister, both of whom are ALU bound. DCT and MatrixTranspose also show speedup when merged together, both due to Matrix Transpose's low overlap value and interleaving the barriers.

5.3 Candidate Elimination

Table 3 shows all of the values for all possible combinations and indicates which combinations are good candidates for merging based on our heuristics. Both the Sobel Filter and Simple Convolution cause an increase in fetch size when going from a local thread dimension size of one to two on AMD's Radeon 5870 and the Sobel Filter on the GTX280. The increase in runtime is greater than the expected benefit from the transformation so the kernels aren't merged when it requires that they be transformed into two dimensional problems. Additionally for the NVIDIA GPU, the combinations that were eliminated were Scholes Twister, Scholes Sobel and Sobel Twister. All of these kernels have high ALU values and since the NVIDIA GPU doesn't use the VLIW architecture there is no possible gain from ALU packing. This causes their runtimes to be serialized and no improvement can be made. These three kernels might also be eliminated from the AMD GPU if not for the possible gain in performance from an increase in ALU packing. The advantage of ALU packing does not need to be profiled and can be attained statically through the compiled ISA. The speedups from these combinations of ALU bound kernels on the AMD GPU comes from ALU packing. The number of ALU cycles saved in the Scholes Twister sample was about double that of the Sobel Twister and Sobel Scholes samples. Figures 8 and 9 show the results of all the combinations that pass the heuristics and are actually merged. The left column in all the figures shows the execution time of kernels executed separately, and the right column shows the execution time of the corresponding merged kernel.

Just for the purpose to show the effectiveness of our heuristics, Figure 10 shows all of the results for the combinations of benchmarks that did not pass our heuristics. The combinations in this figure have almost the same execution time for the merged kernel as the combined execution time of the individual kernels. On the other hand, it also shows that our transformation is "safe" in the sense that even if two wrong kernels are merged, the performance will only marginally decrease.

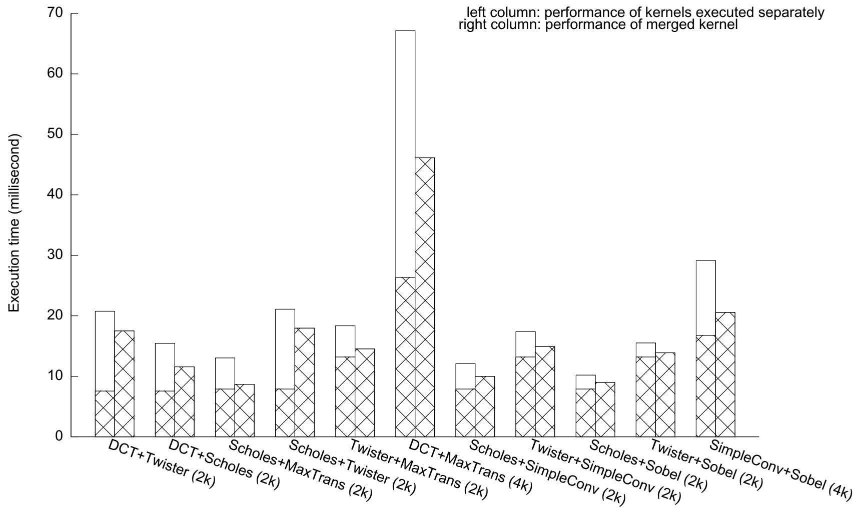


Fig. 8. AMD 5870 Results

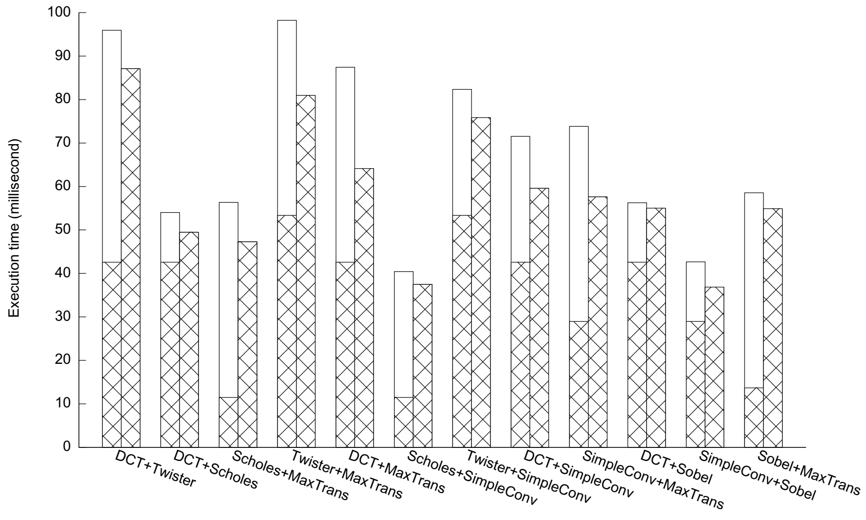


Fig. 9. NVIDIA GTX280 Results

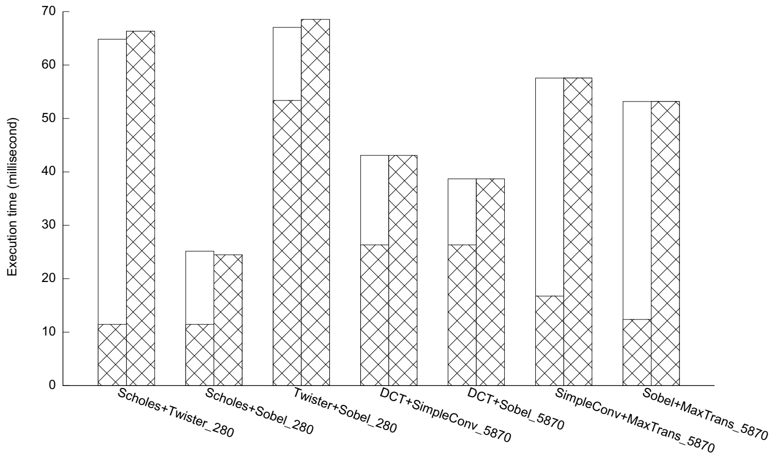


Fig. 10. Results of Poor Merging Candidates

6 Related Work and Conclusion

The performance effect of resource sharing and competition has been studied before in the context of multi-threaded CPU programs. Resources in a CPU including cache, off-chip bandwidth and memory are shared among many multi-tasked threads. A preview of the phenomena of inter-thread conflict and thrashing is already observed in multi-threaded machines such as the Intel Pentium 4 that supports hyperthreading [8,9], even though there are only two threads that share a processor and other resources. [10,11,12] explores the possibility of optimizing multiple programs that will run simultaneously on multi-processors. Generally, the previous approaches are designed for multi-threaded programs with a small number of threads and cannot be easily scaled up to hundreds of threads that run in parallel on a GPU. Intra-thread optimizations for GPU have been long studied. There are many well documented optimizations for GPU programs that are applied at the thread or wavefront/warp level to improve the usage of GPU resources. A recent example is a systematic framework to optimize for GPU memory hierarchy that is proposed in [13]. A good overview of intra-thread optimizations can be found in [14]. On the other hand, however, little is known about optimizations for resources at a higher level such as at the kernel level.

In this paper we present a novel transformation that merges inter-process kernels to improve program performance on the GPU. The proposed transformation is motivated by the fact that while there are many techniques that optimize GPU kernels at the thread level there are no techniques that help to increase resource utilization below the kernel level. Concurrent kernels, while improving overall GPU usage by utilizing more resource units at one time, does not help to improve individual resource units' usage within the GPU. We discuss how the code merging transformation has minimal effect on optimizations done at a finer granularity and can help increase performance when combined with other optimizations. We give a set of heuristics for selecting which kernels to merge and how to merge them along with real world application results from both major GPU vendors for a wide array of benchmarks.

In the future, we would like to extend this work to include larger benchmark applications that contain multiple kernels. We would also like to extend this work to include results for Fermi machines and for the new 4-wide VLIW AMD machines, for both the set of benchmarks presented and the planned future larger benchmark applications. A comparison between the speedups of this technique and the speedups from using concurrent kernel execution will also be included.

References

1. Nvidia OpenCL Programming Guide (May 2010)
2. AMD OpenCL Programming Guide (June 2010)
3. Ryoo, S., Rodrigues, C., Baghsorkhi, S., Stone, S., Kirk, D., Hwu, 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, pp. 73–82. ACM (2008)
4. Ueng, S.-Z., Lathara, M., Baghsorkhi, S.S., Hwu, W.-M.W.: CUDA-Lite: Reducing GPU Programming Complexity. In: Amaral, J.N. (ed.) LCPC 2008. LNCS, vol. 5335, pp. 1–15. Springer, Heidelberg (2008)
5. Zhang, E., Jiang, Y., Guo, Z., Shen, X.: Streamlining GPU applications on the fly. In: Proceedings of the 24th ACM International Conference on Supercomputing, pp. 115–126. ACM (2010)
6. Chen, L., Villa, O., Krishnamoorthy, S., Ga, G.: Dynamic load balancing on single- and multi-GPU systems. In: 2010 IEEE International Symposium on Parallel and Distributed Processing, pp. 1–12. IEEE (2010)
7. Carrillo, S., Siegel, J., Li, X.: A control-structure splitting optimization for GPGPU. In: Proceedings of the 6th ACM Conference on Computing Frontiers, pp. 147–150. ACM (2009)
8. Hily, S., Seznec, A.: Contention on 2nd Level Cache May Limit the Effectiveness of Simultaneous Multithreading. Technical Report PI-1086 (1997)
9. Leng, T., Ali, R., Hsieh, J., Mashayekhi, V., Rooholamini, R.: A Study of Hyper-threading in High-performance Computing Clusters. In: Dell Power Solutions HPC Cluster Environment, pp. 33–36 (2002)
10. Kandemir, M.: Compiler-Directed Collective-I/O. IEEE Trans. Parallel Distrib. Syst. 12(12), 1318–1331 (2001)
11. Hom, J., Kremer, U.: Inter-program Optimizations for Conserving Disk Energy. In: ISLPED 2005: Proceedings of the 2005 International Symposium on Low Power Electronics and Design, pp. 335–338. ACM Press, New York (2005)
12. Ozturk, O., Chen, G., Kandemir, M.: Multi-compilation: Capturing Interactions Among Concurrently-executing Applications. In: CF 2006: Proceedings of the 3rd Conference on Computing Frontiers, pp. 157–170. ACM Press, New York (2006)
13. Yang, Y., Xiang, P., Kong, J., Zhou, H.: A GPGPU compiler for memory optimization and parallelism management. In: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 86–97. ACM (2010)
14. Ryoo, S., Rodrigues, C.I., Stone, S.S., Stratton, J.A., Ueng, S.Z., Baghsorkhi, S.S., Hwu, W.W.: Program optimization carving for GPU computing. Journal of Parallel and Distributed Computing 68(10), 1389–1401 (2008)

Static Compilation Analysis for Host-Accelerator Communication Optimization

Mehdi Amini[1,2], Fabien Coelho[2], François Irigoin[2], and Ronan Keryell[1]

[1] HPC Project, Meudon, France
`name.surname@hpc-project.com`
[2] MINES ParisTech/CRI, Fontainebleau, France
`name.surname@mines-paristech.fr`

Abstract. We present an automatic, static program transformation that schedules and generates efficient memory transfers between a computer host and its hardware accelerator, addressing a well-known performance bottleneck. Our automatic approach uses two simple heuristics: to perform transfers to the accelerator as early as possible and to delay transfers back from the accelerator as late as possible. We implemented this transformation as a middle-end compilation pass in the PIPS/PAR4ALL compiler. In the generated code, redundant communications due to data reuse between kernel executions are avoided. Instructions that initiate transfers are scheduled effectively at compile-time. We present experimental results obtained with the Polybench 2.0, some Rodinia benchmarks, and with a real numerical simulation. We obtain an average speedup of 4 to 5 when compared to a naïve parallelization using a modern GPU with PAR4ALL, HMPP, and PGI, and 3.5 when compared to an OPENMP version using a 12-core multiprocessor.

Keywords: Automatic parallelization, communication optimization, source-to-source compilation, heterogeneous parallel architecture, GPU.

1 Introduction

Hybrid computers based on hardware accelerators are growing as a preferred method to improve performance of massively parallel software. In 2008, *Roadrunner*, using *PowerXCell 8i* accelerators, headed the TOP500 ranking. The June 2011 version of this list and of the Green500 list both include in the top 5 three hybrid supercomputers employing NVIDIA GPUs. These highly parallel hardware accelerators allow potentially better performance-price and performance-watts ratios when compared to classical multi-core CPUs. The same evolution is evident in the embedded and mobile world (NVIDIA Tegra, etc.). In all, the near future of high performance computing appears heterogeneous.

The disadvantage of heterogeneity is the complexity of its programming model: the code executing on the accelerator cannot directly access the host memory and *vice-versa* for the CPU. Explicit communications are used to exchange data, via slow IO buses. For example, PCI bus offers 8 GB/s. This is generally thought to be *the* most important bottleneck for hybrid systems [7,9,10].

Though some recent architectures avoid explicit copy instructions, the low performance PCI bus is still a limitation.

We propose with PAR4ALL [15] an open source initiative to unify efforts concerning compilers for parallel architectures. It supports the automatic integrated compilation of applications for hybrid architectures. Its basic compilation scheme generates parallel and hybrid code that is correct, but lacks efficiency due to redundant communications between the host and the accelerator.

Much work has been done regarding communication optimization for distributed computers. Examples include message fusion in the context of SPDD (*Single Program Distributed Data*) [12] and data flow analysis based on array regions to eliminate redundant communications and to overlap the remaining communications with compute operations [13].

We apply similar methods to offload computation in the context of a host-accelerator relationship and to integrate in a parallelizing compiler a transformation that optimizes CPU-GPU communications at compile time. In this paper we briefly present existing approaches addressing the issue of writing software for accelerators (§ 2). We identify practical cases for numerical simulations that can benefit from hardware accelerators. We show the limit of automatic transformation without a specific optimization for communication (§ 3). We present a new data flow analysis designed to optimize the static generation of memory transfers between host and accelerator (§ 4). Then, using a 12-core Xeon multiprocessor machine with a NVIDIA Tesla GPU C2050, we evaluate our solution on well known benchmarks [22,6]. Finally, we show that our approach scales well with a real numerical cosmological simulation (§ 5).

2 Automatic or Semi-automatic Transformations for Hardware Accelerators

Targeting hardware accelerators is hard work for a software developer when done fully manually. At the highest level of abstraction and programmer convenience, there are APIs and C-like programming languages such as CUDA, OPENCL. At lower levels there are assembly and hardware description languages like VHDL. NVIDIA CUDA is a proprietary C-extension with some C++ features, limited to NVIDIA GPUs. The OPENCL standard includes an API that presents an abstraction of the target architecture. However, manufacturers can propose proprietary extensions. In practice, OPENCL still leads to a code tuned for a particular accelerator or architecture. Devices like FPGAs are generally configured with languages like VHDL. Tools like the Altera C-to-VHDL compiler `c2h` attempt to raise the level of abstraction and convenience in device programming.

2.1 Semi-automatic Approach

Recent compilers comprise an incremental way for converting software toward accelerators. For instance, the PGI Accelerator [23] requires the use of directives. The programmer must select the pieces of source that are to be executed on the accelerator, providing optional directives that act as hints for data allocations and transfers. The compiler generates all code automatically.

HMPP [5], the CAPS compiler, works in a similar way: the user inserts directives to describe the parameters required for code generation. Specialization and optimization possibilities are greater, but with the same drawback as OpenCL extensions: the resulting code is tied to a specific architecture.

JCUDA [24] offers a simpler interface to target CUDA from JAVA. Data transfers are automatically generated for each call. Arguments can be declared as IN, OUT, or INOUT to avoid useless transfers, but no piece of data can be kept in the GPU memory between two kernel launches. There have also been several initiatives to automate transformations for OpenMP annotated source code to CUDA [20,21]. The GPU programming model and the host accelerator paradigm greatly restrict the potential of this approach, since OpenMP is designed for shared memory computer. Recent work [14,19] adds extensions to OpenMP that account for CUDA specificity. But, these lead again to specialized source code.

These approaches offer either very limited automatic optimization of host-accelerator communications or none at all. OpenMPC [19] includes an interprocedural liveness analysis to remove some useless memory transfers, but it does not optimize their insertion. Recently, new directives were added to the PGI [23] accelerator compiler to precisely control data movements. These make programs easier to write, but the developer is still responsible for designing and writing communications code.

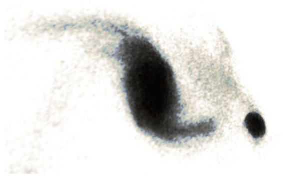
2.2 Fully Automatic Approach: Par4All

Par4All [15] is an open-source initiative that aims to develop a parallelizing compiler based on source-to-source transformations. The current development version (1.2.1) generates OpenMP from C and Fortran source code for shared memory architecture and CUDA or OpenCL for hardware accelerators including NVIDIA and ATI GPUs. The automatic transformation process in Par4All is heavily based on PIPS compiler framework phases [17,1]. The latter uses a linear algebra library [2] to analyze, transform and parallelize programs using polyhedral representations [11]. Parallel loop nests are outlined (*i.e.* extracted) in new functions tagged as eligible on the GPU. They are called *kernels* in the hybrid programming terminology. Convex array region analyses [8] are used to characterize the data used and defined by each kernel.

Par4All often parallelizes computations with no resulting speedup because communication times dominate. The new transformation presented here obtains better speedups by improving communication efficiency.

3 *Stars-PM* Simulation

Small benchmarks like the Polybench suite [22] are limited to a few kernels, sometimes surrounded with a time step loop. Thus they are not representative of a whole application when evaluating a global optimization. To address this issue, we do not limit our experiments to the Polybench benchmarks, but we also include *Stars-PM*, a particle mesh cosmological N-body code. The sequential version was written in C at *Observatoire Astronomique de Strasbourg* and was rewritten and optimized by hand using CUDA to target GPUs [3].



(a) A satellite triggers a bar and spiral arms in a galactic disc.

```
int main(int argc, char *argv[]) {
  /* Read data from a file */
  init_data(argv[1]);
  /* Main temporal loop */
  for(t = 0; t < T; t += DT)
    iteration(...);
  /* Output results to a file */
  write_data(argv[2]);
}
```

(b) Simplified scheme for simulations.

```
void iteration(coord pos[NP][NP][NP],
               coord vel[NP][NP][NP],
               float dens[NP][NP][NP],
               int data[NP][NP][NP],
               int histo[NP][NP][NP]) {

  /* Cut the tri-dimensionnal space
   * in a regular mesh */
  discretisation(pos, data);
  /* Compute density on the grid */
  histogram(data, histo);
  /* Compute potential on the mesh
   * in the Fourier space */
  potential(histo, dens);
  /* For each dimension, compute the
   * force and then update the speed */
  forcex(dens, force);
  updatevel(vel, force, data, 0, dt);
  forcey(dens, force);
  updatevel(vel, force, data, 1, dt);
  forcez(dens, force);
  updatevel(vel, force, data, 2, dt);
  /* Move particles */
  updatepos(pos, vel);
}
```

(c) Time step in the cosmological simulation.

Fig. 1. Outline of the *Stars-PM* cosmological simulation code

This simulation is a model of gravitational interactions between particles in space. It represents three-dimensional space with a discrete grid. Initial conditions are read from a file. A sequential loop iterates over successive time steps. Results are computed from the final grid state and stored in an output file. This general organization is shown in the simplified code shown in Fig. 1b. It is a common technique in numerical simulations. The processing for a time step is illustrated Fig. 1c.

3.1 Transformation Process

The automatic transformation process in PAR4ALL is based on parallel loop nest detection. Loop nests are then outlined to obtain kernel functions.

The simplified code for function `discretization(pos, data)` is provided before and after the transformation in Figs. 2 and 3 respectively. The loop nest is detected as parallel and selected to be transformed into a kernel. The loop body is outlined in a new function that will be executed by the GPU, and the loop nest is replaced by a call to a kernel launch function. PIPS performs several array regions analyses: $\mathcal{W}$ (resp. $\mathcal{R}$) is the region of an array written (resp. read) by a statement or a sequence of statements, for example a loop or a function. PIPS

```
void discretization(coord pos[NP][NP][NP],
                    int data[NP][NP][NP]){
  int i, j, k;
  float x, y, z;
  for (i = 0; i < NP; i++)
    for (j = 0; j < NP; j++)
      for (k = 0; k < NP; k++) {
        x = pos[i][j][k].x;
        y = pos[i][j][k].y;
        z = pos[i][j][k].z;
        data[i][j][k] = (int)(x/DX)*NP*NP
                      + (int)(y/DX)*NP
                      + (int)(z/DX);
      }
}
```

Fig. 2. Sequential source code for function `discretization`

also computes $\mathcal{IN}$ and $\mathcal{OUT}$ [8] regions. These are conservative over-estimates of the respective array areas used ($\mathcal{IN}$) and defined ($\mathcal{OUT}$) by the kernel. $\mathcal{IN}$ regions must be copied from host to GPU before kernel execution. $\mathcal{OUT}$ regions must be copied back afterward.

Looking at function `discretization` (Fig. 2) we observe that the `pos` array is used in the kernel, whereas `data` array is written. Two transfers are generated (Fig. 3). One ensures that data is moved to the GPU before kernel execution and the other copies the result back to the host memory after kernel execution.

```
void discretization(coord pos[NP][NP][NP],int data[NP][NP][NP]) {
 // Pointers to memory on accelerator:
 coord (*pos0)[NP][NP][NP] = (coord (*)[NP][NP][NP]) 0;
 int (*data0)[NP][NP][NP] = (int (*)[NP][NP][NP]) 0;
 // Allocating buffers on the GPU and copy in
 P4A_accel_malloc((void **) &data0, sizeof(int)*NP*NP*NP);
 P4A_accel_malloc((void **) &pos0, sizeof(coord)*NP*NP*NP);
 P4A_copy_to_accel(sizeof(coord)*NP*NP*NP, pos, *pos0);
 P4A_call_accel_kernel_2d(discretization_kernel,NP,NP,*pos0,*data0);
 // Copy out and GPU buffers deallocation
 P4A_copy_from_accel(sizeof(int)*NP*NP*NP, data, *data0);
 P4A_accel_free(data0);
 P4A_accel_free(pos0);
}
// The kernel corresponding to sequential loop body
P4A_accel_kernel discretization_kernel( coord *pos, int *data ) {
   int k; float x, y, z;
   int i = P4A_vp_1; // P4A_vp_* are mapped from CUDA BlockIdx.*
   int j = P4A_vp_0; // and ThreadIdx.* to loop indices
   // Iteration clamping to avoid GPU iteration overrun:
   if (i<=NP&&j<=NP)
      for(k = 0; k < NP; k += 1) {
         x = (*(pos+k+NP*NP*i+NP*j)).x;
         y = (*(pos+k+NP*NP*i+NP*j)).y;
         z = (*(pos+k+NP*NP*i+NP*j)).z;
         *(data+k+NP*NP*i+NP*j) = (int)(x/DX)*NP*NP
                                + (int)(y/DX)*NP
                                + (int)(z/DX);
      }
}
```

Fig. 3. Code for function `discretization` after automatic GPU code generation

3.2 Limit of This Approach

Data exchanges between host and accelerator are executed as DMA transfers between RAM memories across the PCI-express bus, which currently offers a theoretical bandwidth of 8 GB/s. This is really small compared to the GPU memory bandwidth which is close to 150 GB/s. This low bandwidth can annihilate all gain obtained when offloading computations in kernels, unless they are really compute intensive.

With our hardware (see § 5), we measure up to 5.6 GB/s from the host to the GPU, and 6.2 GB/s back. This is obtained for a few tens of MB, but decreases dramatically for smaller blocks. Moreover this bandwidth is reduced by more than a half when the transfered memory areas are not *pinned*—physically contiguous and not subject to paging by the virtual memory manager. Figure 5 illustrates this behavior.

In our tests, using as reference a cube with 128 cells per edge and as many particles as cells, for a function like `discretization`, one copy to the GPU for particle positions is a block of 25 MB. One copy back for the particle-to-cell association is a 8 MB block. The communication time for these two copies is about 5 ms. Recent GPUs offer ECC hardware memory error checking that more than doubles time needed for the same copies to 12 ms. Each buffer allocation and deallocation requires 10 ms. In comparison, kernel execution requires only 0.37 ms on the GPU, but 37 ms on the CPU. We note that the memory transfers and buffer allocations represent the largest potential for obtaining high speedups, which motivates our work.

3.3 Observations

In each time step, function `iteration` (Fig. 1c) uses data defined by a previous one. The parallelized code performs many transfers from the GPU followed immediately by the opposite transfer.

Our simulation (Fig. 1b) exemplifies the common pattern of data dependencies between loop iterations, where the current iteration uses data defined during previous ones. There is clear advantage in allowing such data to remain on the GPU, with copies back to the host only as needed for checkpoints and final results.

4 Optimization Algorithm

We propose a new analysis for the compiler middle-end to support efficient host-GPU data copying. The host and the accelerator have separated memory spaces, this analysis annotates internally the source program with information about where up-to-date copies of data lie—in host and/or GPU memory. This allows additional transformation to statically determine good places to insert asynchronous transfers with a simple strategy: Launch transfers from host to GPU as early as possible and launch those from GPU back to host as late as possible, while still guaranteeing data integrity. Additionally, we avoid launching transfers inside loops wherever possible. We use a heuristic to place transfers as high as possible in the call graph and in the AST.[1]

4.1 Definitions

The analysis computes the following sets for each statement:

- $\mathcal{U}_A^>$ is the set of arrays known to be *u*sed next ($>$) to the *a*ccelerator;
- $\mathcal{D}_A^<$ is the set of arrays known to be lastly ($<$) *d*efined on the *a*ccelerator;
- $\mathcal{T}_{H\rightarrow A}$ is the set of arrays to transfer to the accelerator memory space immediately after the statement;
- $\mathcal{T}_{A\rightarrow H}$ is the set of arrays to transfer from the accelerator on the host immediately before the statement.

4.2 Intraprocedural Phase

The analysis begins with the $\mathcal{D}_A^<$ set in a forward pass. An array is defined on the GPU for a statement S iff this is also the case for its immediate predecessors in the control flow graph and if the array is not used or defined by the host, *i.e.* is not in the set $\mathcal{R}(I)$ or $\mathcal{W}(I)$ computed by PIPS:

$$\mathcal{D}_A^<(S) = \left(\bigcap_{S' \in \text{pred}(S)} \mathcal{D}_A^<(S') \right) - \mathcal{R}(S) - \mathcal{W}(S) \tag{1}$$

The initialization is done at the first kernel call site S_k with the arrays defined by the kernel k and used later ($\mathcal{OUT}(S_k)$). The following equation is used at each kernel call site:

$$\mathcal{D}_A^<(S_k) = \mathcal{OUT}(S_k) \bigcup \left(\bigcap_{S' \in \text{pred}(S_k)} \mathcal{D}_A^<(S') \right) \tag{2}$$

A backward pass is then performed in order to build $\mathcal{U}_A^>$. For a statement S, an array has its next use on the accelerator iff this is also the case for all statements immediately following in the control flow graph, and if it is not defined by S.

$$\mathcal{U}_A^>(S) = \left(\bigcup_{S' \in \text{succ}(S)} \mathcal{U}_A^>(S') \right) - \mathcal{W}(S) \tag{3}$$

Just as $\mathcal{D}_A^<$, $\mathcal{U}_A^>$ is initially empty and is initialized at kernel call sites with the arrays necessary to run the kernel, $\mathcal{IN}(S_k)$, and the arrays defined by the kernel, $\mathcal{W}(S_k)$. These defined arrays have to be transferred to the GPU if we cannot established that they are entirely written by the kernel. Otherwise, we might overwrite still-valid data when copying back the array from the GPU after kernel execution:

[1] PIPS uses a hierarchical control flow graph [17,1] to preserve as much as possible of the AST. However, to simplify the presentation of the analyses, we assume that a CFG is available.

$$\mathcal{U}_A^>(S_k) = \mathcal{IN}(S_k) \bigcup \mathcal{W}(S_k) \bigcup \left(\bigcup_{S' \in \mathrm{succ}(S_k)} \mathcal{U}_A^>(S') \right) \tag{4}$$

An array must be transferred from the accelerator to the host after a statement S iff its last definition is in a kernel and if it is not the case for at least one of the immediately following statements:

$$\mathcal{T}_{A \to H}(S) = \mathcal{D}_A^<(S) - \bigcap_{S' \in \mathrm{succ}(S)} \mathcal{D}_A^<(S') \tag{5}$$

This set is used to generate a copy operation at the latest possible location.

An array must be transferred from the host to the accelerator if its next use is on the accelerator. To perform the communication *at the earliest*, we place its launch immediately after the statement that defines it, *i.e.* the statement whose $\mathcal{W}(S)$ set contains it.

$$\mathcal{T}_{H \to A}(S) = \mathcal{W}(S) \bigcap \left(\bigcup_{S' \in \mathrm{succ}(S)} \mathcal{U}_A^>(S') \right) \tag{6}$$

4.3 Interprocedural Extension

Kernel calls are potentially localized deep in the call graph. Consequently, a reuse between kernels requires interprocedural analysis. Function `iteration` (Fig. 1c) illustrates this situation, each step corresponds to one or more kernel executions.

Our approach is to perform a backward analysis on the call graph. For each function f, *summary* sets $\overline{\mathcal{D}_A^<}(f)$ and $\overline{\mathcal{U}_A^>}(f)$ are computed. They summarize information about the formal parameters for the function. These sets can be viewed as contracts. They specify a data mapping that the call site must conform to. All arrays present in $\overline{\mathcal{U}_A^>}(f)$ must be transferred to the GPU before the call, and all arrays defined in $\overline{\mathcal{D}_A^<}(f)$ must be transferred back from the GPU before any use on the host. These sets are required in the computation of $\mathcal{D}_A^<$ and $\mathcal{U}_A^>$ when a call site is encountered. Indeed at a call site c for a function f, each argument of the call that corresponds to a formal parameter present in $\overline{\mathcal{U}_A^>}$ must be transferred to the GPU before the call, because we know that the first use in the called function occurs in a kernel. Similarly, an argument that is present in $\overline{\mathcal{D}_A^<}$ has been defined in a kernel during the call and not already transferred back when the call ends. This transfer can be scheduled later, but before any use on the host.

Equations 1 and 3 are modified for call site by adding a translation operator, $\mathrm{trans}_{\mathrm{f} \to \mathrm{c}}$, between arguments and formal parameters:

$$\mathcal{D}_A^<(c) = \left(\mathrm{trans}_{\mathrm{f} \to \mathrm{c}}(\overline{\mathcal{D}_A^<}(f)) \bigcup \left(\bigcap_{S' \in \mathrm{pred}(c)} \mathcal{D}_A^<(S') \right) \right) - \mathcal{R}(c) - \mathcal{W}(c) \tag{7}$$

$$\mathcal{U}_A^>(c) = \left(\mathrm{trans}_{\mathrm{f} \to \mathrm{c}}(\overline{\mathcal{U}_A^>}(f)) \bigcup \left(\bigcup_{S' \in \mathrm{succ}(c)} \mathcal{U}_A^>(S') \right) \right) - \mathcal{W}(c) \tag{8}$$

On the code Fig. 4, we observe, comparing the result of the interprocedural optimized code with the very local approach (Fig. 3), that all communications and memory management (allocation/deallocation) have been eliminated from the main loop.

```
void discretization(coord pos[NP][NP][NP],
                    int data[NP][NP][NP]) {
  //generated variable
  coord *pos0 = P4A_runtime_resolve(pos,NP*NP*NP*sizeof(coord));
  int *data0 = P4A_runtime_resolve(pos,NP*NP*NP*sizeof(int));
  // Call kernel
  P4A_call_accel_kernel_2d(discretization_kernel,
                           NP, NP, *pos0, *data0);
}
int main(int argc, char *argv[]) {
  /* Read data from input files */
  init_data(argv[1], ....);
  P4A_runtime_copy_to_accel(pos, ...*sizeof(...));
  /* Main temporal moop */
  for(t = 0; t < T; t+=DT)
    iteration(...);
  /* Output results to a file */
  P4A_runtime_copy_from_accel(pos, ...*sizeof(...));
  write_data(argv[2],....);
}
```

Fig. 4. Simplified code for functions `discretization` and `main` after interprocedural communication optimization

4.4 Runtime Library

Our compiler Par4All includes a lightweight runtime library that allows to abstract from the target (OpenCL and cuda). It also supports common functions such as memory allocation at kernel call and memory transfer sites. It maintains a hash table that maps host addresses to gpu addresses. This allows flexibility in the handling of the memory allocations. Using it, the user call sites and his function signatures can be preserved.

The memory management in the runtime does not free the gpu buffers immediately after they have been used, but preserves them as long as there is enough memory on the gpu. When a kernel execution requires more memory than is available, the runtime frees some buffers. The policy used for selecting a buffer to free can be the same as for cache and virtual memory management, for instance lru or lfu.

Notice in Fig. 4 the calls to the runtime that retrieve addresses in the gpu memory space for arrays `pos` and `data`.

5 Experiments

We ran several experiments using the Polybench Suite, Rodinia, and *Stars-PM*. The Rodinia's tests had to be partially rewritten to match Par4All coding guidelines. Specifically, we performed a one-for-one replacement of C89 array definitions and accesses with C99 variable length array constructs. This was necessary for the automatic parallelization process of Par4All to succeed and provide good input to our communication optimization later in the compilation chain. All benchmarks presented here, including *Stars-PM*, are available with the current development version of Par4All [15].

5.1 Metric

The first question is: what should we measure? While speedups in terms of CPU and wall clock time are most important to users if not to administrators, many parameters impact the results obtained. Let us consider for example the popular *hotspot* benchmark [16]. Its execution time depends on two parameters: the matrix size and the number of time steps. In the general context of GPU the matrix size should be large enough to fully load the GPU. In the context of this paper the time step parameter is at least as important since we move data transfers out of the time step loop. Figure 6 shows how *hotspot* is affected by the number of time step iterations and approaches an asymptote, acceleration ranges from 1.4 to 14. The single speedup metric is not enough to properly evaluate our scheme.

We believe that a more objective measurement for evaluating our approach is the number of communications removed and the comparison with a scheme written by an expert programmer. Focusing on the speedup would also emphasize the parallelizer capabilities.

We propose to count the number of memory transfers generated for each version of the code. When the communication occurs in a loop this metric is parametrized by the surrounding loop iteration space. For instance many benchmarks are parametrized with a number of time steps t, thus if 3 memory transfers are present in the time step loop, and 2 are outside of the loop, the number of communication will be expressed as $3 \times t + 2$. Sometimes a loop that iterate over a matrix dimension cannot be parallelized, either intrinsically or because of limited capacity of the compiler. Memory transfers in such loop have a huge performance impact. In this case we use n to emphasize the difference with the time step iteration space.

5.2 Measurements

Figure 7 shows our results on 20 benchmarks from Polybench suite, 3 from Rodinia, and the application *Stars-PM* (§ 3). Measurements where performed on a machine with two Xeon Westmere X5670 (12 cores at 2.93 GHz) and a NVIDIA GPU Tesla C2050. The OpenMP versions used for the experiments are generated automatically by the parallelizer and are not manually optimized.

Kernels are exactly the same for the two automatically generated versions using Par4All. We used the NVIDIA CUDA SDK 4.0, and GCC 4.4.5 with `-O3`.

For Polybench, we forced Par4All to ignore array initializations because they are both easily parallelized and occur before any of the timed algorithms. Our normal optimizer configuration would thus have "optimized away" the initial data transfer to the GPU within the timed code. The measurements in Fig. 7 includes all communications.

Some of the Polybench test cases use default sizes that are too small to amortize even the initial data transfer to the accelerator. Following Jablin *et al.* [18], we adapted the input size to match capabilities of the GPU.

For the cosmological simulation, the communication optimization speeds up execution by a factor of 14 compared to the version without our optimization, and 52 compared to the sequential code.

We include results for HMPP and PGI. Only basic directives were added by hand. We didn't use more advanced options of the directives, thus the compiler doesn't have any hints on how to optimize communications.

The geometric mean over all test cases shows that this optimization improves by a 4 to 5 factor over PAR4ALL, PGI and HMPP naïve versions.

When counting the number of memory transfers, the optimized code performs very close to a hand written mapping. One noticeable exception is *gramschmidt*. Communications cannot be moved out of any loop due to data dependencies introduced by some sequential code. Some aggressive transformations in the compiler might help by accepting a slower generated code but allowing data to stay on the GPU. This would still be valuable if the slowdown is significantly lower than the communication overhead. The difficulty for the compiler is to evaluate the slowdown and to attempt parallelization only if optimized communications lead to a net performance increase. This is beyond our current capabilities.

5.3 Comparison with Respect to a Fully Dynamic Approach

While our aim has been to resolve the issue of communication optimization at compile time, other approaches are addressing it entirely at runtime. This is the case for the *StarPU* [4] library.

We took an example included with the PAR4ALL distribution and rewrote it using *StarPU* in order to evaluate the overhead of the dynamic management of communication compared to our static scheme. This example performs 400 iterations of a simple *Jacobi* scheme on a 500×500 pixels picture loaded from a file and stores the result in another file.

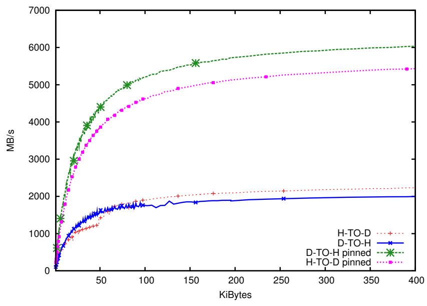


Fig. 5. Bandwidth for memory transfers over the PCI-express bus by data size

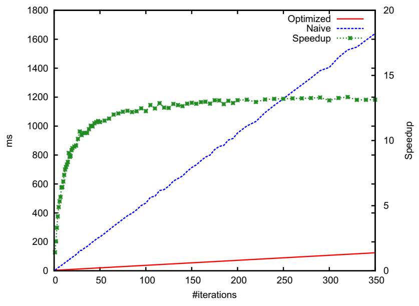
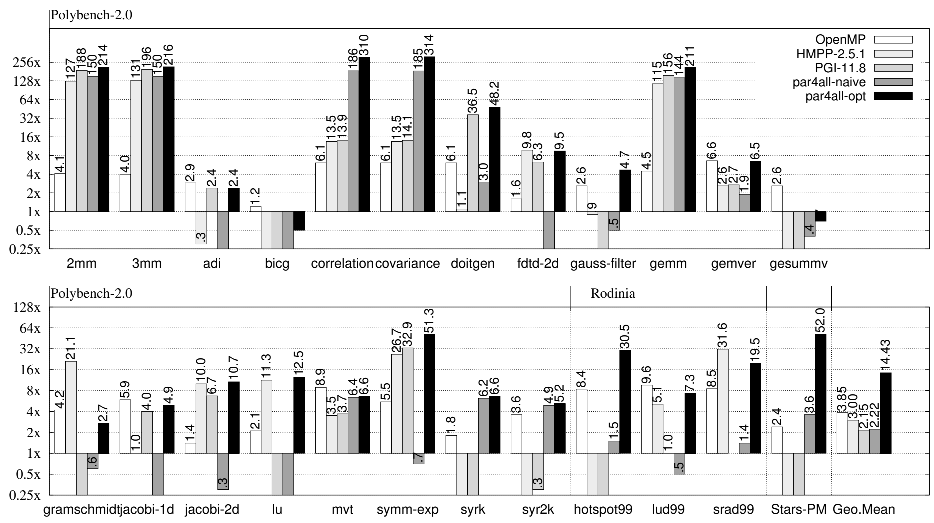


Fig. 6. Execution times and speedups for versions of hotspot on GPU, with different iterations counts



	2mm	3mm	adi	atax	bicg	correlation	covariance	doitgen	fdtd-2d	gauss-filter	gemm	gemver	gesummv	gramschmidt	jacobi-1d	jacobi-2d	lu	mvt	symm-exp	syrk	syr2k	hotspot99	lud99	srad99	Stars-PM
Naïve	8	12	15t +9nt	6+ 4n	6+ 4n	18	15	5	13t	7	4	17	7	9n	6t	6t	4n	8	6n	5	6	5t	4n	24t	25t
Opt.	5	6	4	3	5	2	2	3	6	4	4	10	3	4n +1	3	3	2	7	4	3	4	3	2	t	3
Hand	4	5	4	3	5	2	2	3	6	4	4	10	3	3	3	3	2	7	4	3	4	3	2	t	3

Fig. 7. Speedup relative to naïve sequential version for an OpenMP version, a version with basic PGI and HMPP directives, a naïve CUDA version, and an optimized CUDA version, all automatically generated from the naïve sequential code. Tables show the number of memory transfers.

We were able to confirm that *StarPU* removed all spurious communications, just as our static scheme does. The manual rewrite using *StarPU* and a GPU with CUDA offers a 4.7 speedup over the sequential version. This is nearly 3 times slower than our optimized scheme, which provides a 12.8 acceleration.

Although *StarPU* is a library that has capabilities ranging far beyond the issue of optimizing communications, the overhead we measured confirmed that our static approach is relevant.

6 Related Work

Among the compilers that we evaluated § 2.1, none implements such an automatic optimization. While Lee *et al.* address this issue [20, §.4.2.3], their work is limited to liveness of data and thus quite similar to our unoptimized scheme.

Our proposal is independent of the parallelizing scheme involved, and is applicable to systems that transform OpenMP in CUDA or OpenCL like OMPCuda [21] or OpenMP to GPU [20]. It's also relevant for directives-based compiler, such as JCuda and *hi*CUDA [14]. It would also complete the work done on OpenMPC [19] by not only removing useless communications but moving them up in the call graph. Finally it would free the programmer of the task of adding directives to manage data movements in HMPP [5] and PGI Accelerator [23].

In a recent paper [18], Jablin *et al.* introduce CGCM, a system targeting exactly the same issue. CGCM, just like our scheme is focused on transferring full allocation units. While our granularity is the array, CGCM is coarser and considers a structure of arrays as a single allocation unit. While our decision process is fully static, CGCM takes decisions dynamically. It relies on a complete runtime to handle general pointers to the middle of any heap-allocated structure, which we do not support at this time. We obtain similar overall results, and used the same input sizes. Jablin *et al.* measured a less-than 8 geometric mean speedup vs. ours of more than 14. However, a direct comparison of our measurement is hazardous. We used GCC while Jablin *et al.* used *Clang*, and we made our measurements on a Xeon Westmere while he uses an older Core2Quad Kentsfield. He optimzed the whole program and measured wall clock time while we prevent optimization accross initialization functions and excluded them from our measures. Finally, he used a LLVM PTX backend for GPU code generation, while we used NVIDIA NVCC compilation chain. The overhead introduced by the runtime system in CGCM is thus impossible to evaluate.

7 Conclusion

With the increasing use of hardware accelerators, automatic or semi-automatic transformations assisted by directives take on an ever greater importance.

We have shown that the communication impact is critical when targeting hardware accelerators for massively parallel code like numerical simulations. Optimizing data movements is thus a key to high performance.

We introduced an optimization scheme that addresses this issue, and we implemented it in PIPS and Par4All.

We have experimented and validated our approach on 20 benchmarks of the Polybench 2.0 suite, 3 from Rodinia, and on a real numerical simulation code. The geometric mean for our optimization is over 14, while a naïve parallelization using CUDA achieves 2.22, and the OpenMP loop parallelization provides 3.9. While some benchmarks are not representative of a whole application, we measured on a real simulation an acceleration of 12 compared to a naïve parallelization and 8 compared to an OpenMP version on two 6-core processors. We found that our scheme performs very close to a hand written mapping.

We plan to improve the cache management in the runtime. We can go further than classic cache management algorithms because, unlike hardware cache, our runtime is software managed and can be dynamically controlled. Data flow analyses provide knowledge on the potential future execution of the program.

This can be used in metrics to choose the next buffer to free from the cache. Cheap computations unlikely to be used again should be chosen first. Costly results that are certain to be used should be freed last.

The execution times measured with multi-core processors show that attention should be paid to work sharing between hosts and accelerators rather than keeping the host idle during the completion of a kernel. Multi-core and multi-GPU configurations are another track to explore, with new requirements to determine optimal array region transfers and computation localization.

Acknowledgments. We are grateful to Béatrice CREUSILLET, Pierre JOUVELOT, and Eugene RESSLER for their numerous comments and suggestions which helped us improve our presentation, to Dominique AUBERT who let us use his cosmological simulation, and to Thomas JABLIN for his collaboration and comments.

References

1. Amini, M., Ancourt, C., Coelho, F., Creusillet, B., Guelton, S., Irigoin, F., Jouvelot, P., Keryell, R., Villalon, P.: PIPS is not (just) polyhedral software. In: 1st International Workshop on Polyhedral Compilation Techniques, Impact (in Conjunction with CGO 2011) (April 2011)
2. Ancourt, C., Coelho, F., Irigoin, F., Keryell, R.: A linear algebra framework for static High Performance Fortran code distribution. Scientific Programming 6(1), 3–27 (1997)
3. Aubert, D., Amini, M., David, R.: A Particle-Mesh Integrator for Galactic Dynamics Powered by GPGPUs. In: Allen, G., Nabrzyski, J., Seidel, E., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) ICCS 2009, Part I. LNCS, vol. 5544, pp. 874–883. Springer, Heidelberg (2009)
4. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.-A.: StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience 23, 187–198 (2011); Special Issue: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 863–874. Springer, Heidelberg (2009)
5. Bodin, F., Bihan, S.: Heterogeneous multicore parallel programming for graphics processing units. Sci. Program. 17, 325–336 (2009)
6. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.H., Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing. In: IEEE International Symposium on Workload Characterization (2009)
7. Chen, Y., Cui, X., Mei, H.: Large-scale FFT on GPU clusters. In: 24th ACM International Conference on Supercomputing, ICS 2010 (2010)
8. Creusillet, B., Irigoin, F.: Interprocedural array region analyses. Int. J. Parallel Program. 24(6), 513–546 (1996)
9. Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., Yelick, K.: Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (2008)
10. Fang, W., He, B., Luo, Q.: Database compression on graphics processors. Proc. VLDB Endow. 3, 670–680 (2010)

11. Feautrier, P.: Parametric integer programming. RAIRO Recherche Opérationnelle 22 (1988)
12. Gerndt, H.M., Zima, H.P.: Optimizing Communication in SUPERB. In: Burkhart, H. (ed.) CONPAR 1990 and VAPP 1990. LNCS, vol. 457, pp. 300–311. Springer, Heidelberg (1990)
13. Gong, C., Gupta, R., Melhem, R.: Compilation techniques for optimizing communication on distributed-memory systems. In: ICPP 1993 (1993)
14. Han, T.D., Abdelrahman, T.S.: hiCUDA: a high-level directive-based language for GPU programming. In: Proceedings of GPGPU-2. ACM (2009)
15. HPC Project. Par4All automatic parallelization, http://www.par4all.org
16. Huang, W., Ghosh, S., Velusamy, S., Sankaranarayanan, K., Skadron, K., Stan, M.R.: Hotspot: acompact thermal modeling methodology for early-stage VLSI design. IEEE Trans. Very Large Scale Integr. Syst. (May 2006)
17. Irigoin, F., Jouvelot, P., Triolet, R.: Semantical interprocedural parallelization: an overview of the PIPS project. In: ICS 1991, pp. 244–251 (1991)
18. Jablin, T.B., Prabhu, P., Jablin, J.A., Johnson, N.P., Beard, S.R., August, D.I.: Automatic CPU-GPU communication management and optimization. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, pp. 142–151. ACM, New York (2011)
19. Lee, S., Eigenmann, R.: OpenMPC: Extended OpenMP programming and tuning for GPUs. In: SC 2010, pp. 1–11 (2010)
20. Lee, S., Min, S.-J., Eigenmann, R.: OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In: PPoPP (2009)
21. Ohshima, S., Hirasawa, S., Honda, H.: OMPCUDA: OpenMP Execution Framework for CUDA Based on Omni OpenMP Compiler. In: Sato, M., Hanawa, T., Müller, M.S., Chapman, B.M., de Supinski, B.R. (eds.) IWOMP 2010. LNCS, vol. 6132, pp. 161–173. Springer, Heidelberg (2010)
22. Pouchet, L.-N.: The Polyhedral Benchmark suite 2.0 (March 2011)
23. Wolfe, M.: Implementing the PGI accelerator model. In: GPGPU (2010)
24. Yan, Y., Grossman, M., Sarkar, V.: JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 887–899. Springer, Heidelberg (2009)

Scheduling Support for Communicating Parallel Tasks

Jörg Dümmler[1], Thomas Rauber[2], and Gudula Rünger[1]

[1] Chemnitz University of Technology, Department of Computer Science,
09107 Chemnitz, Germany
{djo,ruenger}@cs.tu-chemnitz.de
[2] Bayreuth University, Angewandte Informatik II,
95440 Bayreuth, Germany
rauber@uni-bayreuth.de

Abstract. Task-based approaches are popular for the development of parallel programs for several reasons. They provide a decoupling of the parallel specification from the scheduling and mapping to the execution resources of a specific hardware platform, thus allowing a flexible and individual mapping. For platforms with a distributed address space, the use of parallel tasks, instead of sequential tasks, adds the additional advantage of a structuring of the program into communication domains that can help to reduce the overall communication overhead. In this article, we consider the parallel programming model of communicating parallel tasks (CM-tasks), which allows both task-internal communication as well as communication between concurrently executed tasks at arbitrary points of their execution. We propose a corresponding scheduling algorithm and describe how the scheduling is supported by a transformation tool. An experimental evaluation of several application programs shows that using the CM-task model may lead to significant performance improvements compared to other parallel execution schemes.

1 Introduction

Task-based approaches have the advantage to allow a decoupling of the computation specification for a given application algorithm from the actual mapping and execution on the computation resources of a parallel target platforms. Many different variations of task-based programming systems have been investigated. An important distinction is whether the individual tasks are executed sequentially on a single execution resource (called single-processor tasks, S-tasks) or whether they can be executed on multiple execution resources (called parallel tasks or multi-processor tasks, M-tasks). S-tasks are often used for program development in shared address spaces, including single multi-core processors, and allow a flexible program development. Efficient load balancing methods can easily be integrated into the runtime system. Examples for such approaches are the task concepts in OpenMP 3.0 [14], the TPL library for .NET [13], or the KOALA framework [10], which provides adaptive load balancing mechanisms. For distributed address spaces, the main challenge is to obtain a distributed load balancing of tasks with a low communication overhead.

Parallel tasks are typically more coarse-grained than S-tasks, since they are meant to be executed by multiple execution units. Each parallel task captures the computations of a specific portion of the application program and can itself be executed by an arbitrary number of execution resources, resulting in a mixed parallel execution. The execution resources may need to exchange information or data during the execution of a parallel task, and thus each parallel task may also comprise task-internal communication. Parallel tasks may have dependencies which then force a specific execution order. Some of the parallel tasks of the program may be independent of each other and can therefore be executed concurrently to each other. In that case, an important decision of the load balancing is dedicated to the question how many and which execution resources should be assigned to each of the concurrently running parallel tasks.

The interaction between different parallel tasks is captured by input-output relations, i.e., one parallel task may produce output data that can be used by another parallel task as input. This parallel programming model is used, for example, by the Paradigm compiler [17], the TwoL model [19], and many other approaches [1,8,20]. In this article, we consider an extended model which allows communication also between parallel tasks that are executed concurrently. This programming model is called communicating M-tasks (CM-tasks) [7] and provides more flexibility for the parallel task structuring due to an additional kind of interaction between parallel tasks. Interactions can be captured by communication between the parallel tasks at specific communication points. For CM-tasks, it must be ensured that tasks with communication interactions are executed at the same time and not one after another. This restricts the execution order of CM-tasks, but provides the possibility for an efficient organization of the communication between the CM-tasks.

This article concentrates on the scheduling of CM-tasks. In particular, the following contributions are made. A new scheduling algorithm for the execution of CM-task programs is proposed that is able to deal with the specific constraints of the scheduling problem for CM-task graphs. In order to support the programming with CM-tasks and especially the new CM-task scheduling algorithm, we have integrated the scheduling into a compiler framework. The compiler framework generates an efficient executable MPI program from a CM-task specification provided by the application programmer. An experimental evaluation for several complex application programs shows that the scheduling algorithm for CM-tasks can lead to significant performance improvements compared to execution schemes resulting from other schedules. Applications from scientific computing offering a modular structure of different program components can benefit tremendously from the CM-task structure and the programming support. Especially, the separation of specification and execution scheme for a specific hardware leads to a portability of efficiency.

The rest of the article is organized as follows. Section 2 gives an overview of the CM-task programming model and defines the scheduling problem for CM-task graphs. Section 3 proposes a scheduling algorithm and Section 4 describes the integration into a compiler tool. Section 5 presents an experimental evaluation. Section 6 discusses related work and Section 7 concludes the article.

2 CM-Task Programming Model

A CM-task program consists of a set of cooperating CM-tasks. Each CM-task implements a specific part of the application and is implemented to be executable on an arbitrary number of processors. Each CM-task operates on a set of input variables that it expects upon its activation and produces a set of output variables that are available after its termination. Moreover, communication phases can be defined in which data can be exchanged with other CM-tasks (that are to be executed concurrently). Dependencies between CM-tasks based on input-output relations and communication interactions between CM-tasks during their execution are captured by the following relations:

- **P-relation**: A P-relation (precedence relation) from a CM-task A to a CM-task B exists if A provides output data required by B as input before B can start its execution. This relation is not symmetric and is denoted by $A\delta_P B$.
- **C-relation**: A C-relation (communication relation) between CM-tasks A and B exists, if A and B have to exchange data during their execution. This relation is symmetric and is denoted by $A\delta_C B$.

2.1 CM-Task Scheduling Problem

A CM-task program can be described by a CM-task graph $G = (V, E)$ where the set of nodes $V = \{A_1, \ldots, A_n\}$ represents the set of CM-tasks and the set of edges E represents the (C and P) relations between the CM-tasks. The set E can be partitioned into two disjoint sets E_C and E_P with $E = E_P \cup E_C$. E_P contains directed edges representing the P-relations defined between CM-tasks. There is a precedence edge from CM-task A to CM-task B in E_P if an input-output relation from A to B exists. E_C contains bidirectional edges representing the C-relations defined between CM-tasks. An example for a CM-task graph is shown in Fig. [1].

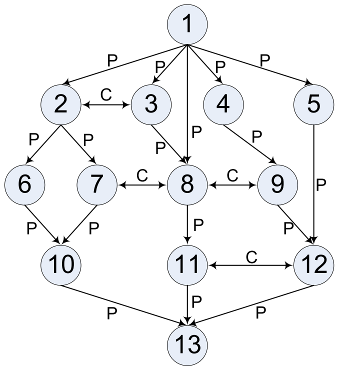


Fig. 1. Example for a CM-task graph with precedence edges (annotation p) and communication edges (annotation c).

The execution times of CM-tasks are assumed to be given by a function

$$T : V \times \{1, \ldots, q\} \to \mathbb{R}$$

where q is the size of the processor set Q of a (homogeneous) target platform. The runtime $T(A, |R|)$ of a CM-task A executed on a subset $R \subseteq Q$ comprises the computation time, the internal communication time, as well as the time for data exchanges with simultaneously running CM-tasks with which A has a C-relation. The data exchange costs are included in the function T because the corresponding communication operations are implemented within the CM-tasks. In practice, T can be an analytical function in closed form determined by curve fitting, see [3,12] for an overview on the performance model. In the following, we assume that T is a non-increasing function in the number of processors used, i.e., using more processors to execute a CM-task does not lead to an increase in execution time. The P-relation edges of the CM-task graph are associated with communication costs

$$T_P : E_P \times \{1, \ldots, q\} \times \{1, \ldots, q\} \to \mathbb{R}$$

where $T_P(e, |R_1|, |R_2|)$ with $e = (A_1, A_2)$ denotes the communication costs between CM-task A_1 executed on set R_1 of processors and CM-task A_2 executed on set R_2 of processors with $R_1, R_2 \subseteq Q$. These communication costs may result from a re-distribution operation that is required between CM-tasks A_1 and A_2 with $A_1 \delta_P A_2$ if $R_1 \neq R_2$ or if $R_1 = R_2$ and A_1 provides its output in a different data distribution as expected by A_2. The re-distribution costs depend on the number of data elements that have to be transmitted from R_1 to R_2 and the communication performance of the target platform.

A schedule S of a given CM-task program maps each CM-task $A_i, i = 1, \ldots, n$, to an execution time interval with start time s_i and a processor set R_i with $R_i \subseteq Q$, i.e.,

$$S : \{A_1, \ldots, A_n\} \to \mathbb{R} \times 2^Q \quad \text{with} \quad S(A_i) = (s_i, R_i).$$

2.2 Scheduling Constraints

The P- and C-relations of a program lead to the following scheduling constraints:

(I) Consecutive time intervals. If there is a P-relation $A_i \delta_P A_j$ between two CM-tasks A_i and A_j, $i, j \in \{1, \ldots n\}, i \neq j$, then the execution of A_j cannot be started before the execution of A_i and all required data re-distribution operations between A_i and A_j have been terminated. Thus, the following condition must be fulfilled:

$$s_i + T(A_i, |R_i|) + T_P\,(e, |R_i|, |R_j|) \leq s_j \qquad \text{with } e = (A_i, A_j).$$

(II) Simultaneous time intervals. If there is a C-relation $A_i \delta_C A_j$, $i, j \in \{1, \ldots, n\}$, $i \neq j$, then A_i and A_j have to be executed concurrently on disjoint sets of processors R_i and R_j, i.e., the following conditions must be fulfilled:

$$R_i \cap R_j = \emptyset \quad \text{and} \quad [s_i, s_i + T(A_i, |R_i|)] \cap [s_j, s_j + T(A_j, |R_j|)] \neq \emptyset.$$

The overlapping execution time intervals guarantee that the CM-tasks A_i and A_j can exchange data during their execution.

(III) Arbitrary execution order. If there are no P- or C-relations between CM-tasks A_i and A_j, $i, j \in \{1, \ldots, n\}, i \neq j$, then A_i and A_j can be executed in concurrent or in consecutive execution order. For a concurrent execution, disjoint processor sets R_i and R_j have to be used, i.e.,

$$\text{if } [s_i, s_i + T_g(A_i, |R_i|)] \cap [s_j, s_j + T_g(A_j, |R_j|)] \neq \emptyset \quad \text{then } R_i \cap R_j = \emptyset$$

where T_g comprises the execution time of A_i as well as the communication time for data exchanges with successively executed CM-tasks A_k with $A_i \delta_P A_k$, i.e.,

$$T_g(A_i, |R_i|) = T(A_i, |R_i|) + \sum_{k=1}^{n} T_P(e, |R_i|, |R_k|) \qquad \text{with } e = (A_i, A_k) \in E_P.$$

In the following, a schedule that meets the constraints (I) - (III) is called *feasible*. A feasible schedule S leads to a total execution time $T_{max}(S)$ that is defined as the point in time when all CM-tasks have been finished, i.e.:

$$T_{max}(S) = \max_{i=1,\ldots,n} \{s_i + T_g(A_i, |R_i|)\}.$$

The problem of finding a feasible schedule S that minimizes $T_{max}(S)$ is called scheduling problem for a CM-task program.

3 Scheduling Algorithm

In this section, we propose a scheduling algorithm for CM-task graphs that works in three phases.

3.1 Transformation of the CM-Task Graph

In the first phase, CM-tasks that are connected by C-relations are identified and combined into larger super-tasks as defined in the following.

Definition 1 (Super-task). *Let $G = (V, E)$ be a CM-task graph. A super-task is a maximum subgraph $\hat{G} = (\hat{V}, \hat{E})$ of G with $\hat{V} \subseteq V$ and $\hat{E} \subseteq E_C$ such that each pair of CM-tasks $A, B \in \hat{V}$ is connected by a path of bidirectional edges in $\hat{E}$.*

Each CM-task and each bidirectional C-relation edge of a CM-task graph belongs to exactly one super-task. A single CM-task without C-relations to any other CM-task forms a super-task by itself. The problem of finding the super-tasks of a CM-task graph is equivalent to discovering the connected components of an undirected graph, considering the C-relations as undirected edges. Using the super-task, the CM-task graph is transformed into a super-task graph as defined next.

Definition 2 (Super-task graph). *Let $G = (V, E)$ be a CM-task graph having l super-tasks $\hat{G}_1 = (\hat{V}_1, \hat{E}_1), \ldots, \hat{G}_l = (\hat{V}_l, \hat{E}_l)$. A super-task graph is a directed graph $G' = (V', E')$ with a set of l nodes $V' = \{\hat{G}_1, \ldots, \hat{G}_l\}$ and a set of directed edges $E' = \{(\hat{G}_i, \hat{G}_j) \mid$ there exists $A \in \hat{V}_i, B \in \hat{V}_j$ with $A \delta_P B\}$.*

Figure 2 (a) shows a super-task graph for the example CM-task graph from Fig. 1. Figure 2 (b)-(d) illustrates the scheduling phases described in the following subsections.

3.2 Load Balancing for Super-Tasks

The second phase is an iterative load balancing algorithm shown in Algorithm 1 that determines a partition of a set of p processors into m subsets where m is the number of CM-tasks inside a specific super-task $\hat{G}$. This is done for different numbers of processors which have to be larger than m and smaller than the total number q of processors. The partition is computed such that the execution time of the entire super-task $\hat{G}$ is at a minimum. The result of the load balancing algorithm is a super-task allocation $L_{\hat{G}}$ for

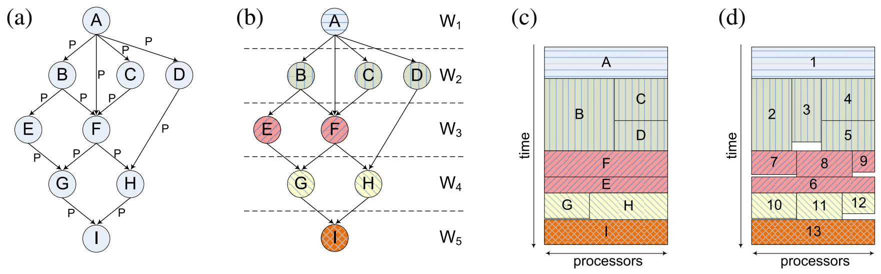


Fig. 2. (a) Super-task graph for the CM-task graph from Fig. 1 with super-tasks $A = \{1\}$, $B = \{2,3\}$, $C = \{4\}$, $D = \{5\}$, $E = \{6\}$, $F = \{7,8,9\}$, $G = \{10\}$, $H = \{11,12\}$, and $I = \{13\}$. (b) Subdivision of the super-task graph into five consecutive layers $W_1 = \{A\}$, $W_2 = \{B,C,D\}$, $W_3 = \{E,F\}$, $W_4 = \{G,H\}$, and $W_5 = \{I\}$. (c) Possible schedule for the super-task graph consisting of a subschedule for each layer of the graph according to Algorithm 2. For super-tasks W_2 and W_4 a division into two groups of processors has been chosen. (d) Schedule for the original CM-task graph including load balancing from Alg. 1.

Algorithm 1: Load balancing for a single super-task

```
1  begin
2      let $\hat{G} = (\hat{V}, \hat{E})$ be a super-task with $\hat{V} = \{A_1, \ldots, A_m\}$;
3      set $L_{\hat{G}}(A_i, m) = 1$ for $i = 1, \ldots, m$;
4      for ($p = m+1, \ldots, q$) do
5          set $L_{\hat{G}}(A_i, p) = L_{\hat{G}}(A_i, p-1)$ for $i = 1, \ldots, m$;
6          choose CM-task $A_k \in \hat{V}$ with maximum value of $T(A_k, L_{\hat{G}}(A_k, p-1))$;
7          increase $L_{\hat{G}}(A_k, p)$ by 1;
```

$\hat{G}$, where $L_{\hat{G}}(A,p)$ specifies how many processors are allocated to CM-task A inside super-task $\hat{G}$ when a total of p processors is available for the super-task $\hat{G}$. Since the number of processors available for super-task $\hat{G}$ is determined not until the next phase of the scheduling algorithm, all possible numbers p of processors are considered, i.e. $m \le p \le q$. The number of processors must be at least m because all m CM-tasks of a super-task have to be executed concurrently to each other.

Algorithm 1 starts with $p = m$ and assigns a single processor to each CM-task of the super-task $\hat{G}$. In each step, the number of available processors is increased by one and the additional processor is assigned to the CM-task A_k having the largest parallel execution time within the current super-task allocation. This usually decreases the execution time of A_k, and another CM-task may then have the largest execution time.

Using the super-task allocation $L_{\hat{G}}$, the cost of a super-task $\hat{G}$ can be calculated by the cost function defined below.

Definition 3 (Costs for super-task graphs). *Let $G=(V,E)$ be a CM-task graph and $G' = (V', E')$ its corresponding super-task graph. A node $\hat{G} = (\hat{V}, \hat{E})$ of G' executed on p processors has costs*

Algorithm 2: Scheduling of the layers of the super-task graph

```
1  begin
2      let $W = \{\hat{G}_1, \ldots, \hat{G}_r\}$ be one layer of the super-task graph $G' = (V', E')$;
3      let $f = \max_{i=1,\ldots,r} |\hat{V}_i|$ be the max. number of CM-tasks in any super-task of $W$;
4      set $T_{min} = \infty$;
5      for ($\kappa = 1, \ldots, \min\{q - f + 1, r\}$) do
6          partition the set of $P$ processors into disjoint subsets $R_1, \ldots, R_\kappa$ such that
               $|R_1| = \max\left\{\left\lceil \frac{P}{\kappa} \right\rceil, f\right\}$ and $R_2, \ldots, R_\kappa$ have about equal size;
7          sort $\{\hat{G}_1, \ldots, \hat{G}_r\}$ such that $T(\hat{G}_i, |R_1|) \geq T(\hat{G}_{i+1}, |R_1|)$ for $i = 1, \ldots, r-1$;
8          for ($j = 1, \ldots, r$) do
9              assign $\hat{G}_j$ to the group $R_l$ with the smallest accumulated execution time
                   and $|R_l| \geq |\hat{V}_j|$;
10         adjust the sizes of the subsets $R_1, \ldots, R_\kappa$ to reduce load imbalances;
11         $T_\kappa = \max_{j=1,\ldots,\kappa}$ accumulated execution time of $R_j$;
12         if ($T_\kappa < T_{min}$) then $T_{min} = T_\kappa$;
```

$$T'(\hat{G}, p) = \begin{cases} \infty & \textit{if } p < |\hat{V}| \\ \max_{A \in \hat{V}} T(A, L_{\hat{G}}(A, p)) & \textit{otherwise.} \end{cases}$$

A directed edge $\hat{e}_{ij} = (\hat{G}_i, \hat{G}_j)$ *with* $\hat{G}_i = (\hat{V}_i, \hat{E}_i)$, $\hat{G}_j = (\hat{V}_j, \hat{E}_j)$, $i \neq j$, *has costs*

$$T'_P(\hat{e}_{ij}, p_i, p_j) = \sum_{e \in RE} T_P(e, L_{\hat{G}_i}(A, p_i), L_{\hat{G}_j}(B, p_j))$$

with $RE = \{e = (A, B) \mid$ *there exists* $A \in \hat{V}_i, B \in \hat{V}_j$ *with* $A \delta_P B\}$.

The cost information is needed for the scheduling algorithm presented next.

3.3 Scheduling of the Super-Task Graph

A super-task graph resembles a standard parallel task graph. However, there is an important difference: the scheduling problem for a super-task graph has the additional restriction that there exists a minimum number of processors which must be assigned to a super-task. This constraint is needed, since it has to be guaranteed that a concurrent execution of all CM-tasks within one super-task is possible.

In the following, we propose a scheduling algorithm for a super-task graph G' that is based on a layer-based scheduling algorithm for parallel tasks [19] and additionally exploits the load balancing information from Algorithm 1. Layer-based scheduling algorithms are well suited for parallel applications consisting of multiple consecutive computation phases. Alternative approaches, such as CPA [16] or CPR [15], try to reduce the length of the critical path of the task graphs. A comparison of layer-based and critical path based scheduling algorithms for parallel tasks shows the advantages of the layer-based ones [5].

The scheduling algorithm proposed consists of several phases. First, the graph G' is partitioned into layers of independent super-task nodes such that the consecutive execution of the layers leads to a feasible schedule for the super-task graph. The partitioning is performed by a greedy algorithm that runs over the super-task graph in a breadth-first manner and puts as many super-tasks as possible into the currently built layer. An illustration is given in Fig. 2 (b). In the second phase, the layers are treated one after another and the scheduling algorithm given in Algorithm 2 is applied to each of them. The goal of the scheduling algorithm is to select a partition of the processor set into κ processor groups. Each of these groups is responsible for the execution of a specific set of super-tasks that are also selected by the scheduling algorithm. An illustration of such a group partitioning and assignment of super-tasks is given in Fig. 2 (c).

The scheduling algorithm for a single layer W with $r = |W|$ super-tasks tests all possible values for the number κ of processor groups with $\kappa \leq r$ and selects the number that leads to the smallest overall execution time (line 5). For a specific value of κ, the set of P processors is partitioned into subgroups such that at least one of the groups is large enough to execute any super-task of W. In particular, at least the largest processor group R_1 contains at least f processors where f denotes the maximum number of tasks which any of the super-tasks contains in its node set $\hat{V}_1, \ldots, \hat{V}_r$ (line 3). If $f \leq P/\kappa$ then a distribution into κ processor sets is chosen (line 6). If $f > P/\kappa$ then one processor group is made large enough to contain exactly f processors and the rest of the processors is evenly partitioned into $\kappa - 1$ processor groups. For the assignment of super-tasks to processor groups, a list scheduling algorithm is employed that considers the super-tasks one after another in decreasing order of their estimated execution time (line 7). The subset R_l of processors for a specific super-task $\hat{G}_j$ is selected such that R_l is large enough to execute $\hat{G}_j$ and assigning $\hat{G}_j$ to R_l leads to the overall smallest accumulated execution time (line 9).

Afterwards, an iterative group adjustment is performed to reduce load imbalances between the subsets of processors (line 10). In each iteration step, two subsets of processors R_i and R_j are identified such that moving a processor from R_i to R_j reduces the total execution time of the layer while R_i is still large enough to execute all super-tasks assigned to it. The procedure stops when there is no such subset left.

4 Programming Support for CM-Task Applications

The parallel programming model of CM-tasks is supported by a compiler framework [6]. The compiler framework expects the following input:

- a platform-independent specification program that describes the task structure of the parallel application, see an example in Fig. 3,
- a machine description that defines a homogeneous target platform by specifying a number of hardware parameters such as the number of processors, the speed of the processors, and the speed of the interconnection network, and
- a set of CM-tasks that are provided as parallel functions to be executed on an arbitrary number of processors, e.g. using *C+MPI*.

```
cmmain pabm (y:svectors:inout:replic) {
 var x,h : scalar; var ort : svectors;
 seq {
  init_step (x,h);
   while (x[0] < X)#100 {
    seq {
     cparfor (k = 0:K-1) {
      pabm_step (k,x,h,y[k],y[K-1],ort); }
     update_step (x,h);
}}}}
```

Fig. 3. Coordination structure for the PABM method using the specification operators

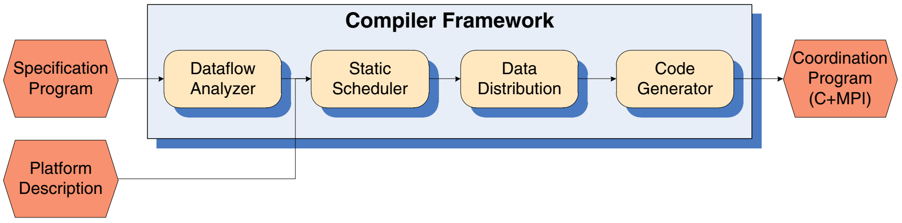


Fig. 4. Overview of the transformation steps of a user-defined specification program and platform description into a *C+MPI* coordination program

A specification program consists of two parts. The first part contains declarations of the parallel functions implemented by the user. Each declaration comprises (i) an interface description with input parameters, output parameters as well as special parameters that are communicated along the C-relations and (ii) a cost formula depending on the number of executing processors and platform-specific parameters whose values are provided in the machine description. The second part describes the coordination structure of the CM-tasks of the program. The coordination structure uses the operators **seq** to define a consecutive execution due to P-relations, **par** to define independent computations that might be executed concurrently or one after another, **cpar** to define a concurrent execution due to C-relations, **for** and **while** to define loops whose iterations have to be executed one after another, **parfor** to define loops whose iterations are independent of each other, **cparfor** to define loops whose iterations have to be executed concurrently, and **if** to define a conditional execution of entire tasks.

Figure 3 shows the coordination structure for a Parallel-Adams-Bashforth-Moulton (PABM) method [21], an implicit solution method for systems of ordinary differential equations. The PABM method performs several time steps that are executed one after another and within each time step, K stage vectors are computed. The CM-tasks utilized are `init_step` to initialize the first time step, `pabm_step` to compute a single stage vector for a single time step and `update_step` to update the step size for the next time step. The term `#100` defines an estimate of the number of iterations of the `while`-loop and can be used to predict the resulting execution time of the application.

The compiler framework performs several transformation steps to translate a specification program into an executable *C+MPI* coordination program that is adapted to a specific parallel platform, see Fig. 4 for an overview. The transformation steps of the framework include the detection of data dependencies, the computation of a platform-dependent static schedule, the insertion of data re-distribution operations, and the final translation into the coordination program. The final coordination program incorporates a schedule determined by the scheduling algorithm from Sect. 3 in which the execution order of independent CM-tasks and the processor groups are given. At runtime, the coordination program is responsible for (i) the actual creation of processor groups and the data exchanges along the C-relations, (ii) the data re-distribution operations to guarantee the correct distribution of input data before starting a CM-task, and (iii) the actual execution of the user-defined parallel tasks. The static scheduler performs the scheduling in the following way:

(1) The specification program is transformed into a set of CM-task graphs. A CM-task graph is constructed for each body of a **for**- or **while**-loop, each branch of the **if**-operator, and for the entire application. The **parfor**- and **cparfor**-loops are unrolled such that different scheduling decisions can be made for each iteration of these loops. The CM-task graphs are organized hierarchically according to the nesting of the corresponding operators. For example, the specification program from Fig. 3 is translated into two CM-task graphs: an upper-level CM-task graph with two nodes representing the function `init_step` and the entire `while`-loop, respectively, and a lower-level CM-task graph for the body of the `while`-loop with $K + 1$ nodes where K nodes represent the instances of function `pabm_step` and one node represents `update_step`.

(2) Next, a feasible CM-task schedule (as defined in Sect. 2.1) is produced for each CM-task graph. The scheduling starts with the CM-task graph representing the entire application and then traverses the hierarchy of CM-task graphs. The scheduling decisions on the upper levels determine the number of processors available on the lower levels. For example, the number of processors assigned to the entire `while`-loop equals the number of available processors when scheduling the loop body. The scheduling for a single CM-task graph is described in Sect. 3.

(3) The resulting CM-task schedules are transformed back and result in a modified specification program. This program includes an additional annotation at each activation of a task that specifies the processor group for the execution. Additionally, the operators of the initial program are modified to reflect the execution order defined by the schedule, i.e., a **par** operator might be transformed into a **seq** operator if a consecutive execution is required. The specific layered structure created by the scheduling algorithm supports the back transformation.

5 Experimental Evaluation

Experimental results have been obtained by applying the scheduling algorithm proposed to complex application benchmark programs from scientific computing. The application benchmarks are executed on two parallel platforms. The **CHiC** cluster consists of 530

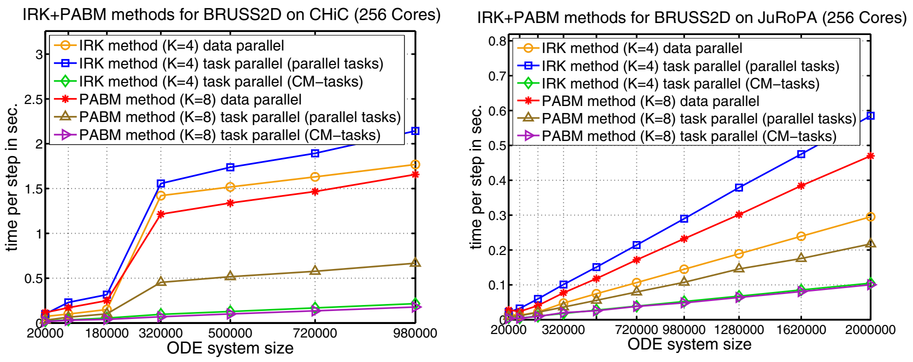
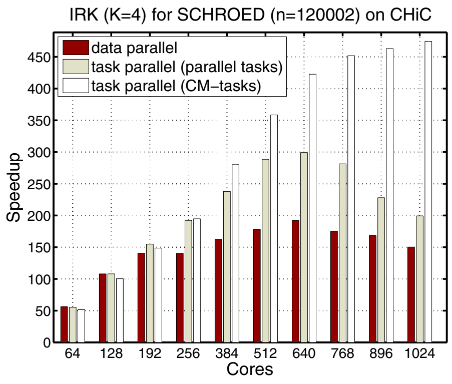


Fig. 5. Measured execution times for a single time step of the IRK method with $K = 4$ stage vectors and the PABM method with $K = 8$ stage vectors on 256 processor cores of the CHiC cluster (left) and the JuRoPA cluster (right).

nodes, each equipped with two AMD Opteron 2218 dual-core processors clocked at 2.6 GHz. The nodes are interconnected by a 10 GBit/s Infiniband network and the MVA-PICH 1.0 MPI library is used. The **JuRoPA** cluster is built up of 2208 nodes, each consisting of two Intel Xeon X5570 (Nehalem) quad-core processors running at 2.93 GHz and the ParaStation MPI library 5.0 is used.

The first set of applications are solvers for systems of ordinary differential equations (ODEs). In particular, we consider the Iterated Runge-Kutta (IRK) method and the Parallel Adams-Bashforth-Moulton (PABM) [21] method. Both methods consist of a large number of time steps that are executed one after another. Each time step computes a fixed number K of stage vectors. Three different parallel implementations are considered: The **data parallel version** computes the K stage vectors of each time step one after another using all available processors and, thus, contains several global communication operations. The **task parallel version** based on standard **parallel tasks** computes the K stage vectors concurrently on K disjoint equal-sized groups of processors. This restricts the task internal communication to groups of processors but leads to additional global communication for the exchange of intermediate results between the processor groups. The **task parallel version** based on **CM-tasks** also computes the stage vectors concurrently but the data exchange between the groups of processors is implemented using orthogonal communication [18]. In the CM-task model, this communication can be modeled using appropriate C-relations. All program versions are implemented in *C* and use the MPI library for communication between the processors.

Figure 5 shows the average execution times of one time step of the IRK and PABM methods for a sparse ODE system that arises from the spatial discretization of the 2D Brusselator equation (BRUSS2D) [9]. The measurements show that a standard data parallel implementation leads to lower execution times compared to standard parallel tasks for the IRK method because additional data re-distribution operations are avoided. For the PABM method, the task parallel version with standard parallel tasks leads to lower runtimes than pure data parallelism because the stage vector computations are decoupled from each other and, thus, much fewer data re-distribution operations are required than for the IRK method. The lowest execution times are achieved by the



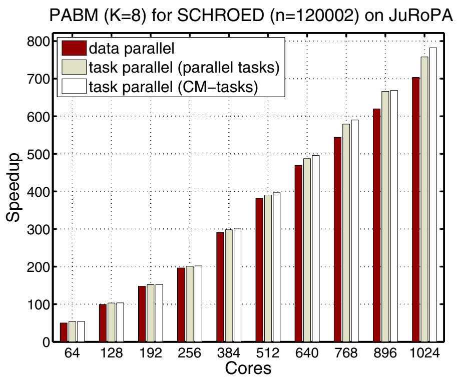
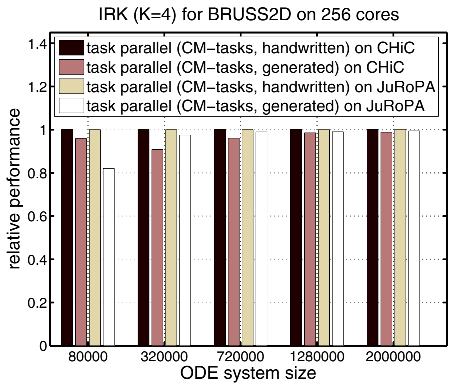


Fig. 6. Speedup values of the IRK method with $K = 4$ stage vectors on the CHiC cluster (left) and of the PABM method with $K = 8$ stage vectors on the JuRoPA cluster (right) for a dense ODE system.



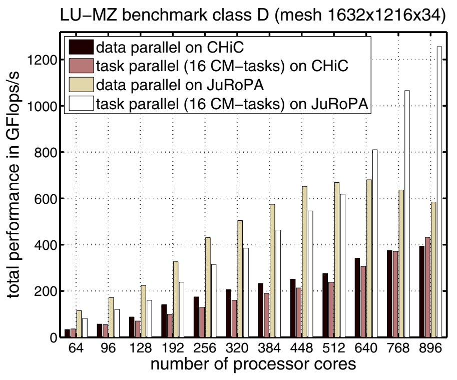


Fig. 7. Relative performance of the generated version of the IRK method compared to a handwritten implementation (left) and performance of the LU-MZ benchmark (right).

task parallel version based on CM-tasks for both, the IRK and the PABM method. For example, the runtime of the data parallel implementation of the IRK method on the CHiC cluster can be reduced to one fifth by employing CM-tasks.

Figure 6 shows the speedup values of the IRK and PABM methods for a dense ODE system that arises from a Galerkin approximation of a Schroedinger-Poisson system. It can be seen that the CM-task version obtains the highest speedup values on both cluster platforms.

Figure 7 (left) compares the performance of the CM-task program version produced by the transformation framework with a handwritten CM-task implementation. The relative performance shown in the figure has been obtained by dividing the average execution time of the handwritten version by the runtime of the generated version. The overhead of the generated version is mainly caused by the data re-distribution operations which are implemented by collective communication in the handwritten version

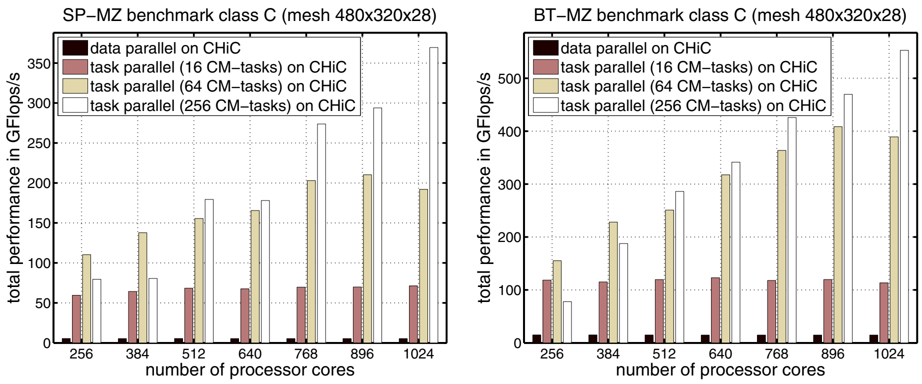


Fig. 8. Performance of the SP-MZ (left) and the BT-MZ (right) benchmarks

and by point-to-point communication in the generated version. The overhead decreases from 20% (for small ODE systems) to under 2% (for large ODE systems) because the share of the coordination code in the total execution time is smaller for larger systems.

The second set of applications is taken from the NAS Parallel Multi-Zone (NAS-MZ) benchmark suite [22]. These benchmarks compute the solution of flow equations on a three-dimensional discretization mesh that is partitioned into zones. One time step consists of independent computations for each zone followed by a border exchange between neighboring zones. Two different implementations are considered: The **data parallel** version uses all processors to compute the individual zones one after another. The **task parallel** version uses a set of CM-tasks each implementing the computations of one zone and C-relations to model the border exchanges between the zones.

Figure 7 (right) shows the measured performance of the LU-MZ benchmark that consists of 16 equal-sized zones leading to 16 equal-sized processor groups in the CM-task implementation. For a lower number of cores, data parallelism leads to a better performance on both platforms due to a better utilization of the cache and the avoidance of communication for the border exchanges. For a higher number of cores, the implementation with CM-tasks shows a much better scalability because the number of cores per zone is smaller leading to smaller synchronization and waiting times.

Figure 8 shows the performance of the SP-MZ and BT-MZ benchmarks which both define 256 zones. We compare program versions with 16, 64, or 256 CM-tasks where each CM-task computes 16, 4, and 1 zones one after another, respectively. Data parallelism is not competitive for these benchmarks, because the individual zones do not contain enough computations to employ a large number of cores. The number of cores per zone is much smaller in the task parallel versions leading to a much higher performance. In the SP-MZ benchmark all zones have the same size and, thus, equal-sized processor groups should be used to execute the CM-tasks. This is only possible when the number of processor cores is a multiple of the number of CM-tasks. In all other cases, load imbalances between the processor groups utilized lead to a degradation of the performance. For example, the program with 256 CM-tasks has almost the same performance on 512 as on 640 processor cores.

The zones in the BT-MZ benchmark have different sizes and, thus, the computation of a schedule that assigns an equal amount of computational work to the processor groups is important. For example, the program version with 256 CM-tasks suffers from load imbalances on 256 cores. These imbalances cannot be eliminated because one core has to be used for each CM-task. For a higher number of cores the CM-task scheduling algorithm is successful in computing a suitable schedule as it is indicated by the high scalability of this program version.

6 Related Work

Several research groups have proposed models for mixed task and data parallel executions with the goal to obtain parallel programs with faster execution time and better scalability properties, see [1,8,20,4] for an overview of systems and approaches. An exploitation of task and data parallelism in the context of a parallelizing compiler with an integrated scheduler can be found in the Paradigm project [17]. The CM-task model considered in this article is an extension of these approaches which captures additional communication patterns.

Many heuristics and approximation algorithms have been proposed for the scheduling of mixed parallel applications based on parallel tasks. Most of these algorithms are based on a two-step approach that separates the computation of the number of processors assigned to each parallel task from the scheduling on specific sets of processors. Examples for such algorithms are TSAS[17], CPR[15], CPA[16], MCPA[2], Loc-MPS[23] and RATS[11]. These algorithms do not capture C-relations between CM-tasks and, thus, have to be adapted before they can be applied to CM-task programs. A work stealing approach for parallel tasks has been presented in [24], but this approach is intended for a shared address space and the concept of C-relations is not required.

7 Conclusions

The performance experiments for typical application programs on different execution platforms with a distributed address space have shown that different CM-task schedules can lead to quite different execution times. The flexible task mapping provided by separating task specification from task scheduling makes it possible to pick the most advantageous scheduling for a specific application on a specific parallel machine. To support the programming with the CM-task approach, we have implemented a transformation tool, which has the advantage that it allows the application programmer to concentrate on the specification of the application and a suitable task decomposition. The proposed scheduling algorithm is an important part of the transformation approach. The overhead of applying the transformation tool is small compared to the performance improvement achievable, thus combining performance efficiency and programmability.

References

1. Bal, H., Haines, M.: Approaches for Integrating Task and Data Parallelism. IEEE Concurrency 6(3), 74–84 (1998)
2. Bansal, S., Kumar, P., Singh, K.: An Improved Two-step Algorithm for Task and Data Parallel Scheduling in Distributed Memory Machines. Parallel Comput. 32(10), 759–774 (2006)
3. Barker, K., Davis, K., Hoisie, A., Kerbyson, D., Lang, M., Pakin, S., Sancho, J.: Using Performance Modeling to Design Large-Scale Systems. IEEE Computer 42(11), 42–49 (2009)
4. Chakrabarti, S., Yelick, K., Demmel, J.: Models and Scheduling Algorithms for Mixed Data and Task Parallel Programs. J. Parallel Distrib. Comput. 47, 168–184 (1997)
5. Dümmler, J., Kunis, R., Rünger, G.: A Scheduling Toolkit for Multiprocessor-Task Programming with Dependencies. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par 2007. LNCS, vol. 4641, pp. 23–32. Springer, Heidelberg (2007)
6. Dümmler, J., Rauber, T., Rünger, G.: A Transformation Framework for Communicating Multiprocessor-Tasks. In: Proc. of the 16th Euromicro Int. Conf. on Parallel, Distributed and Network-Based Processing, PDP 2008, pp. 64–71. IEEE (2008)
7. Dümmler, J., Rauber, T., Rünger, G.: Communicating Multiprocessor-Tasks. In: Adve, V., Garzarán, M.J., Petersen, P. (eds.) LCPC 2007. LNCS, vol. 5234, pp. 292–307. Springer, Heidelberg (2008)
8. Dümmler, J., Rauber, T., Rünger, G.: Mixed Programming Models using Parallel Tasks. In: Dongarra, Hsu, Li, Yang, Zima (eds.) Handbook of Research on Scalable Computing Technologies, pp. 246–275. Information Science Reference (2009)
9. Hairer, E., Nørsett, S., Wanner, G.: Solving Ordinary Differential Equations I: Nonstiff Problems. Springer, Berlin (1993)
10. Hoffmann, R., Rauber, T.: Fine-Grained Task Scheduling Using Adaptive Data Structures. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 253–262. Springer, Heidelberg (2008)
11. Hunold, S., Rauber, T., Suter, F.: Redistribution Aware Two-step Scheduling for Mixed-parallel Applications. In: Proc. of the 2008 IEEE International Conference on Cluster Computing, CLUSTER 2008, pp. 50–58. IEEE (2008)
12. Kühnemann, M., Rauber, T., Rünger, G.: Performance Modelling for Task-Parallel Programs. In: Gerndt, M., Getov, V., Hoisie, A., Malony, A., Miller, B. (eds.) Performance Analysis and Grid Computing, pp. 77–91. Kluwer (2004)
13. Leijen, D., Schulte, W., Burckhardt, S.: The design of a task parallel library. In: Proc. of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2009, pp. 227–242. ACM (2009)
14. OpenMP Application Program Interface, Version 3.0 (May 2008), http://www.openmp.org
15. Radulescu, A., Nicolescu, C., van Gemund, A., Jonker, P.: CPR: Mixed Task and Data Parallel Scheduling for Distributed Systems. In: Proc. of the 15th International Parallel & Distributed Processing Symposium, IPDPS 2001. IEEE (2001)
16. Radulescu, A., van Gemund, A.: A Low-Cost Approach towards Mixed Task and Data Parallel Scheduling. In: Proc. of the International Conference on Parallel Processing, ICPP 2001, pp. 69–76. IEEE (2001)
17. Ramaswamy, S., Sapatnekar, S., Banerjee, P.: A Framework for Exploiting Task and Data Parallelism on Distributed Memory Multicomputers. IEEE Transactions on Parallel Distributed Systems 8(11), 1098–1116 (1997)
18. Rauber, T., Reilein, R., Rünger, G.: Group-SPMD Programming with Orthogonal Processor Groups. Concurrency and Computation: Practice and Experience, Special Issue on Compilers for Parallel Computers 16(2-3), 173–195 (2004)

19. Rauber, T., Rünger, G.: Compiler Support for Task Scheduling in Hierarchical Execution Models. Journal of Systems Architecture 45(6-7), 483–503 (1998)
20. Skillicorn, D., Talia, D.: Models and Languages for Parallel Computation. ACM Computing Surveys 30(2), 123–169 (1998)
21. van der Houwen, P., Messina, E.: Parallel Adams Methods. Journal of Computational and Applied Mathematics 101, 153–165 (1999)
22. van der Wijngaart, R., Jin, H.: The NAS Parallel Benchmarks, Multi-Zone Versions. Tech. Rep. NAS-03-010, NASA Ames Research Center (2003)
23. Vydyanathan, N., Krishnamoorthy, S., Sabin, G., Çatalyürek, Ü., Kurç, T., Sadayappan, P., Saltz, J.: An Integrated Approach to Locality-Conscious Processor Allocation and Scheduling of Mixed-Parallel Applications. IEEE Trans. Parallel Distrib. Syst. 20(8), 1158–1172 (2009)
24. Wimmer, M., Träff, J.: Work-stealing for Mixed-mode Parallelism by Deterministic Team-building. In: Proc. of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2011. ACM (2011)

Polytasks: A Compressed Task Representation for HPC Runtimes[*]

Daniel Orozco[1,2], Elkin Garcia[1], Robert Pavel[1], Rishi Khan[2], and Guang R. Gao[1]

[1] University of Delaware, USA
[2] ET International, USA
{orozco,egarcia,rspavel}@udel.edu, rishi@etinternational.com, ggao@capsl.udel.edu

Abstract. The increased number of execution units in many-core processors is driving numerous paradigm changes in parallel systems. Previous techniques that focused solely upon obtaining correct results are being rendered obsolete unless they can also provide results *efficiently.*

This paper dives into the particular problem of efficiently supporting fine-grained task creation and task termination for runtime systems in shared memory processors.

Our contributions are inspired by our observation of High Performance Computing (HPC) programs, where it is common for a large number of similar fine-grained tasks to become enabled at the same time.

We present evidence showing that task creation, assignment of tasks to processors, and task termination represent a significant overhead when executing fine-grained applications in many-core processors.

We introduce the concept of the *polytask*, wherein the similarity of tasks created at the same time is exploited to allow faster task creation, assignment and termination. The polytask technique can be applied to any runtime system where tasks are managed through queues.

The main contributions of this work are:

1. The observation that task management may generate substantial overhead in fine-grained parallel programs for many core processors.
2. The introduction of the polytask concept: A data structure that can be added to queue-centric scheduling systems to represent groups of similar tasks.
3. Experimental evidence showing that the polytask is an effective way to implement fine-grained task creation/termination primitives for parallel runtime systems in many-core processors.

We use microbenchmarks to show that queues modified to handle polytasks perform orders of magnitude faster than traditional queues in some scenarios. Furthermore, we use microbenchmarks to measure the amount of time spent executing tasks. We show situations where fine-grained programs using polytasks are able to achieve efficiencies close to 100% while their efficiency becomes only 20% when not using polytasks. Finally, we

[*] This research was, in part, funded by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

use several applications with fine granularity to show that the use of polytasks results in average speedups from 1.4X to 100X depending on the queue implementation used.

1 Introduction

The development of processor chip architectures containing hundreds of execution units has unleashed challenges that span from efficient development to efficient execution of applications.

The idea of partitioning computations into *tasks* (or equivalent concepts) has been used by a number of execution paradigms such as pthreads[3], OpenMP [7], Cilk[2] and others to address the efficiency of execution. Our techniques, results and conclusions focus on the aspect of viewing tasks as units of computation, and they are orthogonal to implementation of tasks in particular execution paradigms.

Execution of tasks varies from paradigm to paradigm, and generally includes the use of queues to produce and consume (or execute) tasks as the program progresses. It is a common occurrence in previous research to assume that tasks that become enabled can be executed immediately, overlooking the fact that enqueueing a task to make it available to other threads takes a nonzero time. This assumption is reasonable in systems with few processors or coarse task granularity because the time taken to enqueue a task is negligible compared to the time taken to execute it. However, the enqueueing process can become a significant source of overhead for systems where a large number of processors participate in fine-grained execution.

A simple observation can be used to illustrate the problem: A system that uses a queue (centralized or distributed) for task management where there are P idle processors requires at least P tasks to be enqueued to allow execution in all processors. Even if enough tasks are available, the time to enqueue them becomes relevant as P grows. Advanced algorithms based on distributed structures or trees softens this problem by providing faster primitives due to lower contention, but ultimately they do not intrinsically reduce the total number of queue operations needed.

The overhead of assigning tasks in systems with many processors can be solved by leveraging on a simple observation: *Tasks that become enabled at the same time are frequently very similar* because, in many cases, all tasks direct processors to execute the same instructions, using the same shared data, and only a few parameters and local variables differentiate tasks from one another.

Our contributions are inspired by the idea that similar tasks can be efficiently *compressed* and represented as a single task, that we refer to as a *polytask.* We address the specific case in which tasks can be written in such a way that only a single integer number can be used to retrieve their task-specific data. Many programs have that property: iterations of parallel loops are differentiated by their iteration index, threads in fork-join applications can be differentiated by their thread identifier and so on.

We show that most queue primitives can support compression of tasks if a small data structure containing two integers is added to the task description.

Using the added integers, we show that it is possible to efficiently perform task management on compressed tasks.

The effectiveness of compressing tasks into polytasks is shown in experiments where three traditional queueing techniques are modified to allow compression. Our results show that compression of tasks, when possible, allows much faster queue primitives than their noncompressed counterparts because (1) compressed tasks represent several tasks, making a queue operation on them equivalent to several queue operations on noncompressed tasks and (2) queue operations on compressed tasks are frequently faster than queue operations on regular tasks. We show that queues modified to handle polytasks perform at par with their non-modified versions when compression is not possible.

Our contributions are relevant to queue-centric runtime systems whether they have one queue or many. Other runtime systems that are not necessarily queue centric, such as OpenMP [7] have developed alternatives that are very similar in their implementation and their objective. Our contribution is the presentation of a systematic way to modify queue-centric systems to address task compression.

The advantages of task compression in larger applications are shown in Section 5 using microbenchmarks and applications. All cases present evidence supporting our claims regarding the effectiveness of task compression.

The rest of the paper is organized as follows: Section 2 presents relevant background, Section 3 presents some motivation as well as the specific definition of the problem addressed in this paper, Section 4 represents the core of our paper, presenting our technique for task compression, Section 5 describes our experiments and results, Section 6 presents related work and Section 7 presents conclusions and future work.

2 Background

This section presents background related to our contributions, including a brief summary of the queue algorithms referred throughout the paper, and a description of the processor used for our experiments.

2.1 Queue Algorithms

The central work of this paper tries to enhance the capabilities of existing queue algorithms for the particular case of task management. A large number of queue algorithms exist, for example, Shafiei [16] shows a survey summarizing the relative advantages of many different algorithms. We have chosen three queue algorithms that cover a significant portion of the design space.

The first algorithm, which we refer to as the SpinQueue algorithm, uses a linked list as the basic data structure for the queue and a spinlock to avoid data races on processors accessing the queue. The SpinQueue is a simple implementation, that is easy to understand, easy to program, and that offers excellent performance if there is low contention at the queue. The simple implementation of the SpinQueue makes it suitable for quick development of parallel applications.

The MS-Queue is an advanced non-blocking algorithm, presented by Michael and Scott [12], that uses a Compare and Swap operation to allow concurrent operations on the queue. The MS-Queue algorithm has become an industry *de facto* standard, being used in the Java Concurrency Constructs, and in many other high-profile implementations.

The MC-Queue is a queue algorithm presented by Mellor-Crummey[11] that distributes queue operations over a group of nonblocking queues to maximize performance. In the MC-Queue, the queue structure is composed of several independent nonblocking queue implementations. When queue operations are requested, an atomic addition is used to select one of the available nonblocking queues, effectively distributing the operations across them. The MC-Queue is an excellent choice for applications where a large number processors attempt to execute operations concurrently.

2.2 Cyclops-64 Architecture

Cyclops-64 (C64) is a processor developed by IBM. The architecture and features of C64 have been described extensively in previous publications [6,5].

Each C64 chip has 80 computational cores, no automatic data cache, and 1GB of addressable memory. Each core contains two single-issue thread units, 60KB of user-addressable memory that is divided into stack space and shared memory, a 64 bit floating point unit, and one memory controller.

One of the main features of the C64 chip is that *memory controllers can execute atomic operations*. In C64, each memory controller contains its own Arithmetic and Logical Unit that allows the memory controller to execute integer and logical atomic operations *in memory* without the intervention of a processor or a thread unit. Atomic operations in C64 take 3 cycles to complete at the memory controller. All memory controllers in C64 have the capability to execute atomic operations.

Under the default configuration, C64 has 16KB of stack space for each thread unit, 2.5MB of shared on-chip memory, and 1GB of DRAM memory.

3 Motivation

The difficulties in traditional task management as well as the possibilities of task compression can be illustrated using the kernel of a simulation of an electromagnetic wave propagating using the Finite Difference Time Domain algorithm in 1 Dimension (FDTD1D), shown in Figure 1.

The parallel loops in Figure 1 can be efficiently executed in a many-core processor such as C64 if the iterations in the parallel loops are expressed as tasks. The granularity of the execution can be varied through the tile size (`TileSize` in Figure 1). A small tile size will result in finer grain and more parallelism, but it will also incur a higher runtime system overhead because of the additional burden in task management.

The main problem is that *fine-grained* execution is difficult: The granularity of tasks is limited by the overhead of task management. In general, fine grain execution is useful only when the overhead associated with the execution

```
// TileSize controls the granularity
#define TileSize 16
void FDTD1D( double *E, double *H, int N,
  int Timesteps,
  const double k1, const double k2 )
{
  int i, t;
  for ( t=0;t<Timesteps;t++ )
  {
    parallel for (i=1; i<N/TileSize; i++)
    {
      E_Tile( i, E, H, N, k1, k2 );
    }

    parallel for (i=1; i<N/TileSize; i++)
    {
      H_Tile( i, E, H, N, k1, k2 );
    }
  }
}
```

Fig. 1. FDTD1D Kernel

```
void E_Tile( int TileID,
  double *E, double *H, int N,
  int Timesteps,
  const double k1, const double k2 ) {
  int i, Start, End;
  Start = TileID * TileSize;
  End = Start + TileSize;
  for ( i = Start; i < End; i++ )
    E[i]=k1*E[i]+k2*(H[i]-H[i-1]);
}

void H_Tile( int TileID,
  double *E, double *H, int N,
  int Timesteps,
  const double k1, const double k2 ) {
  int i, Start, End;
  Start = TileID * TileSize;
  End = Start + TileSize;
  for ( i = Start; i < End; i++ )
    H[i]+=E[i]-E[i+1];
}
```

Fig. 2. FDTD1D Compute Tiles

is acceptable. In contrast, coarse-grained executions decrease the proportional overhead of task management at the cost of reducing parallelism and reducing the opportunities for load balancing in many-core systems [9].

Traces of several executions of FDTD1D, executed using the TIDeFlow runtime system [13] with the SpinQueue algorithm in C64 were obtained to show the limits in granularity:

Table 1. Task duration and enqueueing overhead for C64 (SpinQueue Algorithm)

Process	Cycles	Process	Cycles
Enqueue one task	6200	Enqueue a task for each processor	9.9×10^5
Execute a tile of size 1	180	Execute a tile of size 256	16000
Execute a tile of size 1024	56000	Execute a tile of size 16384	9.0×10^5
Execute a tile of size 65536	3.6×10^6		

Table 1 exposes the problem faced by programs with fine granularity: If the tile size is set to 1, approximately 10^6 cycles will be required to enqueue one task for each one of the 160 processors in C64 while a single processor executes one task in only 180 cycles. The conclusion is that programs where tiles take less than 10^6 cycles to execute result in poor performance because processors will consume tasks faster than they are written to the queue.

The solution that we pursue in this paper is based on the observation that in many programs, tasks are very similar, and they may be compressed into a single task to reduce the total time for task creation. Task compression opens a number of questions that we intend to address.

3.1 Problem Formulation

The following question summarizes our research goals:

> How is it possible to exploit the commonly found similarities between tasks created at the same time to achieve efficient representation of tasks?

The main question opens related questions:

– How can several, similar tasks be efficiently compressed into a polytask?
– Is it possible to concurrently and efficiently extract a task from a polytask?
– Is it possible to efficiently support termination operations such as *join* when using compressed task representations such as polytasks?
– Does task compression introduce additional overheads in applications where tasks are dissimilar making task compression unnecessary?

We intend to fully answer these questions in the following sections. We also provide experimental results to back our claims regarding the usability of polytasks.

4 Polytasks: Efficient Building Blocks for Runtime Systems

Creation of similar tasks at the same time is common in scientific programs using execution models that support parallel execution of loops. Figure 1 shows one such example where a parallel loop results in creation of a large number of similar tasks, that execute the same instructions, that access the same global variables and that are distinguished only by their loop index, or rather, by their *execution instance.*

The main idea in our proposed solution is to represent all the tasks related to a parallel loop with a single data structure that we call a polytask. A polytask is a data structure that includes all the information commonly found in a task plus additional information describing the number of tasks it represents (N, the number of iterations in the parallel loop) and their state.

```
typedef struct task_s {
  // Task Information:
  // Environment information
  // Code to be executed
  ...
} task_t
```

Fig. 3. Original task data structure

```
typedef struct polytask_s {
  int TasksAvailable;
  int TasksPending;

  // Task Information:
  // Environment information
  // Code to be executed
  ...
} polytask_t
```

Fig. 4. Polytask data structure

A generic structure that can be used to represent a single, particular task is presented in Figure 3. The structure of Figure 3 has been upgraded (Figure 4) to allow representation of several, similar tasks as a polytask. The polytask structure contains a counter that specifies the number of tasks that are available in the polytask (TasksAvailable) and a counter containing the number of tasks that have not finished (TasksPending). The TasksPending counter facilitates implicit join operations at the end of parallel loops. At initialization, both counters are initialized to N to indicate that N tasks are available and that none of them have finished execution.

Note that any queue algorithm can be used to implement polytasks operations: A polytask can be enqueued into the work queue in the same way as a single task can be enqueued. The algorithm in Figure 6 shows how to upgrade a generic runtime system to support polytasks by introducing three operations for task management: PolyEnqueue is used to create N tasks of a particular type. PolyDequeue is used to extract the next available task from the queue and TaskCompleted is used to count the number of tasks that have finished execution.

A significant advantage of polytasks over individual tasks is that a polytask is enqueued *once* into the work queue. PolyEnqueue, when called, initializes the information in the task structure to specify that N tasks must be created and N tasks must complete. PolyEnqueue enqueues a single queue item containing the polytask to the queue. In general, only minimum modifications to the data structures and to the original enqueueing algorithm are required to support polytasks.

Extracting individual tasks from a polytask is more challenging because an unmodified dequeue operation will remove the polytask from the queue. Instead, PolyDequeue extracts a single task from the polytask at the head of the queue using an atomic decrement on its TasksAvailable counter. The value returned by the atomic decrement is used as the *execution instance* for the execution of the task (e.g. iteration index of a parallel loop) if it is positive. It is possible to obtain invalid (non-positive) execution instances during the extraction process because the atomic decrement is a concurrent process. This is not a problem since polytasks are quickly removed from the queue after all their available tasks have been claimed, effectively presenting a new polytask at the head of the queue. Processors that did not obtain valid execution instances during their fist attempt can retry until the polytask at the head of the queue contains enough available tasks.

Execution instances are assigned to processors starting from N and going down to 1. When a processor extracts the last task from the polytask, (*i.e.* when the execution instance obtained is 1), the processor dequeues the polytask from the queue using the original queue algorithm selected.

TaskCompleted implements *join* behavior for tasks in a particular polytask. The TasksPending counter in the polytask structure can be used as a synchronization point to reliably know the number of tasks that have not finished execution in a particular polytask. Processors atomically decrement the TasksPending counter to indicate termination of individual tasks in the polytask. A processor can know if it executed the last task in the polytask (*i.e.* the join operation

```
/* ---  Queue Variables --- */
typedef struct QueueItem_s {
  polytask_t PolyTask;
  // Other Queue-Specific Members
  ...
}
QueueItem_t;

QueueItem_t  *Head, *Tail;

/* --- PolyTask Functions --- */
task_t * PolyDequeue( void ) {
StartDequeue:
  polytask_t PolyTask = Head->PolyTask;
  int ExecutionInstance =
    Atomic_Decrement(
      &( PolyTask->TasksAvailable ) );

  if ( ExecutionInstance == 1 ) {
    // Removes Task from Queue
    Dequeue();
  }

  if ( ExecutionInstance > 0 ) {
    return( PolyTask );
  }

  goto StartDequeue;
}
```

Fig. 5. Polytask Dequeue

```
void PolyEnqueue( PolyTask_t *PolyTask,
      int N )
{
  PolyTask.TasksAvailable = N;
  PolyTask.TasksPending = N;
  Enqueue( PolyTask );
}

void TaskCompleted( PolyTask_t *PolyTask )
{
  int PendingTasks =
    Atomic_Decrement(
      &( PolyTask->TasksPending ) );

  if ( PendingTasks == 1 ) {
    // This is the last task
    // Join operation successful
    ....
  }
}
```

Fig. 6. Polytask Enqueue and Complete Task operations

is complete) by inspecting the return value of the atomic decrement. Figure 6 shows the structure of the `TaskCompleted` function and how it can be used to handle the behavior of join operations by the runtime system.

Polytask operations are faster than traditional task operations on queues. When task compression is possible ($N > 1$), they provide flexible mechanisms for synchronization and continuation of tasks. The speed up will be significant, especially when fine granularity is used and the number of processors increase. The synchronization between tasks of the same type (*i.e.* in a polytask) is easier and allows smooth support for control flow mechanisms such as creation, termination and continuation found in parallel loops. The reasons for faster operations and easier synchronization and continuation are:

– **Task Creation:** N tasks can be written with a single queue operation.
– **Task Assignment:** In most cases, the concurrent part of the algorithm that assigns a single task to a processor is a single atomic decrement, that can be executed in parallel by all processors with very little contention. This is an improvement over a single task-based approach where the algorithm to access the queue is based on locks and it could be better even on queues using Compare-and-Swap operations. However, on C64, atomic decrements are faster than compare and swap operations because atomic decrements can be executed directly *in memory* by the memory controller (see section 2.2).

- **Join and Continuation Operations:** The use of a single counter (`TasksPending`) modified through atomic operations allows fast task synchronization during the termination phase of parallel loops. The decentralized nature of the count, where any processor may be the last one, reduces the overhead of other implementations where a particular, centralized process is responsible for the termination and continuation of the parallel loop.

Section 5 uses microbenchmarks and some applications to show the advantages of polytasks over traditional queue techniques.

5 Experiments

The effectiveness of compressing tasks into polytasks is analyzed in this section. We have designed experiments that show the effect of task compression in an isolated scenario –using microbenchmarks– and in a full production runtime system –using full applications–. Our results show that applications with many similar tasks greatly benefit from the use of polytasks, without adversely affecting applications without good task similarity.

We have chosen C64, a many-core processor architecture (Section 2.2) as the testbed architecture because it is a logical choice to support task-based runtime systems due to its large number of processors in a shared memory environment and its non-preemptive execution model.

All of our experiments where written in C and they were compiled with ET International's C64 C compiler, version 4.3.2, with compilation flags -O3. We ran all of our experiments using FAST[5], a highly accurate C64 simulator.

5.1 Microbenchmarks

In our first study, we analyze the advantages and disadvantages of polytasks in a controlled environment that attempts to show the behavior of the required queue primitive operations without external perturbations.

In our first set of experiments, we isolate the behavior of queue operations by running programs where all processors in a C64 chip continuously produce and consume randomly generated tasks without executing them. In these experiments, each processor produces 512 tasks and consumes 512 tasks. Embedded hardware performance counters are used to collect timing data. To illustrate the effectiveness of polytasks, the number of tasks that are similar in each experiment was modified. A **task similarity** of N (See Figures 7 and 8) indicates that groups of N tasks are similar and can be potentially compressed into a single polytask. Our experiments show results that range from task similarity of 1, where each task is unique, to task similarity of 256, where groups of 256 tasks are compressed into a single polytask. When the results are reported, enqueueing one polytask that contains N tasks is considered equivalent to enqueueing N tasks directly because a runtime system will observe, in both cases, that N tasks have been created.

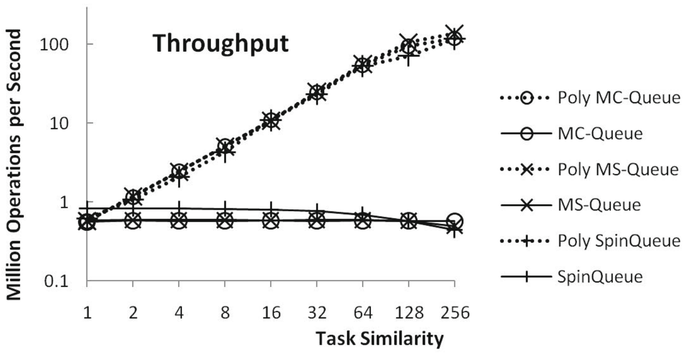


Fig. 7. Effect of polytask compression on throughput

We have explored the sensitivity of polytasks to the underlying queueing algorithm used. From the many queue algorithms that exist (Shafiei [16] has compiled an excellent summary) we have chosen three algorithms that cover a significant portion of the design space, a spinlock-based queue (SpinQueue, the simplest to implement), the MS-Queue algorithm[12] (the most famous nonblocking algorithm) and the MC-Queue algorithm[11] (a distributed queue with very high performance). We implemented polytasks on each one of the selected queue algorithms in an effort to present quality over quantity.

Figures 7 and 8 show the results of our experiments. Inspection of the figures allows us to reach important conclusions:

– Polytasks increase the performance of runtime operations, both in total (aggregated) operations per second in the whole system as well as in terms of reducing the latency of individual operations.
– The overhead of polytask compression is very small. Both in terms of latency and throughput there is not a significant overhead (less than 2% in all but one case) when polytasks are used in situations where tasks are not similar.
– The advantages of polytasks are not dependent on the queue algorithm used to implement the polytask operations: All queue algorithms tested present excellent performance gains, and very low overhead.
– When task similarity is high, the average performance of task operations is improved by up to two orders of magnitude. The reason for the increase in performance is that calls to polytask operations that do not result in calls to queue operations finish quickly.

The results shown in Figures 7 and 8 show that polytasks can benefit applications where task compression is possible without degrading the performance of applications where task compression is not possible. The second set of experiments was designed to show the advantages of the use of polytasks in applications with varying levels of granularity. To avoid external perturbations, synthetic tasks that execute a delay loop of varying duration were used.

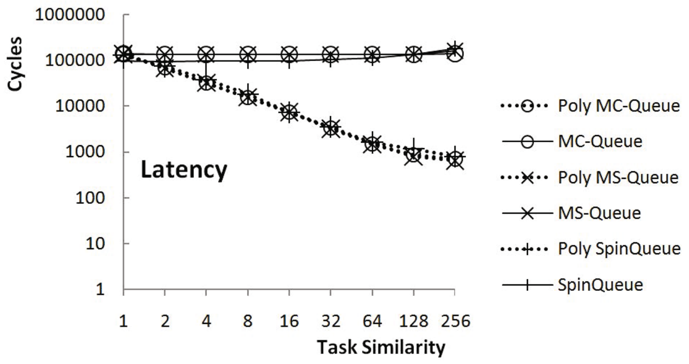


Fig. 8. Effect of polytask compression on latency

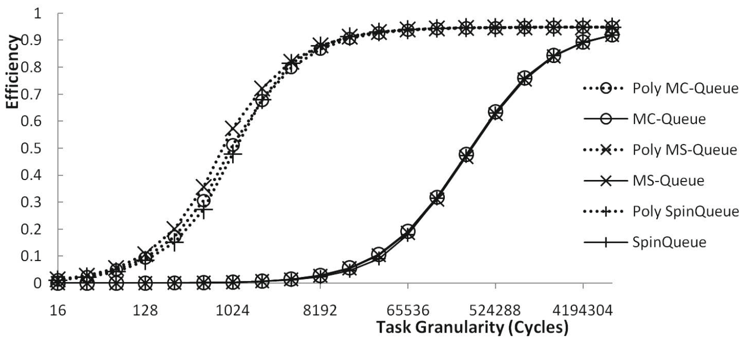


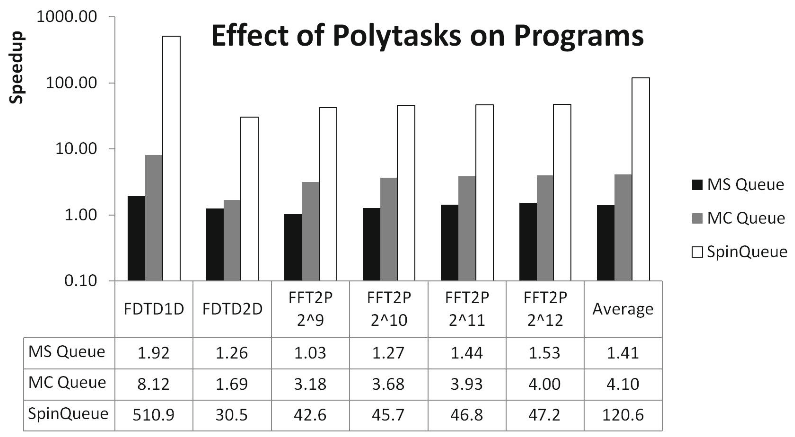
Fig. 9. Advantages of polytasks as a function of program granularity

Figure 9 shows the results of our experiments. In the figure, the efficiency is the fraction of time that the processor spends executing tasks, and it is calculated as the ratio between the time executing tasks and the total execution time, including overheads. The Task Granularity is the duration of the tasks executed. Each data point reported in the figure was obtained by running 40960 tasks that execute a delay loop whose duration is specified in the figure as Task Granularity.

Our results show that fine grained applications greatly benefit from the use of polytasks. Polytasks enable greater runtime system efficiencies at very fine grain synchronization while traditional approaches only allow coarser grain synchronization.

As is to be expected, if the application uses very coarse grained parallelism the burden of task management does not affect the efficiency of the system because the time to perform a queue operation will not be significant when compared to the execution time of a task.

The results of Figure 9 show that the advantages of polytasks under varying task granularity remain, independent of the queue algorithm used. In our



	FDTD1D	FDTD2D	FFT2P 2^9	FFT2P 2^10	FFT2P 2^11	FFT2P 2^12	Average
MS Queue	1.92	1.26	1.03	1.27	1.44	1.53	1.41
MC Queue	8.12	1.69	3.18	3.68	3.93	4.00	4.10
SpinQueue	510.9	30.5	42.6	45.7	46.8	47.2	120.6

Fig. 10. Advantages of polytasks on applications

experiments, we observe that polytasks provide significant advantages in efficiency for all three of the queue algorithms used (SpinQueue, MC-Queue and MS-Queue) in fine-grained environments.

5.2 Applications

The advantages of polytasks in production systems was tested using scientific applications running in typical environments.

Several applications were tested: Fast Fourier Transforms that use the Cooley-Tukey algorithm with two-point butterflies (FFT) and simulations of electromagnetic waves propagating using the Finite Difference Time Domain algorithm in 1 Dimension (FDTD1D)[15] and 2 dimensions (FDTD2D)[14]. FFT was run with input sizes 2^9 (FFT2P 2^9) through 2^{12} (FFT2P 2^{12}), FDTD1D runs a problem of size 20000 with 3 timesteps and tiles of width 16, and FDTD2D runs a problem of size 128 by 128 with 2 timesteps and tiles of width 4 by 4.

The results reported for all applications reflect the complete program and include memory allocation, initialization, computation and reporting of results.

In all cases, the programs were developed to use the TIDeFlow [13] runtime system. TIDeFlow uses a priority queue to assign work to processors in the system.

We compared the effect of polytasks by running several sets of experiments: The first set of experiments consists of pairing each one of the programs with each one of the versions of the TIDeFlow runtime system that use each one of the possible underlying queue algorithms described (SpinQueue, MC-Queue and MS-Queue) without the advantage of polytasks. For the second set of experiments, we ran all combinations of programs and TIDeFlow implementations using polytasks.

Figure 10 reports the speedup of each program with reference to an execution using the unmodified runtime system (Single Task). The objective of showing a comparison of polytasks against single tasks for each one of several underlying queue algorithms is to show that the advantages of polytasks are not exclusive to a particular queue algorithm. Such advantages are primarily the result of a runtime that operates more efficiently.

The reasons for the excellent speedup in the programs tested are: (1) The applications use very fine grain synchronization and a significant portion of the time is spent in task management. As explained in Section 5.1 and in Figure 9, these results hold for fine-grained applications. The impact of polytasks on other coarse-grained applications not considered here may not be as pronounced as the impact in the fine-grained applications that we tested. (2) The applications exhibit a large degree of task similarity because their main computational kernel consists mainly of `parallel for` loops.

6 Related Work

Several execution paradigms based in tasks have been presented in the past.

Intel's Concurrent Collections (CnC)[10] is an execution paradigm where programs are expressed in terms of computational tasks (or steps in the CnC terminology), data dependencies and control dependencies. The current generation of CnC implementations represent tasks as individual items in the work pool. However, polytasks could be a promising addition to CnC because it is frequent that several computation steps are similar in everything except in their control tag value, making the addition of polytasks a natural extension to allow higher scalability and performance.

The University of Delaware's Codelet Model[8], part of the ongoing DARPA UHPC project, is an initiative to achieve unprecedented parallelism in programs by expressing computations as dataflow graphs composed of codelets (computational tasks) that can migrate across large systems. Polytasks can potentially impact the effectiveness of the codelet model because it may help migration of tasks to remote locations if several tasks are compressed to a single polytask.

Other execution paradigms that use tasks in a way or another, and queues to manage tasks include X10[4], EARTH[17], Cilk[2] and Habanero C[1]. Polytasks offer interesting opportunities for those execution paradigms, and may potentially be used to improve their performance and scalability.

7 Conclusions and Future Work

We have shown that polytasks are an effective way to exploit the similarity between tasks that is commonly found in scientific programs that use a queue-centric approach for execution. The polytask technique allows queue-centric runtimes to exploit the same parallel loops as the OpenMP *dynamic* construct. Our research focus on how to develop a systematic technique for task compression rather than addressing particular situations in particular systems. Future work will compare the differences in performance between OpenMP's *dynamic* construct and polytasks.

We have presented a line of thought that concludes that there is a high degree of similarity in tasks that are enabled at the same time in scientific programs, mostly resulting from parallel loops that are expressed as a set of embarrassingly parallel tasks that become enabled.

We have taken advantage of the similarity of tasks and their proximity in time to invent a way to express them in a compressed form, that we call a polytask. We have shown that the data structures and algorithms of runtime systems require only minor modifications to support polytasks.

We have provided evidence, both in our informal analysis and in our experiments of the usability of polytasks for runtime systems.

The polytasks improve the performance of the runtime system when the programs run exhibit high task similarity. In cases where task compression is not possible, polytasks do not introduce significant overhead, and can be used safely.

The effect of polytasks on the speed of the runtime system can only be noticed in applications with fine granularity. In applications where parallelism has been exposed at a coarse-grain level, the issue of task management overhead is not as relevant because most of the application time is spent executing tasks. Nevertheless, if significant parallelism is required in future generations of multiprocessors with a large number of processing units per chip, fine grain parallelism will become a necessity.

We have shown that polytasks are effective for C64, a system that is non-preemptive, that has no cache and that supports atomic operations in-memory. Polytasks are effective for C64 because processors have very little overhead when they start executing a task: There is no cache that needs to be filled, there is no thread-state that needs to be put in place and there is no virtual memory that needs to be made available.

Although future work may analyze the usability of polytasks for preemptive systems, with few cores, caches, and other architectural features, it is the impression of the authors that queue-based runtime systems will primarily benefit systems with massive hardware parallelism where the architecture is directly exposed to the users.

Future work will focus on extending the polytask concept beyond the implementation of runtime systems, including the development of language or compiler extensions to indicate or hint task similarity outside of the trivial case of parallel loops.

References

1. Barik, R., Budimlic, Z., Cave, V., Chatterjee, S., Guo, Y., Peixotto, D., Raman, R., Shirako, J., Tasirlar, S., Yan, Y., Zhao, Y., Sarkar, V.: The habanero multicore software research project. In: Proceeding of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications, OOPSLA 2009, pp. 735–736. ACM, New York (2009)
2. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. In: Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 1995, pp. 207–216. ACM, New York (1995)

3. Butenhof, D.R.: Programming with POSIX threads. Addison-Wesley Longman Publishing Co., Inc., Boston (1997)
4. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. SIGPLAN Not. 40, 519–538 (2005)
5. del Cuvillo, J., Zhu, W., Hu, Z., Gao, G.R.: Fast: A functionally accurate simulation toolset for the cyclops-64 cellular architecture. CAPSL Technical Memo 062 (2005)
6. del Cuvillo, J., Zhu, W., Hu, Z., Gao, G.R.: Toward a software infrastructure for the cyclops-64 cellular architecture. In: High-Performance Computing in an Advanced Collaborative Environment, p. 9 (May 2006)
7. Dagum, L., Menon, R.: Openmp: an industry standard api for shared-memory programming. IEEE Computational Science Engineering 5(1), 46–55 (1998)
8. Gao, G., Suetterlein, J., Zuckerman, S.: Toward an execution model for extreme-scale systems-runnemede and beyond. CAPSL Technical Memo 104
9. Garcia, E., Venetis, I.E., Khan, R., Gao, G.R.: Optimized Dense Matrix Multiplication on a Many-Core Architecture. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010, Part II. LNCS, vol. 6272, pp. 316–327. Springer, Heidelberg (2010)
10. Knobe, K.: Ease of use with concurrent collections (cnc). In: Proceedings of the First USENIX Conference on Hot Topics in Parallelism, HotPar 2009, p. 17. USENIX Association, Berkeley (2009)
11. Mellor-Crummey, J.: Concurrent queues: Practical fetch and phi algorithms. Tech. Rep. 229, Dep. of CS, University of Rochester (1987)
12. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proc. of the 15th ACM Symposium on Principles of Distributed Computing, PODC 1996, pp. 267–275. ACM, New York (1996)
13. Orozco, D., Garcia, E., Pavel, R., Khan, R., Gao, G.R.: Tideflow: The time iterated dependency flow execution model. CAPSL Technical Memo 107 (2011)
14. Orozco, D., Garcia, E., Gao, G.: Locality Optimization of Stencil Applications Using Data Dependency Graphs. In: Cooper, K., Mellor-Crummey, J., Sarkar, V. (eds.) LCPC 2010. LNCS, vol. 6548, pp. 77–91. Springer, Heidelberg (2011)
15. Orozco, D.A., Gao, G.R.: Mapping the fdtd application to many-core chip architectures. In: Proceedings of the 2009 International Conference on Parallel Processing, ICPP 2009, pp. 309–316. IEEE Computer Society, Washington, DC (2009)
16. Shafiei, N.: Non-blocking Array-Based Algorithms for Stacks and Queues. In: Garg, V., Wattenhofer, R., Kothapalli, K. (eds.) ICDCN 2009. LNCS, vol. 5408, pp. 55–66. Springer, Heidelberg (2008)
17. Theobald, K.: EARTH: An Efficient Architecture for Running Threads. Ph.D. thesis (1999)

Detecting False Sharing in OpenMP Applications Using the DARWIN Framework

Besar Wicaksono, Munara Tolubaeva, and Barbara Chapman

University of Houston,
Computer Science Department, Houston, Texas, USA
http://www2.cs.uh.edu/~hpctools

Abstract. Writing a parallel shared memory application that achieves good performance and scales well as the number of threads increases can be challenging. One of the reasons is that as threads proliferate, the contention among shared resources increases and this may cause performance degradation. In particular, multi-threaded applications can suffer from the false sharing problem, which can degrade the performance of an application significantly. The work in this paper focuses on detecting performance bottlenecks caused by false sharing in OpenMP applications. We introduce a dynamic framework to help application developers detect instances of false sharing as well as identify the data objects in an OpenMP code that cause the problem. The framework that we have developed leverages features of the OpenMP collector API to interact with the OpenMP compiler's runtime library and utilizes the information from hardware counters. We demonstrate the usefulness of this framework on actual applications that exhibit poor scaling because of false sharing. To show the benefit of our technique, we manually modify the identified problem code by adjusting the alignment of the data that are causing false sharing; we then compare the performance with the original version.

Keywords: Cache coherence, false sharing, OpenMP, DARWIN, hardware counter, OpenMP collector API.

1 Introduction

With the widespread deployment of multi-core processors, many applications are being modified to enable them to fully utilize the hardware. The de-facto standard OpenMP [4], a shared memory programming model, is a popular choice for programming these systems. OpenMP offers a simple means to parallelize a computation so that programmers can focus on their algorithm rather than expend effort managing multiple threads. However, the simplicity of OpenMP also masks some potential problems from the programmer. One of the well-known problems is avoiding false sharing [21].

False sharing may occur on multi-core platforms as a result of the fact that blocks of data are fetched into cache on a per-line basis. When one thread accesses data that happens to be on the same line as data simultaneously accessed by another thread, both need up-to-date copies of the same cache line. In order to maintain the consistency of the shared data, the processor may then generate additional cache misses that degrade performance. It can be very hard for the application developer to correctly identify the source of such performance problems, since it requires some amount of understanding of the way in which the hardware supports data sharing in the presence of private caches.

We have created a dynamic optimization framework for OpenMP programs, called DARWIN, that is based on the open-source OpenUH [10] compiler, and have shown how it can be used to optimize applications that exhibit ccNUMA data locality problems [23]. The core feature of this framework is its usage of the OpenMP collector API [6] to interact with a running OpenMP program. The collector API can track various OpenMP states on a per-thread basis, such as whether a thread is in a parallel region. DARWIN also utilizes hardware counters to obtain detailed information about the program's execution. When combined with its other capabilities, such as relating the performance data to the data structures in the source code, DARWIN is able to help the application developer to gain insights about dynamic code behavior, particularly in the hot-spots of a parallel program.

In this paper, we explain how false sharing may arise in OpenMP applications and how the DARWIN framework may be used to identify data objects in an OpenMP code that cause false sharing. We describe DARWIN's ability to interact with the OpenMP runtime library and to access hardware counters, which is the starting point for our false sharing detection strategy. We then discuss the two stages involved in our approach. The first stage observes the degree of coherence problem in a program. The second stage isolates the data structure that leads to the false sharing problem.

The paper is structured as follows: section 2 describes the false sharing problem. Section 3 gives an introduction to the DARWIN framework. After presenting our methodology in section 4, we discuss our experiments in section 5. Section 6 discusses prior research closely related to our work. Finally, section 7 summarizes our findings and offers conclusions.

2 False Sharing

In a multi-threaded program running on a multicore processor, data sharing among threads can produce cache coherence misses. When a processor core modifies data that is currently shared by the other cores, the cache coherence mechanism has to invalidate all copies in the other cores. An attempt to read this data by another core shortly after the modification has to wait for the most recent value in order to guarantee data consistency among cores.

The data sharing that occurs when processor cores actually access the same data element is called true sharing. Such accesses typically need to be coordinated in order to ensure the correctness of the program. A variety of synchronization techniques may be employed to do so, depending on the programming interface being used. Some of these techniques are locks, monitors, and semaphores. In the

OpenMP API, *critical* and *atomic* are two constructs that prevent the code they are associated with, from being executed by multiple threads concurrently. The *critical* construct provides mutual exclusion for code blocks in a critical section, whereas the *atomic* construct ensures safe updating of shared variables. When the order of the accesses is important, the OpenMP *barrier* and *taskwait* constructs can be used to enforce the necessary execution sequence.

In contrast to true sharing, false sharing is an unnecessary condition that may arise as a consequence of the cache coherence mechanism working at cache line granularity [2]. It does not imply that there is any error in the code. This condition may occur when multiple processor cores access different data elements that reside in the same cache line. A write operation to a data element in the cache line will invalidate all the data in all copies of the cache line stored in other cores. A successive read by another core will incur a cache miss, and it will need to fetch the entire cache line from either the main memory or the updating core's private cache to make sure that it has the up-to-date version of the cache line. Poor scalability of multi-threaded programs can occur if the invalidation and subsequent read to the same cache line happen very frequently.

```
int *local_count = (int*)malloc(sizeof(int)*NUM_THREADS*PADDING);
int *vector = (int*)malloc(sizeof(int)*VECTOR_SIZE);
for(i=0;i<COUNT;i++)
{
#pragma omp parallel
    {
        int tid = omp_get_thread_num()*PADDING;
        if(tid < 0) tid = 0;

        #pragma omp for
        for(j = 0; j < VECTOR_SIZE; j++)
            local_count[tid] += vector[j]*2;

        #pragma omp master
        {
            int k;
            for(k = 0; k<NUM_THREADS; k++)
            result += local_count[k];
        }
    }
}
```

Fig. 1. OpenMP code snippet with false sharing problem

Figure 1 shows a code snippet from an OpenMP program that exhibits the false sharing problem. This code will read each value of a vector, multiply it by two, and calculate the sum. Its performance is inversely proportional to the number of threads as shown in Table 1.

Mitigating the false sharing effect can lead to an astonishing 57x performance improvement for this code. The reason for the poor performance of the unoptimized code lies in the way it accesses the *local_count* array. When the *PADDING* variable is set to 1, the size of the array is equal to the number of threads. The cache line size of the underlying machine being used is 128 bytes, so even though the threads access different elements of the array, they fetch the same cache line frequently and interfere with each other by causing unnecessary invalidations. By taking the cache line size into account and increasing the *PADDING* value, we can prevent threads from accessing the same cache lines continuously.

Table 1. Execution time of OpenMP code from Figure 1

Code version	Execution time(s)			
	1-thread	2-threads	4-threads	8-threads
Unoptimized	0.503	4.563	3.961	4.432
Optimized	0.503	0.263	0.137	0.078

3 DARWIN Framework

The DARWIN framework is a feedback dynamic optimization system that primarily leverages the features of the OpenMP collector API [6] to communicate with a running OpenMP program. Figure 2 illustrates the major components of the framework.

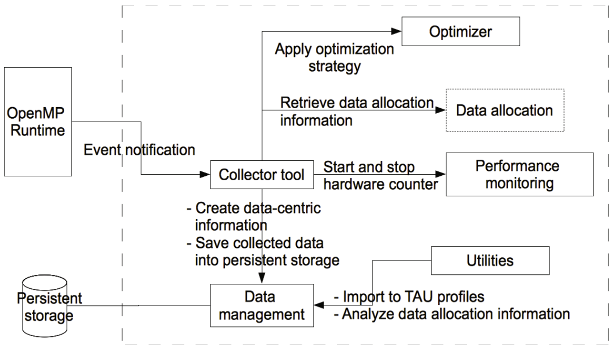

Optimizer
Apply optimization strategy
Retrieve data allocation information
Data allocation
OpenMP Runtime
Event notification
Collector tool
Start and stop hardware counter
Performance monitoring
- Create data-centric information
- Save collected data into persistent storage
Utilities
Persistent storage
Data management
- Import to TAU profiles
- Analyze data allocation information

Fig. 2. The DARWIN Framework

The collector tool is the central component of DARWIN. It utilizes the OpenMP collector API to facilitate transparent communication with the OpenMP application. The OpenUH OpenMP runtime library throws a collector event notification that represents transition to a particular state in the OpenMP program's execution. For example, the *OMP_EVENT_FORK* event is thrown when the OpenMP master thread enters a parallel region. The collector tool catches the event notification and provides it to the event handler that performs the appropriate profiling activity, such as starting and stopping the performance measurement for each thread in a parallel region.

DARWIN has two execution phases, monitoring and optimization. This work focuses on the monitoring phase, which collects performance data to enable the false sharing analysis. In the monitoring phase, the collector tool starts and stops the performance monitoring component when it receives a collector event notification about the start and end of an OpenMP parallel region, respectively.

Currently, the performance monitoring component utilizes Data Event Address Register (DEAR) of the Itanium 2 processor to capture samples of memory load operations, which contain meaningful information such as the referenced memory addresses and the load latency. Other processor models such as AMD Opteron and Intel Core support similar functionality via Instruction Base Sampling (IBS) and Precise Event Base Sampling (PEBS), respectively. Support for these hardware counters will be included in the future work.

At the end of the monitoring phase, the collector tool invokes the data manager to create data-centric information by relating the referenced memory addresses in the sampling results with the data structures at the source code level, thus making the information comprehensible to the programmer. Finally, the data manager saves all of the gathered information to persistent storage.

To support the creation of data-centric information, the data allocation component captures the allocation information of global, static, and dynamic data, which includes the parallel region id, thread id, starting address, allocation size, and the variable name. The allocation information for global and static data is provided by the symbol table of the executable file. For dynamic data, we interpose the memory allocation routines to capture the resulting starting address and allocation size parameter. We also retrieve the program counter of the allocation caller, which is used to find the caller's name and line number from the debug information. Both name and line number are used as the variable name for dynamic data. It is possible that more than one dynamic data allocation are attributed with the same variable name, which may indicate that these data are elements of an array, list, or tree. The allocation information is stored into a B-Tree [1] that offers very good performance for searching, especially when the number of allocation records, and hence the search space, is very large.

DARWIN provides several utilities to support data analysis by the programmer. One tool is used to export the collected performance data into text files that follow the Tuning and Analysis Utilities (TAU) [18] profile format. A second tool can be used to analyze the data allocation information.

TAU is a portable toolkit for performance analysis of parallel programs that are written in Fortran, C, C++, and Python. The programmer can run TAU's *Paraprof* [20] to observe the behavior of the program through its 3D visualization capabilities, and draw conclusions about the observed performance bottlenecks.

4 False Sharing Detection Methodology

Our approach for determining the data that exhibits false sharing consists of two stages. The first stage checks whether the cache coherence miss contributes to a major bottleneck in the program. The second stage isolates the data structures that cause the false sharing problem.

4.1 Stage 1 : Detecting Cache Coherence Problem

False sharing is a cache coherence problem related to the way processors maintain memory consistency. Therefore, observing the hardware behavior is a good way to determine if a program is suffering from coherence misses. Modern processors accommodate a performance monitoring unit (PMU) that can provide hints about the potential existence of cache coherence problems.

For example, the Intel Itanium 2 processor supports the PMU event *BUS_MEM_READ_BRIL_SELF* that gives the number of cache line invalidation transactions [22]. The Intel Core i7 family supports an event called *MEM_UNCORE_RETIRED.OTHER_CORE_L2_HITM* that indicates the number of retired memory load instructions that hit dirty data in sibling cores [7]. If a large number of each event is detected during a program's execution, it indicates that a serious cache coherence problem can occur when the program is executed with multiple threads.

4.2 Stage 2 : Isolating the Data Structures

To identify those data structures that have a major false sharing problem, our method starts by observing the accesses to cache lines with high level symptoms, which are a high memory access latency, and a large number of references. When a cache coherence miss occurs, a read operation from the processor needs to fetch the data from another processor's cache or wait until the memory is updated with the latest version, after which the processor reads directly from memory. This means that a cache coherence miss has longer latency than other kinds of cache misses. It has been reported [9] that a cache coherence miss on the Itanium 2 processor can exceed 180-200 cycles, while the average latency for reading from memory is 120-150 cycles. We are also aware that a modest amount of false sharing misses will not lead to a significant problem for application performance. Therefore, cache lines with a low number of references are ignored.

After the data structures showing the symptoms above have been identified, we examine the data allocation information to look for other data structures that share the problematic cache line. The search is also performed within the elements of the data structure if it is a linear or non-linear data structure. As discussed in section 3, these kinds of data structures are represented by multiple allocations with the same variable name. If the search attempt returns one or more results, we conclude that the problem is due to false sharing. Otherwise, we observe the data access pattern of the data structures that experience the false sharing symptoms.

False sharing can exist on a data structure that is accessed by multiple threads, where each thread only accesses a portion of the data. It is possible that some elements near the boundary of two disjoint data portions are in the same cache line. If the data structure size is small, e.g. it takes up about as many bytes as there are in a cache line, most elements of the data structure might be contained in the same cache line causing an increased risk of false sharing.

The results of our false sharing detection are validated by performing manual optimization to the source code that allocates the falsely shared data, without making extensive program restructuring. The optimization is performed by modifying the data layout to prevent falsely shared data from residing on the same cache line. We use GCC's *aligned* variable attribute and *posix_memalign* function to allocate data on the cache line boundary. The result of the detection is considered to be valid when the performance of the optimized code is substantially better.

5 Experiments

To evaluate our methodology, we performed experiments using the Phoenix suite [17] that implements MapReduce for shared memory systems. Phoenix provides eight sample applications parallelized with the Pthreads library that we have ported to OpenMP. The programs were compiled using the OpenUH compiler with optimization level O2. Each sample program included several input configurations. We chose the one that results in a reasonable execution time (not too long nor too short). The platform that we used was an SGI Altix 3700 consisting of 32 nodes with dual 1.3 GHz Intel Itanium 2 processors per node running the SUSE 10 operating system. The experiment was conducted using an interactive PBS[1] with four compute nodes and eight physical CPUs. Each program was executed with the *dplace* command to enforce thread affinity. The overall wall clock execution time was measured using the shell's *time* function.

5.1 Results of Detecting Cache Coherence Problem

Figure 3 presents the speedup for each program executed with different numbers of threads. The speedup is defined as the execution time of the serial run divided by the execution time of the multi-threaded version. The experiment for each configuration was performed three times and the average execution time was determined.

From all of the programs, only *kmeans* experienced an ideal speedup. The *matrix_multiply* and *pca* programs had reasonable results but did not come close to the ideal speedup. The speedup of *histogram*, *reverse_index*, and *word_count* programs was even lower than that obtained by these benchmarks. The *linear_regression*, and *string_match* programs suffered from a heavy slowdown when using more than one thread.

To determine whether the slowdown or poor scaling came from the cache coherency problem, we observe the cache invalidation event measurement. Table 2 shows the measurement results. It shows that *histogram*, *linear_regression*, *reverse_index*, *string_match*, and *word_count* had a very large number of cache invalidation events when using higher numbers of threads. This is an indication that these programs suffer from a cache coherence problem. Although both

[1] Portable Batch Session.

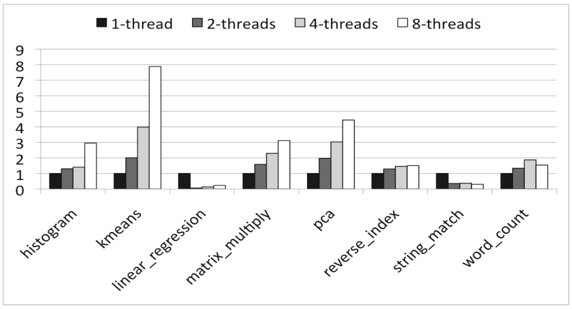


Fig. 3. The speedup of the original program

Table 2. Cache invalidation event measurement result

Program Name	**Total Cache Invalidation Count**			
	1-thread	2-threads	4-threads	8-threads
histogram	13	**7,820,000**	**16,532,800**	**5,959,190**
kmeans	383	28,590	47,541	54,345
linear_regression	9	**417,225,000**	**254,442,000**	**154,970,000**
matrix_multiply	31,139	31,152	84,227	101,094
pca	44,517	46,757	80,373	122,288
reverse_index	4,284	89,466	217,884	**590,013**
string_match	82	**82,503,000**	**73,178,800**	**221,882,000**
word_count	4,877	**6,531,793**	**18,071,086**	**68,801,742**

programs experienced sub-linear speedup in the number of threads as shown in Figure 3, *matrix_multiply* and *pca* had a low number of cache invalidation events.

An examination of the code reveals that the *matrix_multiply* program writes its results into an output file at the end of the program and that the *pca* program has many synchronizations using critical regions, which limits the speedup of both of them. The number of cache invalidation events in *reverse_index* program was pretty high when executed with eight threads, but the increase was not as extreme as with *histogram*, *linear_regression*, *string_match*, and *word_count*. We conclude that the coherence problem may not contribute significantly to the overall performance of *reverse_index*.

We then focused on the programs with a high number of cache invalidation events and collected the information required for identifying the variables that exhibit a false sharing problem. To help us analyze the collected information, we used the TAU *Paraprof* utility that can visualize the collected memory references information of each thread in a parallel region.

Due to the page limitation, we only present the analysis result for *linear_regression*, and *string_match* as shown in Figure 4 and 6 in the following subsections. Both programs experienced an enormous amount of cache invalidations and were slower when executed with multiple threads.

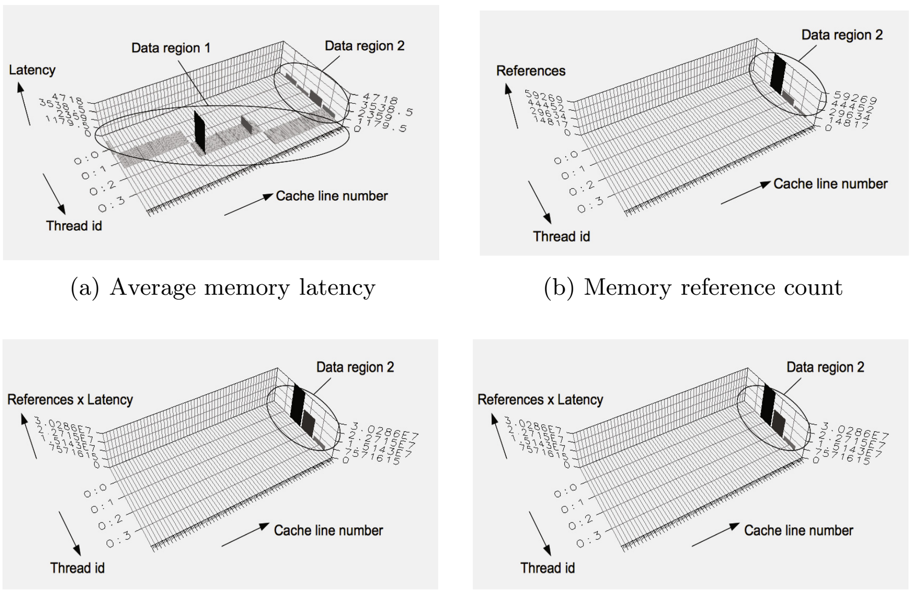
5.2 Linear_Regression

Figure 4(a) shows the average memory latency of the accesses to each cache line. The horizontal axis contains the cache line number. The vertical axis provides the memory access latency. The depth axis shows the thread id. There are two distinct data regions that can be identified in this figure. In data region 1, two cache lines referenced by thread 2 experienced a high latency, while the accesses from all threads to the cache lines in data region 2 had higher latency than most accesses to cache line in data region 1. Next we examine whether the two cache lines in data region 1 and the cache lines in data region 2 also had a high number of references.

Figure 4(b) shows the number of memory references to each cache line. The horizontal axis contains the cache line number. The vertical axis provides the number of references. The depth axis shows the thread id. The accesses to data region 1 were not shown by *Paraprof* because the number of references to this data region was very small compared to the number of references of region 2.

Figure 4(c) gives the total amount of memory latency for each cache line. The total latency is defined as the average memory latency multiplied by the total number of references. Since data region 2 dominated the total memory access latency, we suspected the data structures in this data region to be the leading cause of the false sharing problem.

According to DARWIN's data-centric information, data region 2 contained accesses to a variable named *main_155*. It was dynamic data allocated by the



(a) Average memory latency (b) Memory reference count

(c) Total memory latency (d) main_155 access pattern

Fig. 4. *Linear_regression* memory access visualization

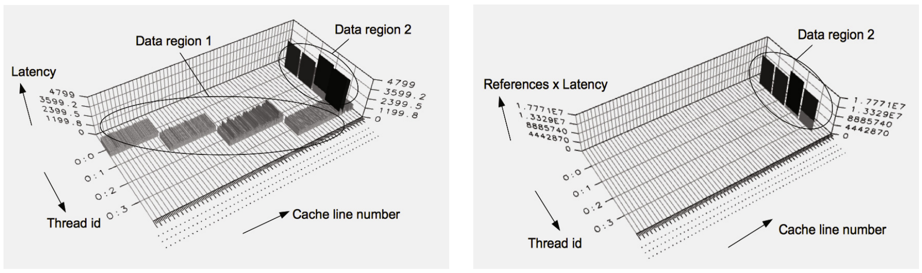
master thread in the main function, at line number 155. Its access pattern, presented in Figure 4(d), shows that updates to this data were distributed among the threads, and that in some cases, multiple threads were accessing the same cache line. In this case, the accesses from thread 1 to 3 hit elements of *main_155* that reside in the same cache line and have much higher latency than the accesses from thread 1. Therefore, we concluded that accesses to *main_155* caused the main false sharing problem in *linear_regression*. We also found a similar situation with *histogram*, where dynamic data shared among threads was causing the most significant bottleneck. We validated the findings by adjusting the data allocation alignment using the *aligned* attribute, as shown in Figure 5. The validation results are presented in Section 5.4.

```
typedef struct
{
    POINT_T *points   __attribute__((aligned (256)));
    int num_elems;
    long long SX, SY, SXX, SYY, SXY;
} lreg_args;
...
tid_args = (lreg_args *)calloc(sizeof(lreg_args),num_procs);
```

Fig. 5. Adjusting the data alignment in *linear_regression*

5.3 *String_match*

Figures 6(a) and 6(b) show the average and total memory access latency of each cache line, respectively. As with *linear_regression*, we identified two distinct



(a) Average memory latency (b) Total memory latency

Fig. 6. *String_match* memory access visualization

data regions with different access patterns. Both figures clearly show that the accesses to data structures in data region 2 were causing the major bottleneck of this program. According to the data-centric information, data region 2 contained the memory accesses to variable *key2_final* and *string_match_map_266*. The first variable was a global variable allocated in the same cache line with other global variables. We encountered the same situation in *reverse_index*, and *word_count*, where some of their global variables resided in the same cache line and were accessed frequently and had high latency.

string_match_map_266 was a dynamic data object allocated in the *string_match_map* function at line number 266. In contrast to the *main_155* variable, whose accesses were distributed among threads in *linear_regression*, *string_match_map_266* was allocated only by thread 3 and privately used within this thread. However, the problematic cache line of this variable was shared with a dynamic variable allocated by other thread.

Table 3 presents several data structures allocated in this program. It confirms that *key2_final* resided in the same cache line with *fdata_keys*. The first cache line of *string_match_map_266*, which is the problematic one, was shared with *string_match_map_268* that was allocated by thread 2.

Table 3. Data allocation information of several variables in *string_match*

Parallel region id	**Thread id**	**Variable name**	**Starting cache line**	**Last cache line**	**Size (bytes)**
0	0	fdata_keys	**0x00004a80**	**0x00004a80**	8
0	0	key_2_final	**0x00004a80**	**0x00004a80**	8
0	0	key_3_final	0x00004b00	0x00004b00	8
2	2	string_match_map_268	0x0c031f00	**0x0c032300**	1024
2	3	string_match_map_266	**0x0c032300**	0x0c032700	1024

The findings were validated by adjusting the data allocation alignment using the *aligned* attribute on *key2_final* and *fdata_keys*. We substitute the *malloc* routine for the allocation of *string_match_map_266* with the *posix_memalign* routine. These attempts are presented in figure 7.

```
char *key2_final __attribute__((aligned (256)));
char *fdata_keys __attribute__((aligned (256)));
...
posix_memalign(&cur_word,256,MAX_REC_LEN);
```

Fig. 7. Adjusting data alignment in *string_match*

5.4 Results of Memory Alignment

Figure 8 shows the speedup of each program over the original one after we performed the adjustments to the source code. The speedup is defined as the execution time of the original program divided by the execution time of the modified one. The memory alignment attempt produced visible improvement of the performance of *histogram*, *linear_regression*, *string_match*, and *word_count* with up to 30x speedup. The *reverse_index* program did not experience any significant improvement. However, this does not mean that our finding on *reverse_index* is invalid.

Table 4 presents the cache invalidation count of *reverse_index* after the memory alignment. It shows that the memory alignment successfully reduced the number of cache invalidations.

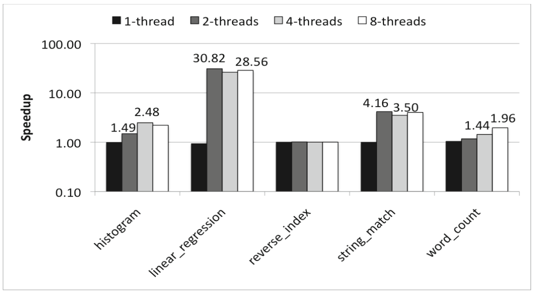


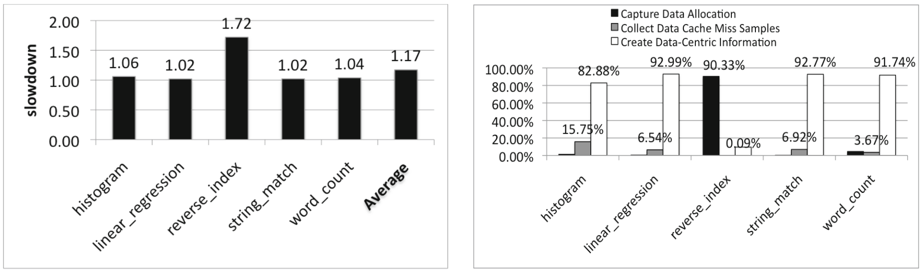
Fig. 8. The speedup after adjusting the memory alignment

Table 4. *Reverse_index* cache invalidation event measurement result

Code Version	Total Cache Invalidation Count			
	1-thread	2-threads	4-threads	8-threads
Unoptimized	4,284	89,466	217,884	590,013
Optimized	4,275	60,115	122,024	240,368

5.5 Performance Overhead

To determine overheads, we compare the execution time of the monitoring attempt with the original program execution time. Figure 9(a) shows the slowdown of the monitoring phase, defined as the monitoring execution time divided by the original program execution time. The monitoring overhead consisted of the time taken to capture data allocation, collect the data cache miss samples, and create data-centric information. Each of them was measured using the *gettimeofday* routine. Figure 9(b) gives the percentage of each overhead component in the total overhead time.



(a) (b)

Fig. 9. (a) The slowdown of the monitoring phase. (b) Overhead breakdown.

According to figure 9(a) the monitoring phase generated a slowdown ranging from 1.02x to 1.72x, with the average around 1.17x. *reverse_index* had the highest overhead with 1.72x slowdown, while the rest of the programs had less than 1.1x slowdown.

Figure 9(b) shows that the majority of *reverse_index*'s overhead was incurred during capture of the data allocation information. The reason for this is because *reverse_index* contained an excessive number of dynamic data allocations. *reverse_index* had 99,675 allocations, while the other programs had less than 50 allocations. Based on our investigation, traversing the stack frame to get the program counter during the dynamic data allocation can consume a significant amount of time, especially when the program has a large number of allocations. Reducing the need to traverse the stack frame is a subject for future work.

6 Related Work

False sharing is a performance problem that occurs as a consequence of the cache coherence mechanism working at cache line granularity. Detecting false sharing accurately requires complete information on memory allocation and memory (read and write) operations from each thread. Previous work [5,15,13,14] has developed approaches for memory analysis that use memory tracing and cache simulation. This starts by tracking the memory accesses (both loads and stores) at runtime. A cache simulator then takes the captured data to analyze the sequence of each memory operation and determine the type and amount of cache misses generated during the simulation. The main drawback of this approach is within the memory tracing part, which can incur very large overheads. A memory shadowing technique was used [24] in an attempt to minimize the overhead of tracking the changes to the data state. Our approach does not analyze the sequence of memory operations. We exploit higher level information, such as the latency, and the number of memory access references. By utilizing the hardware performance monitoring unit, the overhead of capturing this information can be minimized. Furthermore, our approach can quickly pinpoint the falsely shared data that has a significant impact on the performance.

Others [12,16,7,14] also use information from the hardware performance monitoring unit to support performance analysis. HPCToolkit[12], and Memphis[16] use the sampling result from AMD IBS to generate data centric information. HPCToolkit utilizes the information to help a programmer find data structures with poor cache utilization. Memphis focuses on finding data placement problem on ccNUMA platform. Intel PTU[7] utilizes event-based sampling to identify the data address and function that is likely to experience false sharing. Our work is more focused on finding the actual data objects that suffer greatly from false sharing. Our work is similar to [14] in terms of the utilization of the performance monitoring unit to capture memory load operations. While [14] directs the captured information to the cache simulator in order to analyze the memory access sequences, our approach uses it to analyze the latency, and the number of references.

Several attempts have been made to eliminate the false sharing problem. For example, careful selection of runtime scheduling parameters such as chunk size and chunk stride when distributing loop iterations to threads has been used to prevent false sharing [3]. Proposed data layout optimization solutions include array padding [8] and memory alignment methods[19]. A runtime system called Sheriff [11] performs both detection and elimination of false sharing in C/C++ applications parallelized using the Pthreads library. Sheriff eliminates false sharing by converting each thread into a process and creating a private copy of shared data for each process. It detects false sharing by observing each word in the private copy of each process. False sharing is detected when there is a modification to a word adjacent with another word of another process's private copy, and both words reside on the same cache line. Our work is primarily concerned with detecting the false sharing problem, rather than eliminating it.

7 Conclusion

This paper describes our implementation of a technique to detect false sharing in OpenMP applications in the DARWIN framework. DARWIN provides features to enable communication with the OpenMP runtime library, and to capture the data access pattern.

The technique consists of two stages, which are 1) detection of coherence bottlenecks in the program with the help of hardware counters and 2) identification of data objects that cause the false sharing problem. The second stage utilizes the data allocation and access pattern information to distinguish the data structures that cause major bottlenecks due to false sharing.

To observe the effectiveness of our method, the technique was applied to several Phoenix programs that exhibit false sharing. Bottlenecks caused by the cache coherence problem were detected, and the data causing serious false sharing problems were identified. Optimizing the memory alignment of the problematic data greatly improved the performance, thus indicating that our method is capable of identifying the data responsible for the false sharing problem. DARWIN's monitoring approach created an average of 1.17x slowdown of the programs used. Traversing the stack frame when capturing the dynamic data allocation can incur large overhead, especially when the program has a large number of dynamic data allocations.

Acknowledgement. The authors thank their colleagues in the HPCTools group for providing useful feedback during the writing of this paper and the Texas Learning and Computation Center (TLC2) for use of their hardware[2]. This work is supported by the National Science Foundation under grant CCF-0702775 (see also `http://www.cs.uh.edu/~hpctools/darwin`).

[2] `http://tlc2.uh.edu`

References

1. Bayer, R., McCreight, E.: Organization and Maintenance of Large Ordered Indices. Mathematical and Information Sciences Report No. 20 (1970)
2. Chapman, B., Jost, G., Pas, R.V.D.: Using OpenMP: Portable Shared Memory Parallel Programming. The MIT Press (2008)
3. Chow, J.-H., Sarkar, V.: False Sharing Elimination by Selection of Runtime Scheduling Parameters. In: Proceedings of the ICPP (1997)
4. Dagum, L., Menon, R.: OpenMP: An Industry-Standard API for Shared-Memory Programming. IEEE Computational Science and Engineering (1998)
5. Günther, S.M., Weidendorfer, J.: Assessing Cache False Sharing Effects by Dynamic Binary Instrumentation. In: Proceedings of the WBIA (2009)
6. Hernandez, O., Chapman, B., et al.: Open Source Software Support for the OpenMP Runtime API for Profiling. In: P2S2 (2009)
7. Intel. Avoiding and Identifying False Sharing Among Threads (2010)
8. Jeremiassen, T.E., Eggers, S.J.: Reducing False Sharing on Shared Memory Multiprocessors Through Compile Time Data Transformations. SIGPLAN (1995)
9. Kim, J., Hsu, W.-C., Yew, P.-C.: COBRA: An Adaptive Runtime Binary Optimization Framework for Multithreaded Applications. In: ICPP (2007)
10. Liao, C., Hernandez, O., Chapman, B., Chen, W., Zheng, W.: OpenUH: An Optimizing, Portable OpenMP Compiler. In: CPC (2006)
11. Liu, T., Berger, E.: Sheriff: Detecting and Eliminating False Sharing. Technical report, University of Massachusetts, Amherst (2010)
12. Liu, X., Mellor-Crummey, J.: Pinpointing Data Locality Problems Using Data-centric Analysis. In: CGO (2011)
13. Marathe, J., Mueller, F.: Source-Code-Correlated Cache Coherence Characterization of OpenMP Benchmarks. IEEE Trans. Parallel Distrib. Syst. (June 2007)
14. Marathe, J., Mueller, F., de Supinski, B.R.: Analysis of Cache-Coherence Bottlenecks with Hybrid Hardware/Software Techniques. ACM TACO (2006)
15. Martonosi, M., Gupta, A., Anderson, T.: MemSpy: Analyzing Memory System Bottlenecks in Programs. SIGMETRICS Perform. Eval. Rev. 20, 1–12 (1992)
16. McCurdy, C., Vetter, J.: Memphis: Finding and Fixing Numa-Related Performance Problems on Multi-Core Platforms. In: ISPASS (2010)
17. Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., Kozyrakis, C.: Evaluating MapReduce for Multi-core and Multiprocessor Systems. In: HPCA (2007)
18. Shende, S.S., Malony, A.D.: The TAU Parallel Performance System. Int. J. High Perform. Comput. Appl. (2006)
19. Torrellas, J., Lam, H.S., Hennessy, J.L.: False Sharing and Spatial Locality in Multiprocessor Caches. IEEE Trans. Comput. 43, 651–663 (1994)
20. University of Oregon. ParaProf User's Manual
21. van der Pas, R.: Getting OpenMP Up To Speed (2010)
22. Vogelsang, R.: SGA Altix Tuning OpenMP Parallelized Applications (2005)
23. Wicaksono, B., Nanjegowda, R.C., Chapman, B.: A Dynamic Optimization Framework for OpenMP. In: Chapman, B.M., Gropp, W.D., Kumaran, K., Müller, M.S. (eds.) IWOMP 2011. LNCS, vol. 6665, pp. 54–68. Springer, Heidelberg (2011)
24. Zhao, Q., Koh, D., Raza, S., Bruening, D., Wong, W.-F., Amarasinghe, S.: Dynamic Cache Contention Detection in Multi-threaded Applications. In: VEE (2011)

Author Index

Amini, Mehdi 237

Bae, Hansang 1, 141
Bagwell, Phil 156
Brinkers, Daniel 46

Chapman, Barbara 283
Chatterjee, Sanjay 203
Chung, JaeWoong 124
Coelho, Fabien 237
Cohen, Albert 185

Dümmler, Jörg 252

Edahiro, Masato 31
Eigenmann, Rudolf 1, 141

Fifield, Jeffrey 16

Gan, Ge 106
Gao, Guang R. 106, 268
Garcia, Elkin 268
Gottschlich, Justin E. 124
Grossman, Max 203
Grunwald, Dirk 16
Guo, Ziyu 171

Halle, Sean 185
Hayashi, Akihiro 31

Irigoin, François 237

Jo, Gangwon 76
Jubair, Fahed 1

Kasahara, Hironori 31
Keryell, Ronan 237
Khan, Rishi 268
Kim, Jungwon 76
Kim, Soohong P. 141
Kimura, Keiji 31
Kitaki, Shumpei 31
Kwon, Okwan 1

Lee, Jaejin 76
Lee, Jun 76
Li, Xiaoming 218
Lin, Hao 141

Manzano, Joseph B. 106
Mase, Masayoshi 31
Midkiff, Samuel P. 1, 141
Mikami, Hiroki 31
Min, Seung-Jai 1

Nah, Jeongho 76

Oancea, Cosmin E. 61
Odersky, Martin 156
Orozco, Daniel 268

Pavel, Robert 268
Philippsen, Michael 46, 91
Prokopec, Aleksandar 156

Rauber, Thomas 252
Rauchwerger, Lawrence 61
Ributzka, Juergen 106
Rünger, Gudula 252

Sarkar, Vivek 203
Sbîrlea, Alina 203
Schreier, Silvia 91
Seo, Sangmin 76
Shen, Xipeng 171
Shimaoka, Mamoru 31
Shrestha, Sunil 106

Taylor, Ryan 218
Tillmann, Nikolai 46
Tolubaeva, Munara 283

Werth, Tobias 91
Wicaksono, Besar 283