The 2nd Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures

PESPMA 2009

June 21, 2009
Austin, TX, USA
(In conjunction with ISCA 2009)
Message from the Workshop Co-chairs

It is our great pleasure to welcome you to the 2nd Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures (PESPMA 2009) which is held in Austin, TX on June 21, 2009.

This workshop, which was debuted last year in Beijing, focuses on improving the performance and power efficiency of sequential applications on multi-core architectures. It provides a premier forum for researchers and engineers from academia and industry to discuss their latest research on sequential program execution on multi-core from the areas of computer architecture, compilers and programming languages. The workshop is intended to drive the efficiency of legacy sequential programs, sequential sections in parallel programs, and new sequential applications in the multi-core era.

This year’s high quality technical program includes two keynote presentations and 10 technical papers. We are honored to have David August, Princeton University, and Erik Altman, IBM Research, as our keynote speakers. We received 19 submissions and the program committee selected 10 papers for presentation at the workshop. Each submission was reviewed carefully by at least 4 program committee members and given high quality feedback.

Finally, we would like to thank all PC members for completing their reviews under really tight time constraints. Thanks to Jeff Hao of the University of Michigan for setting up and maintaining the submission and review web site.

We hope you all will like the workshop and find it a valuable event.

Wei Liu, Intel Labs
Scott Mahlke, University of Michigan
Tin-fook Ngai, Intel Labs
PESPMA 2009 Co-chairs
Workshop Organization

Co-Chairs

Wei Liu, Intel Labs, USA
Scott Mahlke, University of Michigan, USA
Tin-fook Ngai, Intel Labs, USA

Program Committee

Doug Burger, Microsoft Research, USA
Calin Cascaval, IBM Research, USA
Luis Ceze, University of Washington, USA
Nate Clark, Georgia Institute of Technology, USA
Josep M. Codina, Intel Barcelona Research Center, Spain
Alexandra Fedorova, Simon Fraser University, Canada
Rajiv Gupta, University of California, Riverside, USA
Hsien-Hsin S. Lee, Georgia Institute of Technology, USA
Onur Mutlu, Carnegie Mellon University, USA
Rodric Rabbah, IBM Research, USA
Ronny Ronen, Intel Corporation, Israel
James Tuck, North Carolina State University, USA
Table of Contents

Keynote I:

The Audacity of Hope: Thoughts on Restoring Computing’s Former Glory
David August, Princeton University .................................................................1

Session I: Thread Level Parallelism

Toward Automatic Data Structure Replacement for Effective Parallelization
Changhee Jung and Nathan Clark (Georgia Institute of Technology)..........................2

Factoring Out Ordered Sections to Expose Thread-Level Parallelism
Hans Vandierendonck, Sean Rul and Koen De Bosschere (Ghent University)...........12

Dynamic Concurrency Discovery for Very Large Windows of Execution
Jacob Nelson and Luis Ceze (University of Washington).....................................20

Parallelization Spectroscopy: Analysis of Thread-level Parallelism in HPC Programs
Arun Kejariwal (University of California, Irvine) and Calin Cascaval (IBM Corporation)......30

Keynote II:

Exploiting Hidden Parallelism
Erik Altman, IBM Research.............................................................................40

Session II: Binary and Run-time Execution

DBT86: A Dynamic Binary Translation Research Framework for the CMP Era
Ben Hertzberg and Kunle Olukotun (Stanford University)....................................41

A Concurrent Trace-based Just-In-Time Compiler for Single-threaded JavaScript
Jungwoo Ha (University of Texas at Austin), Mohammad R. Haghighat, Shengnan Cong
(Intel Corporation) and Kathryn S. McKinley (University of Texas at Austin)...........47

Session III: Multicore Architecture and Speculation

Exploring Practical Benefits of Asymmetric Multicore Processors
Jon Hourd, Chaofei Fan, Jiasi Zeng, Qiang (Scott) Zhang, Micah J. Best, Alexandra
Fedorova and Craig Mustard (Simon Fraser University)........................................55

Hybrid Operand Communication for Dataflow Processors
Dong Li, Behnam Robatmili, Sibi Govindan, Doug Burger and Steve Keckler (University of
Texas at Austin)..............................................................................................51

Hierarchical Control Flow Speculation: Support for Aggressive Predication
Hadi Esmaeilzadeh (University of Texas at Austin) and Doug Burger (Microsoft Research)....61

Tolerating Delinquent Loads with Speculative Execution
Chuck (Chengyan) Zhao, J. Gregory Steffan, Cristiana Amza (University of Toronto) and
Allan Kielstra (IBM Corporation)......................................................................81
The Audacity of Hope: Thoughts on Restoring Computing's Former Glory *

David August
Princeton University

Abstract:

Multicore is the manifestation of a failure on the part of computer architects to continue the decades old, universal performance trend despite the continuation of Moore's law. Even in delivering all that is promised, commercial and academic research efforts will only reduce the enormous negative impact multicore will have on companies, individuals, and society. Failure is certain when we act on the belief that success is impossible. The purpose of this talk is to build faith, by evidence and by demonstration, in a solution without compromises, a solution which sustains generations of scalable performance even on notoriously sequential legacy codes, a solution which preserves our most precious natural resource (sanity) and restores computing's former glory.

* with apologies to our current President.

Bio:

David I. August is an Associate Professor in the Department of Computer Science at Princeton University. With the Liberty Research Group (http://liberty.princeton.edu), David fights for freedom from the tyranny of those who would have us employ parallel programming to address the software impact of multicore computing.
Toward Automatic Data Structure Replacement for Effective Parallelization

Changhee Jung and Nathan Clark
College of Computing
Georgia Institute of Technology
{cjung9, ntclark}@cc.gatech.edu

ABSTRACT
Data structures define how values being computed are stored and accessed within programs. By recognizing what data structures are being used in an application, tools can make applications more robust by enforcing data structure consistency properties, and developers can better understand and more easily modify applications to suit the target architecture for a particular application.

This paper presents the design and application of DDT, a new program analysis tool that automatically identifies data structures within an application. A binary application is instrumented to dynamically monitor how the data is stored and organized for a set of sample inputs. The instrumentation detects which functions interact with the stored data, and creates a signature for these functions using dynamic invariant detection. The invariants of these functions are then matched against a library of known data structures, providing a probable identification. That is, DDT uses program consistency properties to identify what data structures an application employs. The empirical evaluation shows that this technique is highly accurate across several different implementations of standard data structures.

1. INTRODUCTION
Data orchestration is rapidly becoming the most critical aspect of developing effective manycore applications. Several different trends drive this movement. First, as technology advances getting data onto the chip will become increasingly challenging. The ITRS road map predicts that the number of pads will remain approximately constant over the next several generations of processor integration [11]. The implication is that while computational capabilities on-chip will increase, the bandwidth will remain relatively stable. This trend puts significant pressure on data delivery mechanisms to prevent the vast computational resources from starving.

Application trends also point toward the importance of data orchestration. A recent IDC report estimates that the amount of data in the world is increasing tenfold every five years [8]. That is, data growth is outpacing the current growth rate of transistor density. There are many compelling applications that make use of big data, and if systems cannot keep pace with the data growth then they will miss out on significant opportunities in the application space.

Lastly, a critical limitation of future applications will be their ability to effectively leverage massively parallel compute resources. Creating effective parallel applications requires generating many independent tasks with relatively little communication and synchronization. To a large extent, these properties are defined by how data used in the computation is organized. As an example, previous work by Lee et al. found that effectively parallelizing a program analysis tool required changing the critical data structure in the program from a splay-tree to a simpler binary search tree [14]. While a splay-tree is generally faster on a single core, splay accesses create many serializations when accessed from multicore processors. Proper choice of data structure can significantly impact the parallelism in an application.

All of these trends point to the fact that proper use of data structures is becoming more and more important for effective manycore software development.

Unfortunately, selecting the best data structure when developing applications is a very difficult problem. Often times, programmers are domain specialists with no knowledge of performance engineering, and they simply do not understand the properties of data structures they are using. One can hardly blame them; when last accessed, the Wikipedia list of data structures contained 74 different types of trees! How is a developer, even a well trained one, supposed to choose which tree is most appropriate for their current situation?

And even if the programmer has perfect knowledge of data structure properties, it is still extraordinarily difficult to choose the best data structures. Architectural complexity significantly complicates traditional asymptotic analysis, e.g., how does a developer know which data structures will best fit their cache lines or which structures will have the least false-sharing? Beyond architecture, the proper choice of data structure can even depend on program inputs. For example, splay-trees are designed so that recently accessed items are quickly accessed, but elements without temporal locality will take longer. In some applications it is impossible to know a priori input data properties such as temporal locality. Data structure selection is also a problem in legacy code. For example, if a developer created a custom map that fit well into processor cache lines in 2002, that map would likely have suboptimal performance using the caches in modern processors.

Choosing data structures is very difficult, and poor data structure selection can have a major impact on application performance. For example, Liu and Rus recently reported a 17% performance improvement on one Google internal application just by changing a single data structure [15]. We need better tools that can identify when poor data structures are being used, and can provide suggestions to developers on better alternatives.
In an ideal situation, an automated tool would recognize what data structures are utilized in an application, use sample executions of the program to determine whether alternative data structures would be better suited, and then automatically replace poor data structure choices.

In this work we attempt to solve the first step of this vision: data structure identification. The Data-structure Detection Tool, or DDT, takes an application binary and a set of representative inputs and produces a listing of the probable data structure types corresponding to program variables. DDT works by instrumenting memory allocations, stores, and function calls in the target program. Data structures are predominantly stored in memory, and so instrumentation tracks how the memory layout of a program evolves. Memory layout is modeled as a graph: allocations create nodes, and stores to memory create edges between graph nodes. DDT makes the assumption that access to memory comprising a data structure is encapsulated by interface functions, that is, a small set of functions that can insert or access data stored in the graph, or otherwise modify nodes and edges in the graph.

Once the interface functions are identified, DDT uses the Daikon invariance detection tool [7] to determine the properties of the functions with respect to the graph. For example, an insertion into a linked list will always increase the number of nodes in the memory graph by one and the new node will always be connected to other nodes in the list. A data value being inserted into a splay-tree will always be located at the root of the splay-tree. We claim that together, the memory graph, the set of interface functions, and their invariants uniquely define a data structure. Once identified in the target application, the graph, interface, and invariants are compared against a predefined library of known data structures for a match, and the result is output to the user. The information about data structure usage can be fed into performance modeling tools, which can estimate when alternative data structures may be better suited for an application/architecture, or simply used by developers to better understand the performance characteristics of the applications they are working on.

We have implemented DDT as part of the LLVM toolset [13] and tested it on several real-world data structure libraries: the GNOME C Library (GLib) [23], the Apache C++ Standard Library (STDCXX) [22], Borland C++ Builder’s Standard Library implementation (STLport) [21], and a set of data structures used in the Triman research compiler [25]. This work demonstrates that DDT is quite accurate in detecting data structures no matter what the implementation.

2. RELATED WORK

There is a long history of work on detecting how data is organized in programs. Shape analysis (e.g., [9, 19, 26]) is among the most well known of these efforts. The goal of shape analysis is to statically prove externally provided properties of data structures, e.g., that a list is always sorted or that a graph is acyclic. Despite significant recent advances in the area [12, 27], shape analysis is provably undecidable and thus necessarily conservative.

Related to shape analysis are dynamic techniques that observe running applications in an attempt to identify properties of data structures [6]. These properties can then be used to automatically detect bugs, repair data structures online, or improve many other software engineering tasks [5]. While this type of analysis is not sound, it can detect properties outside the scope of static analysis and has proven very useful in practice.

This previous work statically proved or dynamically enforced data structure consistency properties in order to find bugs or add resilience to applications. The work here takes a different approach, where we assume the data structure is consistent (or mostly consistent), and use the consistency properties to identify how the data structure operates. We are leveraging consistency properties to synthesize high-level semantics about data structures in the program.

Similar to this goal, work by Guo et al. [16] tries to automatically derive higher-level information from a program. In Guo’s work, a program analyzer uses dataflow analysis to monitor how variables interact. The interactions are used to synthesize subtypes in situations where a single primitive type, such as integer, could have multiple meanings that are not meant to interact, such as distance and temperature. Our work is different in that we are not inferring types, so much as we are recognize the functionality provided by complex groups of variables. Identifying data structures involves not only separating values into partitioned groups, but also identifying interface functions to the structure and recognizing what the interface functions do. Both Guo’s technique and our technique have important uses and can be leveraged to improve programmer understanding of the application.

The reverse-engineering community has also done work similar to this effort [1, 17]. These prior works use a variety of static, dynamic, and hybrid techniques to detect interaction between objects in order to reconstruct high-level design patterns in the software architecture. In this paper we are interested not just in the design patterns, but also in identifying the function of the structures identified.

The three works most similar to ours are by Raman et al. [18], Dekker et al. [4], and Cozzie et al. [3]. Raman’s work introduced the notion of using a graph to represent how data structures are dynamically arranged in memory, and utilized that graph to perform optimizations beyond what is possible with conservative points-to or shape analysis. Raman’s work differs from this work in that it was not concerned with identifying interface functions or determining exactly what data structure corresponds to the graph. Additionally, we extend their definition of a memory graph to better facilitate data structure identification.

Dekker’s work on data structure identification is exactly in line with what we attempt to accomplish in this paper. The idea in Dekker’s work was to use the program parse tree to identify patterns that represent equivalent implementations of data structures. Our work is more general, though, because (1) the DDT analysis is dynamic and thus less constrained, (2) DDT does not require source code access, and (3) DDT does not rely on the ability to prove that two implementations are equivalent at the parse tree level. DDT uses program invariants of interface functions to identify equivalent implementations, instead of a parse tree. This is a fundamentally new approach to identifying what data structures are used in applications.

Cozzie’s work presented a different approach to recognizing data structures: using machine learning to analyze raw data in memory with the goal of matching groups of similar data. Essentially, Cozzie’s approach is to reconstruct the memory graph during ex-
ecution and match graphs that look similar, grouping them into types without necessarily delineating the functionality. Instead this paper proactively constructs the memory graph during allocations, combines that with information about interface functions, and matches the result against a predefined library. Given the same application as input, Cozzie’s work may output “two data structures of type A, and one of type B,” whereas DDT would output “two sets implemented as red-black trees and one doubly-linked list.”

The take away is that DDT collects more information to provide a more informative result, but requires a predefined library to match against and more time to analyze the application. Cozzie’s approach is clearly better suited for applications such as malware detection, where analysis speed is important and information on data structure similarity is enough to provide a probable match against known malware. Our approach is more useful for applications such as performance engineering where more details on the implementation are needed to intelligently decide when alternative data structures may be advantageous.

The following are the contributions of this paper:

- **A new approach to identifying data structures:** DDT dynamically monitors the memory layout of an application, and detects interface functions that access or modify the layout. Invariant properties of the memory graph and interface functions are matched against a library of known data structures, providing a probabilistic identification. This approach significantly improves on previous work, as it is less conservative, does not require source code access, and is not dependent on data structure implementation.

- **An empirical evaluation demonstrating DDT’s effectiveness:** We test the effectiveness of DDT on several real-world data structure libraries and show that, while unsound, this analysis is both reasonably fast and highly accurate. DDT can be used to help programmers understand performance maladies in real programs, which ultimately helps them work with the compiler and architecture to choose the most effective data structures for their systems.

3. DATA STRUCTURE IDENTIFICATION ALGORITHM DETAILS

The purpose of DDT is to provide a tool that can correctly identify what data structures are used in an application regardless of how the data structures are implemented. The thesis of this work is that data structure identification can be accomplished by the following: (1) Keeping track of how data is stored in and accessed from memory; this is achieved by building the memory graph. (2) Identifying what functions interact with the memory comprising a data structure; this is achieved with the help of the annotated call graph. (3) Understanding what those functions do; invariants on the memory organization and interface functions are the basis for characterizing how the data structure operates.

Figure 1 shows a high-level diagram of DDT. An application binary and sample inputs are fed into a code instrumentation tool, in this case a dynamic compiler. It is important to use sample executions to collect data, instead of static analysis, because static analysis is far too conservative to effectively identify data structures. It is also important for DDT to operate on binaries, because often times data structure implementations are hidden in binary-only format behind library interfaces. It is unrealistic to expect modern developers to have source code access to their entire applications, and if DDT required source code access then it would be considerably less useful.

Once instrumented, the sample executions record both memory allocations and stores to create an evolving memory graph. Loads are also instrumented to determine which functions access various parts of the memory graph, thus helping to delineate interface functions. Finally, function calls are also instrumented to describe the state of the memory graph before and after their calls. This state is used to detect invariants on the function calls. Once all of this information is generated by the instrumented binary, an offline analysis processes it to generate the three traits (memory graph, interface functions, and invariants) needed to uniquely identify a data structure. Identification is handled by a hand-designed decision tree within the library that tests for the presence of the critical characteristics that distinguish data structures. For example, if nodes in a memory graph always have one edge that points to NULL or another node from the same allocation site, and there is an insert-like function which accesses that graph, etc., then it is likely that this memory graph represents a singly-linked list. The remainder of this section describes in detail how DDT accomplishes these steps using C++-based examples.
3.1 Tracking Data Organization with a Memory Graph

Part of characterizing a data structure involves understanding how data elements are maintained within memory. This relationship can be tracked by monitoring memory regions that exist to accommodate data elements. By observing how the memory is organized and the relationships between allocated regions, it is possible to partially infer what type of data structure is used. This data can be tracked by a graph whose nodes and edges are sections of allocated memory and the pointers between allocated regions, respectively. We term this a memory graph.

The memory graphs for an application are constructed by instrumenting memory allocation functions (e.g., `malloc`) and stores. Allocation functions create a node in the memory graph. DDT keeps track of the size and initial address of each memory allocation in order to determine when memory accesses to each region occur. An edge between memory nodes is created whenever a store is encountered whose target address and data operands both correspond to addresses of nodes that have already been allocated. The target address of the store is maintained so that DDT can detect when the edge is overwritten, thus adjusting that edge during program execution.

Figure 2 (a) illustrates how a memory graph is built when two memory cells are created and connected to each other. Each of the allocations in the pseudo-code at the top of this figure create a memory node in the memory graph. The first two stores write constant data `NULL` to the offset corresponding to `next`. As a result, two edges from each memory node to the data are created. For the data being stored, two nodes are created. To distinguish data from memory nodes, they have no color in the memory graph. The target address of the store is maintained so that DDT can detect when the edge is overwritten, thus adjusting that edge during program execution.

Extending the Memory Graph: The memory graph as presented thus far is very similar to that presented in previous work [18]. However, we have found that using this representation is not sufficient to identify many important invariants for data structure identification. For example, if the target application contained a singly-linked list of dynamically allocated objects, then it would be impossible to tell what part of the graph corresponded to list and what part corresponded to dynamically allocated data.

Extending the Memory Graph: The memory graph as presented thus far is very similar to that presented in previous work [18]. However, we have found that using this representation is not sufficient to identify many important invariants for data structure identification. For example, if the target application contained a singly-linked list of dynamically allocated objects, then it would be impossible to tell what part of the graph corresponded to list and what part corresponded to dynamically allocated data.

The purpose of allocation-site-based typing of the memory nodes is to solve exactly the problem described above: differentiating memory nodes between unrelated data structures. Many people have previously noted that there is often a many-to-one mapping between memory allocation sites and a data structure type [10]. Thus, if we color nodes in the memory graph based on their allocation site, it is easy to determine what part of the memory graph corresponds to a particular data structure and what part corresponds to dynamically allocated data.

However, in the many-to-one mapping, an allocation site typically belongs to one data structure, but one data structure might have many allocation sites. In order to correctly identify the data structure in such a situation, it is necessary to merge the memory node types. This merging can be done by leveraging the observation that even if memory nodes of a data structure are created in different allocation sites, they are usually accessed by the same method in another portion of the application. For example, even if a linked-list allocates memory nodes in both `push_front` and `push_back`, the node types can be merged together when a `back` method is encountered that accesses memory nodes from both allocation sites.

While empirical analysis suggests this does help identify data structures in many programs, allocation-site-based coloring does not
help differentiate graph nodes in applications with custom memory allocators. That is because multiple data structures can be created in a single allocation site, which is the custom memory allocator. This deficiency could be remedied by describing the custom memory allocators to DDT so that they could be instrumented as standard allocators, such as malloc, currently are.

The second extension proposed for the memory graph is typing of edges. As with node coloring, typing the edges enables the detection of several invariants necessary to differentiate data structures. We propose three potential types for an edge in the memory graph: child, foreign, and data. Child edges point to/from nodes with the same color, i.e., nodes from the same data structure. The name “child” edge arose from when we first discovered their necessity when trying to identify various types of trees. Foreign edges point to/from memory graph nodes of different colors. These edges are useful for discovering composite data structures, e.g., `list<set<vector>>`. Lastly, data edges simply identify when a graph node contains static data. These edges are needed to identify data structures which have important properties stored in the memory graph nodes. E.g., a red-black tree typically has a field which indicates whether each node is red or black.

A single edge in the memory graph can have several different uses as the dynamic execution evolves, e.g., in Figure 2 (a), the `next` pointer is initially assigned a data edge pointing to `NULL` and later a child edge pointing to `new_node`. The online invariant detection characterizes the data structure based on a single type for each edge though, thus Figure 2 (b) shows classification system for edges. When a store instruction initially creates an edge, it starts in one of the three states. Upon encountering future stores that adjust the initial edge, the edge type may be updated. For example, if the new store address and data are both pointers from the same allocation site, the edge becomes a child edge, no matter what the previous state was. However, if the edge was already a child edge, then storing a pointer from another allocation site will not change the edge type.

The reason for this can be explained using the example from Figure 2 again. Initially the `next` pointer in a newly initialized node may contain the constant `NULL`, i.e., a data edge, and later on during execution `next` will be overwritten with `new_node` from the same allocation site, i.e., a child edge. Once `next` is overwritten again, DDT can produce more meaningful results if it remembers that the primary purpose of `next` is to point to other internal portions of the data structure, not to hold special constants, such as `NULL`. The prioritization of child edges above foreign edges serves a similar purpose, remembering that a particular edge is primarily used to link internal data structure nodes rather than external data.

Figure 3 gives an example demonstrating why typing nodes and edges in the memory graph is critical in recognizing data structures. The code snippet in this figure creates a `vector` with four `lists` and inserts integer numbers between 0 and 19 into each `list` in a round robin manner. Nodes are colored differently based on their allocation site, and edges types are represented by different arrow structures. To identify the entire data structure, DDT first recognizes the shape of a basic data structure for each allocation site by investigating how the “child” edges are connected. Based on the resulting graph invariants, DDT infers there are two types of basic data structures, `vector` and `list`. Then, DDT checks each “foreign” edge to identify the relationship between the detected data structures. In this example, all the elements of `vector` point to a memory node of each `list`, which is a graph invariant. Without the node or edge typing it would be impossible to infer that this is a composite vector-of-lists instead of some type of tree, for example.

One potential drawback of this approach is that typing of edges and nodes is input dependent, and therefore some important edges may not be appropriately classified. For example, even though an application uses a binary tree, DDT may report it is linked-list if all the left child pointers of the tree have `NULL` values due to a particular data insertion pattern. However, our experimental analysis demonstrated no false identifications for this reason, and if a binary tree were behaving as a linked-list, this pathological behavior would be very useful for a developer to know about.

### 3.2 Identifying Interface Functions for the Memory Graph

Understanding how data is organized through the memory graph is the first step toward automatically identifying data structures, but DDT must also understand how that data is retrieved and manipulated. To accomplish this DDT must recognize what portions of code access and modify the memory graph. DDT makes the assumption that this code can be encapsulated by a small set of *interface functions* and that these interface functions will be similar for all implementations of a particular data structure. E.g., every linked-list will have an insertion function, a remove function, etc. The intuition is that DDT is trying to identify the set of functions an application developer would use to interface with the data structure.

Identifying the set of interface functions is a difficult task. One cannot simply identify functions which access and modify the mem-

---

**Figure 3:** (a) Code snippet of the program using a `vector` of `lists` and (b) its memory graph.
ory graph, because often one function will call several helper functions to accomplish a particular task. For example, insertions into a set implemented as a red-black tree may call an additional function to rebalance the tree. However, DDT is trying to identify set functionality, thus rebalancing the tree is merely an implementation detail. If the interface function is identified too low in the program call graph (e.g., the tree rebalancing), the “interface” will be implementation specific. However, if the interface function is identified too high in the call graph, then the functionality may include operations outside standard definitions of the data structure, and thus be unmatchable against DDT’s library of standard data structure interfaces.

Figure 4 (a) shows an example program call graph for a simple application using the vector class from the C++ Standard Template Library, or STL [20]. In the figure each oval represents a function call. Functions that call other functions have a directed edge to the callee. Boxes in this figure represent memory graph accesses and modifications that were observed during program executions. This figure illustrates the importance of identifying the appropriate interface functions, as most STL data structures’ interface methods call several internal methods with call depth of 3 to 9 functions. The lower level functions calls are very much implementation specific.

To detect correct interface functions, DDT leverages two characteristics of interface functions. First, functions above the interfaces in the call graph never directly access data structures; thus if a function does access the memory graph, it must be an interface function, or a successor of an interface function in the call graph. Figure 4 demonstrates this property on the call graph for STL’s vector. Boxes in this figure represent memory graph accesses. The highest nodes in the call graph that modify the memory graph are colored, representing the detected interface functions.

It should be noted that when detecting interface functions, it is important to consider the memory node type that is being modified in the call graph. That is, if an interface function modifies a memory graph node from a particular allocation site, that function must not be an interface for a different call site. This intuitively makes sense, since the memory node types represent a particular data structure, and each unique data structure should have a unique interface.

You can see that finding the highest point in the call graph that accesses the memory graph is fairly accurate. There is still room for improvement, though, as this method sometimes identifies interface functions too low in the call graph. For example, __m_insert_aux is identified in this example.

The second characteristic used to detect interface functions is that generally speaking, data structures do not modify the data. Data is inserted into and retrieved from the data structure, but that data is rarely modified by the structure itself. That is, the data is immutable. Empirically speaking, most interface functions enforce data immutability at the language-level by declaring some arguments const. DDT leverages this observation to refine the interface detection.

For each detected interface function, DDT examines the arguments of the function and determines if they are modified during the function using either dataflow analysis or invariant detection. If there are no immutable arguments, then the interface is pushed up one level in the call graph, and the check is repeated recursively. The goal is to find the portion of the call graph where data is mutable, i.e., the user portion of the code, thus delineating the data structure interface.

Using the example from Figure 4, __m_insert_aux is initially detected as an interface function. However, its parent in the call graph, push_back, has the data being stored as an immutable argument as described in Figure 4 (b). In turn, DDT investigates, its parent, foo to check whether or not it is real interface function. Even if foo has the same argument, it is not immutable. Thus DDT finally selects push_back as an interface function. Detecting immutability of operands at the binary level typically requires only liveness analysis, which is a well understood compiler technique. When liveness is not enough, invariant detection on the function arguments can provide a probabilistic guarantee of immutability. By detecting memory graph modifications, and immutable operands DDT was able correctly to detect that the yellow-colored ovals in Figure 4 (a) are interface functions for STL’s vector.

One limitation of the proposed interface detection technique is that it can be hampered by compiler optimizations such as function inlining or procedure boundary elimination [24]. These optimizations destroy the calling context information used to detect the interface. Future work could potentially address this by detecting interfaces from arbitrary sections of code, instead of just function boundaries. A second limitation is that this technique will not accurately detect the interface of data structures that are not well encapsulated, e.g., a class with entirely public member variables accessed by arbitrary pieces of code. However, this situation does not commonly occur in modern applications.
Target Variables of Invariant Detection: The step first step of invariant detection for interface functions is defining what variables DDT should detect invariants across. Again, we are primarily concerned with how functions augment the memory graph, thus we would like to identify relationships of the following variables before and after the functions: number of memory nodes, number of child edges, number of data edges, and data pointer. The first three variables are used to check if an interface is a form of insertion. The last two variables are used to recognize the relationship between the data value and the location it resides in, which determines how the value affects deciding the location that accommodates it.

As an example, consider the STL deque’s\(^2\) interface functions, push_front and push_back. DDT detects interesting invariant results from the target variables mentioned above, as shown on the left side of Figure 5. Since the STL deque is implemented using dynamic array, number of memory nodes and number of child edges remain unchanged when these interface functions are called. DDT recognizes that these interface functions insert elements; however, because number of data edges, represented as ‘data_edges’ in the figure, increases whenever these functions are called. In the push_front, data pointer decreases while it increases in the push_back, meaning that data insertion occurs in the head and tail of the deque, respectively. That lets us know this is not an STL vector because vector does not have the push_front interface function.

The right side of Figure 5 shows another example of the seven invariants DDT detects in STL set’s interface function insert. The first two invariants imply that the insert increases number of memory nodes and number of child edges. That results from the fact the insert creates a new memory node and connects it to the other nodes. In particular, the third invariant, “2 * number of memory nodes - number of child edges - 2 == 0,” tells us that every two nodes are doubly linked to each other by executing the insert function. The next three invariants represent that the value in a memory node is always larger than the first child and smaller than the other child. This means the resulting data structure is similar to a binary tree. The last invariant represents that there is a data value that always holds one or zero. STL set is implemented by using red-black tree in which every node has a color value (red or black), usually represented by using a boolean type.

Similar invariants can be identified for all interface functions, and a collection of interface functions and its memory graph uniquely define a data structure. In order to detect invariants, the instrumented application prints out the values of all relevant variables to a trace file before and after interface calls. This trace is post-processed by the Daikon invariant detector [7] yielding a print out very similar to that in Figure 5. While we have found invariants listed on the graph variables defined here to be sufficient for identifying many data structures, additional variables and invariants could easily be added to the DDT framework should they prove useful in the future.

\(^2\)deque is similar to a vector, except that it supports constant time insertion at the front or back, where vector only supports constant time insertion at the back.

Figure 5: Invariant detection examples of interface functions; (a) STL deque, (b) STL set.
3.4 Matching Data Structures in the Library

DDT relies on a library of pre-characterized data structures to compare against. This library contains memory graph invariants, a set of interface functions, and invariants on those interface functions for each candidate data structure. The library is comprised of a hand-constructed decision tree that checks for the presence of critical invariants and interface functions in order to declare a data structure match. That is, the presence of critical invariants and interface functions is tested, and any additional invariants/interfaces to not override this result.

The invariants are picked that distinguish essential characteristics of each data structure, based on its definition rather than on implementation. That is, for a linked list, the decision tree attempts to look for an invariant, “an old node is connected to a new node” instead of “a new node points to NULL.” The latter is likely to be implementation specific. Intuitively, the memory graph invariants determine a basic shape of data structures, e.g., each node has two child edges. Meanwhile, the invariants of interface functions distinguish between those data structures which have similar shapes.

At the top of the decision tree, DDT first investigates the basic shape of data structures. After the target program is executed, each memory graph that was identified will have its invariants computed. For example, an STL vector will have the invariant of only having a single memory node. With that in mind, DDT guesses the data structure is array-like one. This shape information guides DDT into the right branch of the decision tree in the next to check desired function invariants.

Among the detected interface functions, DDT initially focuses on insert-like functions. That is because most data structures have at minimum an insertion interface function, and they are very likely to be detected regardless of program input. If the required interface are not discovered, DDT reports that the data structure does not match. After characterizing the insertion function, DDT further investigates other function invariants traversing down the decision tree to refine the current decision. As an example, in order to determine between deque and vector, the next node of the decision tree investigates if there is the invariant corresponding to push_front as shown in Section 3.3. It is important to note that the interface functions in the library contain only necessary invariants. Thus if the dynamic invariant detection discovers invariants that resulted only because of unusual test inputs, DDT does not require those conservative invariants to match what is in the library.

Figure 6 shows a portion of DDT’s decision tree used to classify binary trees. At the top of the tree, DDT knows that the target data structure is a binary tree, but it does not know what type of binary tree it is. First, the decision tree checks if there is the invariant corresponding to a “binary search tree”. If not, DDT reports that the data structure is a simple binary tree. Otherwise, it checks if the binary tree is self-balancing. Balancing is implemented by tree rotations and they are achieved by updating child edges of pivot and root, shown in the top-left of Figure 6. The rotation function is detected by the invariant that two consecutive and different “child” edges are overwritten (shown in bold in Figure 6). If tree-rotation is not detected in the insert, DDT reports that the data structure is a “simple binary search tree.” More decisions using the presence of critical functions and invariants further refine the decision until arriving at the leaf of the decision tree, or a critical property is not met, when DDT will report an unknown data structure. After data structures are identified, the decision tree is repeated using any “foreign” edges in the graph in order to detect composite data structures, such as vector<list<int>>.

Using invariant detection to categorize data structures is probabilistic in nature, and it is certainly possible to produce incorrect results. However, this approach has been able to identify the behavior of interface functions for several different data structure implementations from a variety of standard libraries, and thus DDT can be very useful for application engineering. Section 4 empirically demonstrates DDT can effectively detect different implementations from several real-world data structure libraries.

4. EVALUATION
In order to demonstrate the utility of DDT, we implemented it as part of the LLVM toolset. DDT instruments the LLVM intermediate representation (IR), and the LLVM JIT converts the IR to x86 assembly code for execution. Output from the instrumented code is then fed to Daikon [7] to detect invariants needed to identify data structures. These invariants are then compared with a library of data structures that was seeded with simple programs we wrote using the C++ Standard Template Library (STL) [20]. The entire system was verified by recognizing data structures in toy applications that we wrote by hand without consulting the STL implementation. That is, we developed the classes MyList, MySet, etc. and verified that DDT recognized them as being equivalent to the STL implementations of list, set, etc. Additionally, we verified DDT’s accuracy using four externally developed data structure libraries: the GNOME project’s C-based GLib [21], the Apache C++ Standard Library STDCXX [22], Borland C++ Builder’s Standard Library STLport [21], and a set of data structures used in the Trimaran research compiler [25].

Even though the current implementation of DDT operates on compiler IR, there is no technical issue preventing DDT’s implementation on legacy program binaries. The LLVM IR is already very close to assembly code, with only two differences worth addressing. First, LLVM IR contains type information, which is implemented using a red-black tree. To accomplish this, the sole incorrect identification was for GLib’s GTree, which is implemented as an AVL tree. DDT correctly identified that the set from STL, STDCXX, STLport were all implemented using a red-black tree. To accomplish this, DDT correctly identified that the set from STL, STDCXX, STLport were all implemented using a red-black tree. To accomplish this, DDT correctly identified that the set from STL, STDCXX, STLport were all implemented using a red-black tree. To accomplish this, DDT correctly identified that the set from STL, STDCXX, STLport were all implemented using a red-black tree. To accommodate this, DDT also detected that Trimaran’s Set exploits list-based implementation and GLib’s GQueue is implemented using a doubly-linked list.

Table 1 shows how DDT correctly detects a set of data structures from STL, STDCXX, STLport, GLib, and Trimaran. The data structures in this table were chosen because they represent some of the most commonly used, and they exist in most or all of the libraries examined. However, the overhead for instrumenting the code to recognize data structures was about 200X. The dynamic instrumentation overhead for memory/call graph generation was about 50X while the off-line analysis time including interface identification and invariants detection occupies the rest of the overhead. In particular, the interface identification time was sufficiently negligible that it occupies less than 3% of the whole overhead. While this analysis does take a significant amount of time, it is perfectly reasonable to perform heavy-weight analysis like this during the software development process.

5. SUMMARY
The move toward manycore computing is putting increasing pressure on data orchestration within applications. Identifying what data structures are used within an application is a critical step toward application understanding and performance engineering for

<table>
<thead>
<tr>
<th>Library</th>
<th>Data structure type</th>
<th>Main data structure</th>
<th>Reported data structure</th>
<th>Identified?</th>
</tr>
</thead>
<tbody>
<tr>
<td>STL</td>
<td>vector</td>
<td>dynamic array</td>
<td>vector</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>deque</td>
<td>double-ended dynamic array</td>
<td>deque</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>list</td>
<td>doubly-linked list</td>
<td>doubly-linked list</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>set</td>
<td>red-black tree</td>
<td>red-black tree</td>
<td>yes</td>
</tr>
<tr>
<td>Apache (STDCXX)</td>
<td>vector</td>
<td>dynamic array</td>
<td>vector</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>deque</td>
<td>double-ended dynamic array</td>
<td>deque</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>list</td>
<td>doubly-linked list</td>
<td>doubly-linked list</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>set</td>
<td>red-black tree</td>
<td>red-black tree</td>
<td>yes</td>
</tr>
<tr>
<td>Borland (STLport)</td>
<td>vector</td>
<td>dynamic array</td>
<td>vector</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>deque</td>
<td>double-ended dynamic array</td>
<td>deque</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>list</td>
<td>doubly-linked list</td>
<td>doubly-linked list</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>set</td>
<td>red-black tree</td>
<td>red-black tree</td>
<td>yes</td>
</tr>
<tr>
<td>GLib</td>
<td>GArray</td>
<td>double-ended dynamic array</td>
<td>de que</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>GQueue</td>
<td>doubly-linked list</td>
<td>doubly-linked list</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>GSLList</td>
<td>singly-linked list</td>
<td>singly-linked list</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>GTree</td>
<td>AVL tree</td>
<td>balanced binary search tree</td>
<td>no</td>
</tr>
<tr>
<td>Trimaran</td>
<td>Vector</td>
<td>dynamic array</td>
<td>vector</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>List</td>
<td>singly-linked list</td>
<td>singly-linked list</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>Set</td>
<td>singly-linked list</td>
<td>singly-linked list</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 1: Data structure detection results of representative C/C++ data structure libraries.
the underlying manycore architectures. This work presents a fundamentally new approach to automatically identifying data structures within programs.

Through dynamic code instrumentation, our tool can automatically detect the organization of data in memory and the interface functions used to access the data. Dynamic invariant detection determines exactly how those functions modify and utilize the data. Together, these properties can be used to identify exactly what data structures are being used in an application, which is the first step in assisting developers to make better choices for their target architecture. This paper demonstrates that this technique is highly accurate across several different implementations of standard data structures. This work can provide a significant aid for assisting programmers in parallelizing their applications.

6. REFERENCES
Factoring Out Ordered Sections to Expose Thread-Level Parallelism

Hans Vandierendonck, Sean Rul and Koen De Bosschere
Ghent University,
Dept. of Electronics and Information Systems,
Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium
{hvdieren,srul,kdb}@elis.ugent.be

Abstract

With the rise of multi-core processors, researchers are taking a new look at extending the applicability auto-parallelization techniques. In this paper, we identify a dependence pattern on which auto-parallelization currently fails. This dependence pattern occurs for ordered sections, i.e. code fragments in a loop that must be executed atomically and in original program order. We discuss why these ordered sections prohibit current auto-parallelizers from working and we present a technique to deal with them. We experimentally demonstrate the efficacy of the technique, yielding significant overall program speedups.

1. Introduction

Chip manufacturers have made a shift towards building multi-core processors. This puts a large demand on techniques for extracting thread-level parallelism (TPP) from applications. As these multi-core processors are shipped in all kinds of systems (server, desktop and laptop), it has become necessary to exploit TPP also in applications that have traditionally not been considered as good candidates for multi-threaded execution: programs with complex control flow and hard-to-predict memory access patterns.

We have found that, in these applications, TPP is often limited due to small details. In particular, a loop nest can be highly parallel, except for a few statements in the loop that introduce a dependency between loop iterations. Yet, these statements are so few (or they are perhaps never executed in practice), that parallelisation should be possible, at least to some extent. Some of these dependence patterns are well known, e.g. reductions and critical sections, and solutions have been in use for a long time.

The contribution of this paper is to provide a solution to a more complicated case: the case where operations must remain in program order (more stringent than critical sections), but the parallelized code region does not depend on values computed by these operations. These ordered sections may be very benign, such as the generation of debugging output when a flag is set. On the other hand, they may also concern updating critical data structures. We present a method to allow parallelization of this construct.

Moreover, we demonstrate that our method also helps to deal with respecting the order of system calls and dealing with multiple call sites to exit() by callee functions. These issues are very important for dealing with program side-effects and for respecting sequential program semantics.

We demonstrate the effectiveness of our method on several benchmarks (bzip2 from SPECint2000, mcf from SPECint2006, clustalw from BioPerl). The intent of this work is to present methods than can be implemented in auto-parallelizing compilers. While we are working on such a compiler, actual compiler algorithms and implementation are out of the scope of this work.

This paper is structured as follows. In Section 2 we discuss the researched problem and we present our solution in Section 3. We experimentally evaluate our solution in Section 4. We discuss related work in Section 5 and summarize conclusions in Section 6.
2. The Problem

It is recognized that programs with complex control-flow and/or complex memory access patterns require specific types of parallelism. Rather than DOALL loops, one should search for pipelines \[9, 12\], parallel-stage pipelines \[10\] and task parallelism. Furthermore, some researchers advocate using speculation to reduce the code structure to one of these types of parallelism and thus expose parallelism \[4, 13\].

In any of the cases above, we demand that dependencies between statements respect a particular pattern. This is easily represented using the program dependence graph (PDG) \[6\], where statements are represented as nodes and dependencies between statements are represented as directed edges. Edges are inserted for control and data dependencies, and also for memory dependencies. Parallelism is detected in the PDG by computing strongly connected components (SCC) in the graph \[9\]. The cited types of parallelism can be graphically represented by the dependencies between the strongly connected components (Figure 1).

However, more often than not, the parallel code regions may contain code fragments that introduce dependencies and inhibit the exploitation of parallelism. These code fragments may be very benign, such as the generation of debugging output when a flag is set, or they may be updates to data structures that must be executed in the original program order. Hence, we call these code fragments ordered sections, in correspondence to the same term in OpenMP \[8\]. In this paper, we focus on particular ordered sections, namely those that do not produce values consumed by the remainder of the loop or task. As such, there is a certain amount of slack in executing these ordered sections and an opportunity for exploiting parallelism arises.

To illustrate the problem, we draw upon an example. The pseudo-code in Figure 2 describes a parallelizable loop with ordered sections. The loop takes input data \(x\) and transforms it into output data \(z\) in two steps. We assume that a good parallel partitioning of the loop places functions \(f(\cdot)\) and \(g(\cdot)\) in separate pipeline stages. This pipeline, however, is obscured by the presence of ordered sections in these functions. Each of the ordered sections draws upon some values produced by the main code and they update a shared global variable \(G\). Function calls are represented as a single node in the program dependence graph. Hereby, they are treated as an undivisible entity. It is no longer possible to separate the ordered sections from the remainder of the code. The corresponding PDG is depicted in Figure 3(a). We observe that nodes \(f(\cdot)\) and \(g(\cdot)\) are cyclically dependent through variable \(G\).

Note that the cyclic dependence is there only because the ordered sections are located in functions called from the loop. Hence, the ordered sections are necessarily merged together with the remainder of the

---

**Figure 2.** Pseudo-code for a parallelizable loop with ordered sections.
functions \(f(\cdot)\) and \(g(\cdot)\). The problem disappears when, for instance, we inline the bodies of the functions into the loop. The corresponding PDG (Figure 3(a)) contains two nodes for each function: the operate step (op) and the ordered section (OS). The ordered sections still reduce to a single SCC. As the remainder of the code is no longer dependent on the ordered sections, the pipeline can be readily seen.

Note that the situation of Figure 3(b) corresponds to loops where ordered sections do not occur in callee functions, a situation that is parallelizable by prior approaches [9, 5, 13]. In this work, we propose a scalable solution to the general problem of ordered sections in callee functions. Function inlining is not a suitable solution to this problem; we used it only for didactical reasons. Function inlining is not suitable because (i) function inlining is an optimization in its own right with sensitive cost/performance models, and (ii) it is not sufficiently generic in the face of deep call trees and (mutually) recursive functions.

3. The Solution

Conceptually, the proposed solution is to remove ordered sections from callee functions and to move them to the loop body. Hereby the necessary dependences between function calls remain while the ordered sections reduce to an SCC without outgoing edges. The method to achieve this is to create a queue of ordered sections remaining to be processed. Ordered sections are enqueued when a callee function would have entered one. They are dequeued only when program dependences allow so, typically in the last stage of the pipeline.

3.1. Dissecting Ordered Sections

Three properties of ordered section are important to describe how ordered sections are factored out.

The **update set** of an ordered section is the set of all program variables that are modified by the ordered section, i.e. basic blocks and instructions. The factored code contains all the instructions operating on the update set and all of their dependents. Control flow instructions between basic blocks must be properly duplicated. Separating factored code from the remainder of the code is actually quite similar to splitting a loop body in pipeline stages and can be handled in the same way, e.g. [9].

The **factored code** of an ordered section is a fragment of the control flow graph of the containing function, i.e. basic blocks and instructions. The factored code contains all the instructions operating on the update set and all of their dependents. Control flow instructions between basic blocks must be properly duplicated. Separating factored code from the remainder of the code is actually quite similar to splitting a loop body in pipeline stages and can be handled in the same way, e.g. [9].

The factored code may not include function return statements as this would violate the proposition that the majority of the code is not dependent on the ordered section. Furthermore, ordered sections do not cross function boundaries for reasons of simplicity.

3.2. Operations on the PDG

When factoring out an ordered section from a callee function, we must update the program dependence graph to reflect a reduction of dependencies for the function call node.

Here, we add a **consuming call node** to the PDG, besides the original function call node. The original call node represents the non-ordered section part of the function, plus the code to enqueue ordered sections. The consuming call node represents taking ordered sections from the queue and executing them.

Dependencies in the PDG are updated in the following way:
1. All outgoing dependencies from the original call node that indicate updates to a variable in the update set of the ordered section are redirected to start from the consuming call node.

2. All incoming dependencies to the original call node that indicate dependence on a variable in the update set of the ordered section are redirected to point to the consuming call node.

3. A dependence is added from the original call node to the consuming call node to indicate the causality between producing ordered sections in the queue and consuming them. All incoming dependencies of the ordered section that are not part of its update set are implicitly captured by this queue dependence.

There can be only one consuming call node for every original call node, even when multiple ordered sections are factored out. In the case of multiple ordered sections, the PDG is updated by steps 1 and 2.

3.3. Code Transformation

Hopefully, the modified PDG will expose additional parallelism. When parallelism is found and parallel code is generated, additional code must be included to factor out ordered sections and to consume the ordered sections.

To enqueue a message for an ordered section, a message is constructed containing an ordered section ID (each ordered section is assigned a unique ID to identify it) and a copy of all the variables in its copy set.

In the functions, instructions belonging to ordered sections are removed. Instead, code is inserted to queue the ordered section. The queued information consists of an ordered section ID and a copy of every program variable in its copy set.

Finally, additional code is added to the main loop to dequeue ordered sections. This code consists of a loop that takes every ordered section from the queue and executes the corresponding code using program variables from the copy set where necessary. When the number of enqueued ordered sections is expected to be small, then it is possible to add the consuming loop to the last pipeline stage of the loop. In this case, the queue grows to its maximum size during the execution of each pipeline stage. The queue is emptied only after the last pipeline stage has executed.

The general solution is to create an additional thread that executes ordered sections as they are inserted in the queue. The thread starts when the first pipeline stage starts executing. It stops by a special message that is sent when the last pipeline stage finishes. It is likely that this thread is idle very often. It is needed only to limit the size of the queue.

When the size of the ordered section queue is expected to be bounded, and because it is often the case that ordered sections are responsible for only a fraction of the code executed in the loop, the first approach works quite well. It is used in all the examples in the evaluation section.

However, when a queue grows beyond acceptable size, it is possible to limit memory consumption by blocking a thread when it tries to add additional elements to the queue. When all previous loop iterations have finished, the ordered sections may be dequeued and executed and the thread may continue execution.

An execution chart of the resulting code transformation is depicted in Figure 4 for a 3-stage pipeline. Code block $S_{p,i}$ executes pipeline stage $p$ for loop iter-

![Figure 4. Execution chart of loop with pipeline parallelism and factored ordered sections.](image-url)
ation $i$. Code blocks $S_{1,i}$ and $S_{2,i}$ may insert ordered sections in queue $Q_i$, which is specific to loop iteration $i$. Ordered sections are consumed in step $OS_i$, before executing step $S_{3,i}$. The latter pipeline stage directly executes ordered sections (this is an optimization to avoid unnecessary queueing overheads). Note that the queue for iteration $i + 1$ is not consulted before iteration $i$ has completely executed. This timing is necessary to respect the semantics of ordered sections.

3.4. Handling System Calls

Ordered sections help dealing with system calls. Again, these remarks apply to system calls embedded in functions called from a parallelized loop.

Compilers generally treat system calls with great conservatism, causing all system calls embedded in functions called from a loop to be mutually dependent. Such a situation kills parallelism. Building on ordered sections, it is possible to ameliorate this situation. One starts by trying to factor out every system call as an ordered section. This operation fails when the system call produces input data for the loop, e.g. a `read()` call.\footnote{Conservatism requires that calls to `read()` are treated as aliased to other system calls which may interfere, even `write()` on a different file descriptor. E.g. the program may be in communication with another program over a UNIX pipe, causing reads to block on writes from the other program. Interchanging reads and writes on such a program may lead to deadlock.}

If, however, all system calls can be factored out, then conservatively correct thread-level parallelism may be exposed.

Furthermore, when functions called from a parallelized loop contain multiple calls to `exit()` or `abort()`, then ordered sections help to provide a correct implementation. Without recognizing program exits, it is possible that the wrong exit is taken and perhaps the wrong error message is printed. This can happen, e.g., when pipeline stage $S_{1,n}$ executing iteration $i$ exits the program before pipeline stage $S_{2,1}$ is executed and has the opportunity to exit.

Handling multiple exits is straightforward using ordered sections: every function call that may not return is an ordered section and is factored out. Furthermore, at the point of executing the non-returning function call, the thread sets a flag that the current iteration $i$ is finished, nullifying all code in pipeline stages $S_{p,i}$. Furthermore, all threads executing pipeline stages $q ≤ p$, where $p$ is the current pipeline stage, are blocked. The last pipeline stage continues execution and executes the first exit it encounters, respecting sequential semantics.

4. Experimental Evaluation

We evaluate the proposed code transformation for exposing thread-level parallelism using 3 benchmarks: bzip2 (taken from SPEC CPU2000\footnote{http://www.spec.org/}), mcf (SPEC CPU2006) and clustalw (BioPerf [1]).

We test these benchmarks on an Intel I7 quad-core processor (8 threads in total) and a Sun Niagara T1 8-core processor (32 threads in total). We use gcc 4.1.2 on the Niagara and gcc 4.3.2 on the Intel I7. Parallelism is expressed using POSIX or OpenMP. When using OpenMP, we compile with gcc 4.4 to have OpenMP 3.0 support.

4.1. Bzip2

The main compression loop in bzip2 (SPECint2000) is a 4-stage pipeline, where stages 2 and 3 are parallel stages [11]. Many functions in bzip2, however, may print debugging information depending on the value of a verbosity flag. To reconcile the goals of parallelizing the code and guaranteeing the correct ordering of print statements, we isolate these print statements in separate tasks using the method of this paper. Hereby, the pipeline becomes valid. The same transformation is applied to the 2-stage pipeline in the decompression code.

We implemented the parallel pipelines using the POSIX threads library [11]. The timing measurements (Figure 5) reveal that the compression stage benefits from up to 6 threads on the Niagara processor and up to 4 threads on the I7. Decompression can benefit from at most 2 threads due to the structure of the pipeline. Overall, a speedup of 2.97 and 2.00 is obtained on the Niagara and I7 processors, respectively.

4.2. Mcf

An almost DO-ALL loop occurs in the `primal_bea_mpp()` function of the mcf benchmark. This loop scans over the edges in a graph and
Figure 5. Execution time of bzip2 on two multi-core processors.

Figure 6 shows that performance scales to a 1.52 speedup with 4 threads on the I7 and to 2.18 with 16 threads on the Niagara T1.

4.3. Clustalw

Clustalw spends almost all its execution time in two stages: pairwise alignment and progressive alignment [14].

Pairwise alignment of sequences using the Smith-Waterman algorithm is trivially parallel as the alignment of every pair of sequences is independent. However, the code prints the score of each pair of sequences as it progresses, hence current compiler techniques do not recognize this parallelism. We use the same code transformation as in bzip2 to enable the parallelism. Figure 7 shows the performance scaling of this highly parallel loop, which shows near-perfect scaling.

The progressive alignment stage contains much less parallelism. The bulk of the computation is in a function \( \text{pdiff}() \) where two loop nests can be executed in parallel. Figure 7 shows that this parallelism reduces progressive alignment execution time from 17.0 seconds to 10.0 seconds, using 2 threads.

Additional parallelism is present between the doubly-recursive calls in this function, as each call is almost independent of the other calls. The term “most” refers to a serialization that occurs due to updates to a shared array (\( \text{displ[]} \)) that occurs mostly in the leaf calls of the recursion and sometimes in between the two recursive calls. With the technique of this paper and using 4 threads, performance improves by 5.4% (Figure 7). Although this speedup is not very high, the aim of the exercise is to show that the parallelism can be correctly extracted.

Performance can probably be improved with more implementation work. We used the OpenMP task construct to express the parallelism, but OpenMP schedules tasks in a sub-optimal order, especially since tasks are also used to express the parallelism between the loop nests. Performance is significantly improved if processors are allocated in pairs, one for each loop.

\[^{3}\text{Nesting OpenMP sections in tasks turned out to perform worse due to repetitive creation and destruction of threads.}\]
nest, a trick we applied in our implementation for the Cell B.E. [14].

5. Related Work

The context of this work is the search for ways to automatically parallelize control-flow intensive applications. Decoupled software pipelining is a compilation technique to recognize pipelines and to distribute the pipeline stages across threads [9]. Parallel-stage pipelines are pipelines where some stages are not dependent on themselves and allow additional parallelism [10, 11]. Program demultiplexing attempts to extract threads by viewing a sequential program as a set of interleaved threads [2]. All of these approaches succeed in discovering TLP to some extent. In our experience, however, they get stuck on particular dependence patterns, one of which is discussed in this paper.

Thread-level speculation (TLS) aims to expose TLP that is not provably correct [7, 3]. By adding hardware and/or software checkpointing and restore mechanisms for memory, it is possible to undo the effects of mis-speculation. Many proposals of TLS mechanisms can, however, not expose the same coarse-grain parallelism as the proposed technique can. For instance, speculative threads may not perform side-effects that cannot be undone, e.g. I/O. Also, TLS executes ordered sections speculatively, making these an important source of thread squashes. These thread squashes may be avoided with the technique proposed in this paper.

Copy-or-Discard [13] is a software TLS technique that exploits parallel-stage pipeline parallelism speculatively. The code transformation proposed by the authors performs a similar transformation as the one proposed in this paper: Instructions that are likely part of a cross-iteration dependence are moved to a sequentially executed epilogue for the loop. In their case, however, they base themselves on profile information to determine what code to move. Consequently, if a dependence is not caught by profiling information, then the dependent instructions remain in the parallel loop body and speculation will fail. This work, in contrast, yields parallel speedups whether the dependence is excised or not.

The IPOT programming model [15] provides annotations for identifying transactions in programs, which are executed by an underlying transactional substrate. IPOT provides several annotations that allow the programmer to identify cross-transaction dependences, which can reduce dependence violations on ordered sections. These include reduction patterns, which are a particular type of ordered section, and race conditions that do not impact the program outcome (e.g. updates to a cut-off limit in branch and bound algorithms).

The term ordered section is based on the OpenMP construct that indicates that a critical section must execute in the original program order. In OpenMP, however, ordered sections are limited to loops and at most one ordered section may exist per loop [8]. With our technique, we allow multiple ordered sections per loop that are strung together in the correct program order. Furthermore, our discussion of ordered sections also applies to non-DOALL loops and to task parallelism.
6. Conclusion

Ordered sections are code fragments that must be executed in original program order. Within an otherwise parallel loop, they can strongly inhibit the efficient exploitation of parallelism.

This paper presents a method for efficiently executing such ordered sections in the case where the remainder of the loop is not dependent on the values computed in the ordered sections. We extract the ordered sections into tasks and we copy part of the data environment, if necessary. When executing an ordered section, a task is generated for it and placed in queues. Finally, the tasks are taken from the queues in sequential program order and are executed.

We demonstrate the efficacy of this technique on several benchmarks, allowing the parallelization of loops that are otherwise not parallelizable. Scalability of the parallelization is discussed on two multicore processors: a quad-core Intel I7 and a 32-thread Sun Niagara processors. We are currently working on implementing the proposed code transformation in a compiler.

References


Dynamic Concurrency Discovery for Very Large Windows of Execution

Jacob Nelson       Luis Ceze
Computer Science and Engineering
University of Washington
{nelson, luisceze}@cs.washington.edu

Abstract

Dynamically finding parallelism in sequential applications with hardware mechanisms is typically limited to short windows of execution due to resource limitations. This is because precisely keeping track of memory dependences is too expensive. We propose trading off precision for efficiency. The key idea is to encode a superset of the dependences in a way that saves storage and makes traversal for concurrency discovery efficient. Our proposal includes two alternative hardware structures: the first is a FIFO of Bloom Filters; the second is an imprecise map of memory addresses to timestamps that summarizes dependences on-the-fly. Our evaluation with SPEC2006 applications shows that they lead to little imprecision in the dependence graph and find similar parallelization opportunities as an exact approach.

1. Introduction

Multicore microprocessors are ubiquitous today, and exploiting their potential requires parallel programs. But legacy sequential code still exists, and new sequential code is still being written. Parallelizing this code automatically (e.g., speculative multithreading) or manually benefits from information about where effort might be most fruitful. Data dependences define where concurrency, and thus opportunities for parallelization, exist.

Dependence detection to find parallelization opportunities has been done in many contexts before. Compilers can do it, but often must resort to conservative estimates without runtime information. Superscalar processors capture data dependences dynamically, but in windows of hundreds of instructions.

The goal of this work is to enable collection of memory dependences over windows of billions of instructions, with little resource usage and no performance overhead. We make three contributions. First, we introduce the concept of imprecise dependence capture, which trades accuracy in the captured dependence data for efficiency in the capture hardware. Second, we propose two hardware structures that use this concept to enable dependence capture over very large windows. Third, we demonstrate that these structures are able to find fruitful concurrency even with imprecision.

Our technique is applicable to a number of problems benefiting from dynamic memory dependence information, including dependence profiling, task recommendation for programmers [11], and speculative multithreading. Figure 1 shows one context in which our dependence collection structures could be applied: a system for dynamic speculative multithreading. The hardware would profile control and data dependences and communicate these to the runtime, which would use this information to partition the program into tasks for speculative execution. This paper focuses on capturing data dependences through memory from a sequential execution of a program; a complete system would combine this with other techniques to obtain a complete view of a program’s execution, select tasks, and ensure correct speculative execution.

2. Related work

There have been several pieces of work on dynamic collection of dependence information. We divide the space of prior work into three categories: (1) using profiler-collected information for program decomposition [6, 9]; (2) monitoring mis-speculation events in speculative multithreading [7, 5]; and (3) hardware support for efficient on-line profiling [3, 12]. (2) and (3) are the most relevant to our work.

Prior work on speculative multithreading has explored using information on mis-speculations to guide better task formation (e.g., [7, 4]). This indirectly produces information about dependences, since dependences are the cause of mis-speculations.
However, the information that can be inferred is limited by the structure of tasks chosen beforehand. We approach this differently: we collect dependence information from the original sequential execution of a program, and therefore do not impose artificial restrictions on the dependence information collected.

Our work fits in the category of hardware support for efficient dependence profiling. To the best of our knowledge, the most relevant prior work was TEST [3], which proposed using timestamps per cache-line and a FIFO queue of memory references to keep track of recent accesses; dependences were computed based on that. The main limitation of that approach is the relatively short window size. Small windows limit the ability to find coarser-grain parallelism, which is what is desirable to tolerate the inter-core communication costs. We show instead that by allowing some amount of imprecision, we can design simple structures that are able to find coarse-grain parallelism in very large windows.

3. Model and Background

We model a program’s trace of execution as a directed graph. Nodes are dynamic instances of basic blocks; we refer to a particular dynamic basic block (DBB) with a dynamic block identifier (DBID), which is a tuple (BBID, timestamp) where BBID is the static basic block ID (i.e., the address of the first instruction) and the timestamp is an unique monotonically increasing value that identifies ordered instances of basic blocks. We assume all instructions execute in unit time, so the time to execute a DBB is given by its instruction count.

Edges are dynamic data dependences through memory. They impose an order on the sequence in which DBBs can be executed. By executing each DBB as early as possible in this order, we can find a lower bound on the execution time for a parallel version of the code. Some path through the graph will be the longest, defining the minimum execution time; this is the critical path. The ratio of the overall instruction count to the length of the critical path is the graph’s available parallelism.

Figure 2(a) shows a sequence of instructions divided into DBBs, along with the instructions’ exact dependences. We can reorder the sequence of execution so that each DBB runs as early as the dependence graph allows; Figure 2(b) shows this maximally hoisted graph with exact dependences. The critical path includes DBBs 1, 3, and 6 for a length of 7.

We chose to work at basic block granularity to ensure that no predefined task structure affects the data dependence information collected. Also, note that we only consider data dependences through memory in this work, since they are the hardest to detect and the biggest limiting factor in exploiting coarse-grain parallelism.

It would be ideal to identify these data dependences exactly. This is possible for small windows: simply store a history of writes along with DBIDs, and for each read, search for previous writes with matching addresses. But this fails for larger windows: some of the benchmarks we use for evaluation have millions of active variables. On a modern 64-bit machine, the history could require gigabits of on-chip storage. A more efficient solution is needed if we want to enable concurrency discovery in very large windows.

4. Imprecise dependence profiling

We will trade precision for resources and performance in computing the dependence graph. By capturing dynamic dependences that are imprecise in restricted ways, we may reduce storage requirements and search overhead while still capturing the important parts of the graph.

Our starting point will be the probabilistic set-inclusion data structure known as a Bloom Filter [1], whose clean mapping of set operations to hardware primitives has been exploited before [2, 10]. Bloom Filters may return false positives but will never return false negatives; this will be the source of our imprecision.

A Bloom Filter is a vector of \( m \) bits, along with \( k \) independent hash functions mapping a key to some subset of the bits of the vector. As in [2], we implement the \( k \) hash functions by partitioning a permuted key into \( k \) bit fields and applying an \( n \)-to-\( 2^n \) binary decoder to each \( n \)-bit field; the concatenation of all of these decoder results is the Bloom Filter’s bit vector. The size and position of each of the \( k \) bit fields, along with the choice of permutation, determines the Bloom Filter’s rate of false positives.

We present two structures for dependence detection. The first, called a Bloom History, is straightforward but resource-hungry; the second, called a Bloom Map, extends the first in
4.1. Bloom History

A simple starting point is to encode the write set and read set of each DBB as Bloom Filters and store a history of encoded write sets; if the read set Bloom Filter for a DBB intersects with one of the write set Bloom Filters for a previously-executed basic block, a dependence exists between them. This is a Bloom History.

For each DBB, the Bloom History performs three steps. As the block executes, addresses read by the block are collected in a Bloom Filter, and addresses written by the block are collected in another Bloom Filter. Then, the read set Bloom Filter is intersected with each of the previous write set Bloom Filters; any intersections are recorded as dependence edges in an edge buffer. The contents of the read set Bloom Filter are not needed after this search is complete. Finally, the write set Bloom Filter is added to the write history, and the next DBB is executed.

Figure 4 shows a simple example with two hash functions (1-bit fields of the address) stored in 4-bit Bloom Filters. The code is the same as in Figure 2. Figure 4(c) shows the Bloom Filter encoding for addresses. Figure 4(e) shows the contents of the Bloom History after all DBBs have executed. Figure 4(b) shows the read set Bloom Filter values formed to search for intersections in the Bloom History for each DBB; these values are needed only during DBB execution and are not stored.

DBBs 1 and 2 perform only writes, updating the Bloom History. DBBs 3 and 4 demonstrate the Bloom History finding dependences that match the exact graph: DBB 3’s read set is compared with the Bloom History entries for DBBs 1 and 2, where a dependences is found; DBB 4’s read set is compared with the entries for DBBs 3, 2, and 1, finding dependences from DBBs 2 and 1. DBB 5 shows imprecision; even though address A is never written in the window of execution, the combination of addresses A and D read by DBB 5 (when ORed together as part of the Bloom Filter set union) produces a read set that intersects imprecisely with the write sets of DBBs 2, 3, and 1. This is read aliasing. The Bloom Filter for DBB 6’s writes to addresses B and C intersects with DBB 7’s read of address A, illustrating write aliasing.

Figure 4(d) shows the resulting maximally-hosted dynamic dependence graph. It shares nodes 1, 3, and 6 with the exact graph but also includes 7, increasing the critical path length to 8.

Since we cannot create an arbitrarily large Bloom History, we must collect dependence information in windows. This limits the length of dependences we collect, and therefore may miss long dependences, such as the ones between iterations of large outer loops.

Bloom History Hardware. The structure of one potential hardware Bloom History is shown in Figure 5. It is organized like a shift register, storing a DBID and write set for each DBB.

As execution progresses, the read set of the currently-executing DBB is intersected with all previous write sets in parallel; when the DBB finishes executing, edges that have been identified are latched in the edge buffer and the new write set and DBID are shifted in.

Read aliasing can be avoided by encoding and searching for each read address individually. We use this technique in our evaluation.

4.2. Bloom Map

The Bloom History captures critical dependences, but it also captures many other dependences that do not limit concurrency. The Bloom Map attempts to focus its storage on the critical dependences of the graph. For a DBB on the critical path, one or more of its dependences forces it to be on the path. Often this is the DBB’s most recent (or shortest) dependence; the Bloom Map tries to store only that dependence.

The Bloom Map is a straightforward extension of a Bloom Filter that allows us to store a value at a key’s hash locations. Rather than setting bits, it stores the dynamic block identifier tuple (BBID, timestamp). When the Bloom Map is queried with an address, it returns the matching tuple with the most recent timestamp. This gives us essentially a “view” of the Bloom History: for each reading basic block, the Bloom Map tries to return the shortest dependence that the Bloom History would have found.

A Bloom Map consists of \( k \) arrays of \((BBID, timestamp)\) tuples. Each array is indexed by a different hash function. Just as a Bloom Filter uses independent hash functions to reduce the rate of false positives, the Bloom Map uses multiple arrays to reduce imprecision in returned dependences. If all arrays...
Figure 4. A sequence of executed instructions and their data dependences, split into dynamic basic blocks (a) and observed by a Bloom History (b, e) with hash mapping shown in (c). The resulting precise and imprecise maximally-hoisted dependence graphs are shown with the blocks’ instruction counts in (d).

Figure 5. Bloom History architecture.

agree that a dependence may exist, one of the stored dependence sources must be selected as the result. Careful design of this selection helps avoid imprecision in the returned dependences.

Figure 6 illustrates a Bloom Map with two arrays. During a write operation, we use the fields of the data address to index into each of the Bloom Map’s arrays. Then we write the tuple into each selected array element, discarding any previous value. Reads are more involved: to process a single read, we use the fields of the read address to index into the Bloom Map’s arrays. Then we must choose one of the selected tuples to return. Since a write to the address would have overwritten the tuples in all the arrays, by selecting the tuple with the oldest timestamp, we emulate returning the newest matching write from the Bloom History.

It is possible to process multiple reads, but an additional choice is required. First, since one hash function may point to multiple tuples in an array, we must choose one of them as the result for that array. The indexed tuples in an array may have been written by correct writes at different times: if we want to approximate returning the newest dependence to a reading DBB, we must choose the newest tuple in each array. Then we can again choose the oldest tuple between the arrays, since any correct dependence source must have updated all the arrays.
Figure 7 shows a Bloom Map with two arrays observing a simple sequence of memory operations. The code, shown in Figure 7(a), is the same as Figure 2; Figure 7(c) shows the address mapping. Since a single Bloom Map is used for the whole window, in Figure 7(b) we show the contents of the Bloom Map whenever a DBB’s writes cause an update. We show only the timestamp of the DBIDs stored in the Bloom Map. Arrows indicate which cells of the current Bloom Map state are queried by a DBB’s read set and used in choosing a final result; these arrows are not stored.

The first two DBBs contain only writes, and thus only update the Bloom Map state. DBB 2 overwrites one of the tuples written by DBB 1. When DBB 3 queries the previous state of the Bloom Map, it must then choose between DBIDs 1 and 2; choosing 1, the older dependence source, yields a dependence that matches the exact graph. DBB 3 also writes, overwriting a relevant tuple for DBB 4’s reads, but again, choosing the older tuple between the arrays causes DBB 2 to be selected, matching the newest dependence in the exact graph. DBB 5’s query illustrates shadowing; the writes in DBBs 2 and 3 obscure the true dependence source. Read aliasing due to the inclusion of the (unmatched) load from A causes 3 to be returned, rather than 2 if the load from D was alone. Likewise, the writes in DBB 6 cause DBB 7’s query to find an edge from 6, even though the exact graph has no edge to 7.

Figure 7(d) shows the resulting maximally-hoisted dynamic dependence graph; again, it shares nodes 1, 3, and 6 with the exact graph but also includes 7, increasing the critical path length to 8.

**Result Choice Example.** Figure 8(a) demonstrates the three possible situations in choosing a result for a single read. DBBs 1 and 2 show the simplest case: if no intervening writes have overwritten the relevant tuples, any choice is acceptable.

If only some of the fields have been overwritten, the correct dependence source may still be found, as in DBBs 3 and 4. Since writes executed after the correct one will have newer timestamps, we may simply select the tuple with the oldest timestamp to find the correct dependence source.

If all the relevant tuples have been overwritten, we must return an imprecise dependence source. Returning any of the tuples will give us a dependence whose source DBB executed after the correct DBB, but the oldest tuple is closest to the correct tuple. In Figure 8(a), DBB 3 is a better choice than DBB 5 as a source for the dependence in DBB 6, since DBB 3 is closer to DBB 1, the correct source.

Figure 8(b) continues the example with multiple reads in each DBB. In DBB 7, it’s easy to see that we should choose DBB 5 as the dependence source, not DBB 3; choosing the newer dependence inside each array gives the right result. In DBB 9, by choosing the newest tuple in each array and the oldest tuple between arrays, we still get the correct source of DBB 5.

**Bloom Map Hardware.** Figure 9 shows a simple Bloom Map design that processes reads and writes individually. No associative structure or content-addressable memory is required for a Bloom Map; for both reads and writes, the arrays are indexed directly by fields of the address. Standard memory arrays can be used.

For each write address, we must store the DBID in the right location in each memory array. We do not want to compare the reads and writes of the same basic block, so we must insert a dynamic basic block’s writes after processing its reads. A conceptually simple way to implement this is to buffer the writes until the end of the dynamic basic block. Then, for each write, we use the fields of the write address to index into the arrays, storing the DBID tuple at each location.

Processing reads is similar. To find the correct tuple in each of the memory arrays, we use the appropriate field of the read address to index into the array. Once each array has chosen the relevant tuple, a final result must be chosen. We use a reduction tree made up of comparator-mux pairs. The comparator examines only the timestamp of the DBID. The output of the comparator drives the mux to pass along the correct tuple: the one with the oldest timestamp. If all arrays contained a valid tuple, the final result is valid.

Once the DBID tuple with the oldest timestamp ends up at the root of the tree, we have the dependence source: we store its DBID together with that of the currently-executing basic block (the destination) in the dependence edge buffer.

**4.3. Discussion**

The Bloom History and Bloom Map store data in different ways to capture different parts of the graph. The Bloom History stores write sets for each DBB; storage is allocated for each DBB whether or not it writes to an address. The Bloom Map stores potential dependence endpoints; storage is allocated for
Figure 7. A sequence of executed instructions and their data dependences, split into dynamic basic blocks (a) and observed by a Bloom Map (b) with hash mapping shown in (c). The resulting precise and imprecise maximally-hoisted dependence graphs are shown with the blocks' instruction counts in (d).

(and shared between) addresses that are written. The Bloom Map stores the most recent dependence source for an address, while the Bloom History stores all dependence sources for an address. For short critical dependences, the Bloom History wastes space storing dependences sources that don’t matter. An analogy can be drawn to data encoding: the Bloom History stores a DBB’s critical dependence in one-hot fashion, whereas the Bloom Map stores a DBB’s critical dependence in a binary index encoding form. This suggests that a Bloom Map is likely to be more efficient in encoding dependences.

The Bloom History and Bloom Map each exhibit different types of imprecision in their captured dynamic dependence graphs. The Bloom History’s imprecision within a window is limited to adding edges to the exact graph. Any edge in the exact dependence graph will be in the Bloom History’s graph, but the latter may contain additional edges. Hoisting based on this graph will never lead to a critical path shorter than that in the exact graph: the Bloom History never over-estimates available parallelism. The Bloom Map, on the other hand, emits only one dependence per DBB, and it will always emit a dependence no longer than the equivalent dependence found by an exact detector. Since we select the dependence with the oldest timestamp, and since timestamps always increase, any imprecision could only result in an edge from a DBB executed after the correct one. But this imprecision may have consequences when computing available parallelism: shadowing may obscure a dependence that is part of the exact graph’s critical path, leading the Bloom Map to find a shorter critical path and thus over-estimate concurrency. Imprecision may also lead the Bloom Map to find a longer critical path and under-estimate concurrency, just as with the Bloom History.

The Bloom History and Bloom Map both have limitations on the length of dependences they can collect. The Bloom History is limited by the window size, chosen at design time. The Bloom Map is limited by aliasing, and while the configuration of the Bloom Map’s fields plays a role, this depends mainly on the series of addresses being stored.

Both the Bloom History and Bloom Map require comparison logic to identify dependences. The number of comparisons required for a Bloom History depends on the chosen window size. The number of comparisons required for a Bloom Map is independent of window size.
Figure 8. Examples for Bloom Map tuple selection. Only the timestamps of DBBs are shown. Cells show contents after the code for that DBB has executed. Arrows indicate which cells will be considered when computing the dependence for the reads in that DBB.

5. Evaluation

5.1. Setup

To explore the performance of our Bloom structures, we built a simulator using Intel’s dynamic binary instrumentation tool Pin [8]. We used SPECint 2006 as our benchmark set, running the test workload with base tuning parameters. We ran our experiments on a 64-bit Mac Pro running Ubuntu Linux, as well as on Large and Extra Large nodes at Amazon’s EC2 cloud computing facility. For each experiment, each benchmark was executed once, and the same trace of execution was used by each candidate structure.

The Bloom History structures have high simulation cost, so we implemented a sampling trace collector for experiments involving these structures. We chose a sample window size as large as practical; at each window boundary, we flipped a weighted coin to decide whether or not to sample in that interval. Dependences were tracked only within each sample interval, and the critical path was formed by concatenating the critical paths from each sampled interval. Available parallelism was calculated using only the instruction count from the sampled regions.

Addresses were compared at word granularity. Dependences were sampled in windows of 200,000 dynamic basic blocks to ensure that each sample window covered more than one million instructions. The sampled windows covered one percent of the program’s execution trace. An exact, hashtable-based software dependence detector was included in the runs to provide a baseline for comparison. This detector observed the same trace as the Bloom structures and followed the same sampling rules.

We executed the SPECint 2006 benchmark set with a set of Bloom History and Bloom Map structures. We chose four Bloom Map configurations with bit budgets between 1Mbit (128KBytes) and 1Gbit along with Bloom History configurations for the larger three bit budgets, shown in Table 1. A small Bloom History could not be created, since even the 200,000 64-bit BBIDs required to cover our chosen simulation window would require 12Mbits of storage alone, exceeding the 1Mbit bit budget.

5.2. Results

We evaluate the performance of our mechanisms with two metrics. The first is edge error: our measure of how different the edges collected by our Bloom structures are from those in the exact dynamic dependence graph. Missing edges are included in this metric as well. This metric is defined as the ratio of edges not in both the precise and imprecise graphs (the symmetric difference) to total edges in both graphs, or in set-theoretic terms:

\[
\text{edge error} = \frac{|\text{imprecise edge set} \Delta \text{precise edge set}|}{|\text{imprecise edge set} \cup \text{precise edge set}|}.
\]

We measured this edge error over the whole collected graph, as well as the graph of all edges from intermediate critical paths computed as the program executed. This latter metric is intended to show that the Bloom structure not only finds the crit-
The Bloom Map is clearly a better choice based on our simulations. It is able to observe the entire trace of a program’s execution, potentially finding long, outer-loop concurrency. Since all writes are inserted into a single structure, aliasing is likely, but all the area can be allocated to this single structure, rather than spreading it across the window as the Bloom History does. Could the Bloom History ever be a better choice? It has the advantage that for exact dependences that fit within a sample window, it will always find those dependences, avoiding the over-estimation of concurrency (rarely) allowed by the Bloom
Map. But given the Bloom History’s significantly higher complexity and area and lower accuracy per bit, a Bloom Map of equivalent size will probably provide better performance.

7. Conclusion

We have shown that allowing imprecision in dependence collection supports capturing important dependences with a small data structure. We described two efficient hardware structures for dependence collection that allow restricted classes of errors as a tradeoff for saving space and lowering collection overhead. Even with these errors, the structures are able to find similar opportunities for parallelization as a resource-hungry exact collector.

Acknowledgments

We thank Amazon.com for their donation of EC2 time. The anonymous reviewers and members of the WASP and Sampa groups at the University of Washington provided invaluable feedback in preparing this manuscript. Special thanks go to Doug Burger from Microsoft Research and Aaron Kimball from Cloudera for very helpful discussions in developing the ideas and the text. This work was supported in part by NSF under grant CCF-0702225 and a gift from Intel.

References


**Figure 10.** Overall edge error incurred by Small, Medium, Large, and X-Large Bloom Map configurations (SM, MM, LM, XM), and Medium, Large, and X-Large Bloom History configurations (MH, LH, XH) observing each of the SPECint benchmarks, as well as the harmonic mean of all the SPECint results. Good configurations have low edge error.

**Figure 11.** Edge error on intermediate critical paths incurred by Small, Medium, Large, and X-Large Bloom Map configurations (SM, MM, LM, XM), and Medium, Large, and X-Large Bloom History configurations (MH, LH, XH) observing each of the SPECint benchmarks, as well as the harmonic mean of all the SPECint results. Good configurations have low critical path edge error.

**Figure 12.** Available parallelism discovered in the SPECint benchmarks by Small, Medium, Large, and X-Large Bloom Map configurations (SM, MM, LM, XM), and Medium, Large, and X-Large Bloom History configurations (MH, LH, XH), as well as the harmonic mean of all the SPECint results, relative to available parallelism discovered by an exact analyzer (E). Good configurations match the exact analyzer.
Parallelization Spectroscopy: Analysis of Thread-level Parallelism in HPC Programs

Arun Kejariwal
University of California, Irvine

Călin Cașcaval
IBM T.J. Watson Research Center

Abstract

In this paper, we present a thorough analysis of thread-level parallelism available in production high performance computing (HPC) codes. We survey a number of techniques that are commonly used for parallelization and classify all the loops in the applications studied using a sensitivity metric: how likely is a particular technique is successful in parallelizing the loop. We call this method parallelization spectroscopy. Using parallelization spectroscopy, we show that in most of the benchmarks, at the loop level, more than >75% percent of the runtime is inherently parallel.

1. Introduction

The need for high performance systems coupled with power constraints has driven the development of both homogeneous and heterogeneous multi-core chips. Examples include Intel’s Nehalem and Sandy Bridge, IBM’s Cell and POWER processors, and Sun’s UltraSPARC T* family. Such systems require large scale (thread-level) parallel program execution, wherein the threads are mapped onto different physical processors. However, in practice, efficient exploitation of thread-level parallelism (TLP) is non-trivial. This, in part, is due to the lack of abstractions for expressing parallelism at the programming language level, lack of easy-to-use parallel programming models, limitations of compiler-driven program parallelization and many other practical limitations such as the threading overhead, destructive cache interference between the threads and non-graceful scaling of resources such as the memory bus bandwidth. Recently, there has been a large body of work addressing these issues as discussed below:

Software: There has been an increasing impetus in the development of new programming languages to efficiently capture the inherent parallelism early in the software development cycle. Examples include IBM’s X10 [59], Sun’s Fortress [15] and Cray’s Chapel [10] languages. On the other hand, new programming models such as OpenMP [41] and PGAS [52] are being developed or extended to ease parallel programming. Further, parallel data structures [30] and libraries [6] are being developed to assist the component-based software development, a key to high achieving high productivity.

Hardware: As shown in [31], applications from recognition, mining, and synthesis (RMS) domains have small (parallel) tasks thereby limiting the speedup achievable via multithreading in software. This also requires architectural support for exploiting fine-grain TLP. Additionally, thread-level speculation (TLS) [49] and transactional memory (TM) [22] have been proposed as a means to extract optimistic concurrency from potentially parallel program regions.

Workload characterization (Figure 1) plays a critical role in guiding research and development of both software and hardware [7]. This stems from the fact that the introduction of any new idea into mainstream software or hardware is highly dependent on its applicability to existing and emerging workloads. In fact, based on workload characterization, there has been an increasing emphasis on the design of (i) new partitioned global address space (PGAS) languages such as UPC, X10 and Chapel, (ii) new domain-specific programming languages such as MATLAB for scientific computing, parallel version of SQL for database applications [23] and (iii) architectures [48, 3]. For this purpose, we present a detailed and practical analysis of the available TLP in production HPC codes. Specifically, we determine the coverage, defined as the percentage of the total execution time, of inherently parallel program regions such as parallel loops (or D0ALL loops [36]). Additionally, we highlight the granularity, of the available parallelism, and we detail the factors inhibiting parallelization. We refer to this analysis as parallelization spectroscopy.

The main contributions of this paper are:

- A thorough characterization of the task level parallelism for loops in a large number of HPC workloads that are characteristic of production level codes;
- A synopsis of techniques used for parallelization; an integrated framework for workload characterization with respect to parallel execution, the parallelization spectroscopy, that was used for our characterization work;
- A realistic estimation of the speculation potential for scientific workloads; more than 75% of the execution time in these benchmarks is inherently parallel. To attain this parallelism we need enhanced compiler support or user annotations, but not necessarily speculation support. We also conclude that good parallel library development is critical for the performance of scientific codes;
- A number of tools [55, 58] use techniques such as the parallelization spectroscopy to guide the selection of loops. In that context, some of the manual analyses presented in this paper have been...
automated, such as dependence profiling, reduction recognition, and profitability measures. Most of the other analysis techniques presented here can also be automated. However, the focus of this paper is the detailed characterization of the HPC workloads, rather than the description of the tool.

The rest of the paper is organized as follows: Section 2 walks through the various facets of parallelization spectroscopy. Section 3 briefly the benchmarks used in this work. Experimental setup is described Section 4. Evaluation of the available thread-level parallelism in the benchmarks is presented in Section 5. Related work is discussed in Section 6. Finally, in Section 7 we conclude with directions for future work.

2. Parallelization Spectroscopy

In this section, we introduce the different dimensions of our spectroscopic analysis of loop-level parallelization of HPC codes.

First, we survey the set of techniques required for parallelizing loops in HPC codes, and we log the frequency of applicability of such techniques. We define a metric, denoted as $S(L, T)$, to measure the parallelization sensitivity of an application $P$ with respect to a technique $T$ (such as scalar privatization).

$$S(L, T) = \frac{\text{Number of loops to which $T$ is applied}}{|L|}$$

where $L$ is the set of all the loops in $P$. Note that $S(L, T) \in [0, 1]$. A high value of $S(L, T)$ signifies that $T$ is required for the parallelization of a large number of loops in $P$. However, $S(L, T)$ does not quantify the performance gain achievable based on the loops parallelized using $T$. Thus, we define the following metric:

$$S_{cov}(L, T) = \frac{\sum_{L_i \in L} \text{cov}(L_i)}{\sum_{L_i \in L} \text{cov}(L_i)}$$

where $L_{T}$ is the set of loops ($\subseteq L$) parallelized via application of $T$ and $\text{cov}(L_i)$ denotes the coverage of loop $L_i$. A higher value of $S_{cov}(L, T)$ signifies that loop parallelization based on $T$ has a large performance gain potential. Note that $S_{cov}(L, T) \in [0, 1]$. For many loops, as evidenced by the analysis presented in Section 5, more than one technique may be required for parallelization. In such cases, we define the following relational metric:

$$\text{RelS}_{cov}(L, T, T_j) = \frac{\sum_{L_i \in L_{(T_j)_k}} \text{cov}(L_i)}{\sum_{L_i \in L_{(T_j)_k}} \text{cov}(L_i)}$$

where $T_j \neq T_k \land T_j, T_k \in T$, $T$ is the set of all techniques and $L_{(T_j)_k}$ is the set of loops ($\subseteq L$) parallelized via application of both $T_j$ and $T_k$. Note that $\text{RelS}_{cov}(L, T, T_j) \in [0, 1]$.

In practice, the achieved speedup is subject to a multitude of factors such as (but not limited to) the underlying architecture and threading overhead. The $\text{RelS}_{cov}$ metric provides a valuable guidance to programmers and compiler writers to select transformations sequences that provide maximum performance impact. Note that the order of the transformations may affect the performance as well. In this paper we do not consider the order in which the transformations were applied.

Second, we identify and characterize the bottlenecks such as I/O, which inhibit loop parallelization. These bottlenecks must be removed by user intervention, re-coding the application to use parallel I/O operations, as there are no automatic techniques to handle parallel I/O. And third, we assess amount of nested TLP in the HPC codes. For outer loops (in a given loop nest) with small number of iterations, it is critical to exploit, wherever possible, nested TLP. This is particularly important in light of the increasing number of hardware contexts in the emerging multi-core systems.

The following lists the techniques or transformations we considered for spectroscopy analysis:

- **Reduction**: It exploits commutative property of a computation such as accumulation, to drive loop parallelization. As shown in Section 5, reduction is widely applicable in parallelization of HPC codes;
- **Scalar/Array Privatization**: It is used to eliminate a loop-carried dependence by instantiating a local copy of the source of the loop-carried dependence in each iteration of the loop [34, 51]. Akin to reduction, privatization is also widely applicable in parallelization of HPC codes;
- **Loop Transformations**: A large variety of loop transformations such as (but not limited to) loop permutation have been proposed for (or to assist) loop parallelization[4]. We shall discuss their efficacy case by case for our workloads;
- **Symbolic Analysis**: Loop-carried dependences can be eliminated via symbolic analysis [20]. We assess the applicability of this technique;
- **Call-site Analysis**: In real codes, it is not uncommon that a hot loop may not be intrinsically amenable for parallelization. In such case, it imperative to explore parallelizability at higher levels of abstraction. For this, we carried demand-driven multi-level call-site analysis of the function(s) containing the hot loop(s). Specifically, we traverse the call graph and identify whether the call site of such function(s) belongs to a parallel program region.

3. Benchmark Selection

Several limit studies have assessed the amount of inherent thread-level/task-level parallelism [42, 29, 28, 27, 26]. These studies were primarily based on the SPEC CPU benchmarks [50]. In contrast, we focus on a set of larger applications selected from multiple industry-strength suites such as the Sequoia benchmark suite [45] from LLNL, publicly available codes such as CPMD [12] and production codes used in the DARPA HPCS program. The evaluation of available TLP presented in this paper complements prior work by shedding light on a wider set of applications.

Table 1 lists the benchmarks, their size in terms of number of lines of code, programming language and parallelization support in the original source code.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Lines of Code</th>
<th>Language</th>
<th>Parallelization Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMG</td>
<td>108169</td>
<td>C</td>
<td>MPI</td>
</tr>
<tr>
<td>CrystalMk</td>
<td>468</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>IRSmk</td>
<td>457</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>CPMD</td>
<td>194823</td>
<td>Fortran</td>
<td>OpenMP, MPI</td>
</tr>
<tr>
<td>POP</td>
<td>65654</td>
<td>Fortran90</td>
<td>OpenMP, MPI</td>
</tr>
<tr>
<td>UMT2K</td>
<td>19931</td>
<td>Fortran/C</td>
<td>OpenMP, MPI</td>
</tr>
<tr>
<td>RF-CTH</td>
<td>534382</td>
<td>Fortran/C</td>
<td>MPI</td>
</tr>
<tr>
<td>SPPM</td>
<td>20957</td>
<td>Fortran</td>
<td>OpenMP or MPI</td>
</tr>
<tr>
<td>HYCOM</td>
<td>32187</td>
<td>Fortran</td>
<td>OpenMP or MPI</td>
</tr>
<tr>
<td>Sweep3d</td>
<td>1952</td>
<td>Fortran</td>
<td>MPI</td>
</tr>
</tbody>
</table>

Table 1. Overview of the benchmarks

4. Experimental Setup

The analysis presented in Section 5 is empirical. In order to alleviate the artifacts of one system, we performed the experiments on two different systems, whose detailed configuration is given in Table 2. We compiled the applications listed in Table 1 using the IBM XLC v9/XLF v10 compiler (for the Power system) and gfortan/gcc 4.1.2 [17] (for the X86 system). We used the -pg option along with
other options and then used gprof [18] to obtain the function-level coverage profiles. For reproducibility of the results presented in this paper, the application specific compiler optimization flags and the run time commands used are reported in the respective subsections in Section 5. For applications parallelized using MPI and/or OpenMP directives, the results are presented for one MPI task or a single OpenMP thread, unless stated otherwise.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Intel Xeon, 2.8 GHz</th>
<th>POWER5, 1.6 GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>512 MB</td>
<td>3.8 GB</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>8 KB</td>
<td>64 KB</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>512 KB</td>
<td>32 MB</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>None</td>
<td>32 MB</td>
</tr>
<tr>
<td>OS</td>
<td>Linux 2.6.9 Fedora Core 3</td>
<td>AIX 5.3</td>
</tr>
</tbody>
</table>

Table 2. Experimental Setup

In order to be consistent with methodology employed in previous limit studies, based on the SPEC CPU benchmarks, we did not modify the source code of any application before compilation.

5. Parallelism Evaluation

In this section we present a detailed evaluation of available parallelism in production codes listed in Table 1. For this, we employed the following approach:

a) First, for each application, we breakdown the function-level coverage profile into three categories: (1) inherently parallel (IP) program regions; (2) potentially parallel (PP) program regions and (3) “mostly” serial (MS) program regions. The coverage of IP serves as an upper bound on the speedup achievable via conventional multithreaded execution. The coverage of PP serves as an upper bound on the speedup achievable via multithreaded execution with support for explicit inter-thread synchronization and/or optimistic parallelization, such as TLS. In practice, the performance gain achievable is subject to a wide variety of factors such as cache interference between the different threads and threading overhead. A detailed analysis of these factors requires a precise machine model and is beyond the scope of this paper. In the two profiles is due to the difference in heuristics and phase ordering employed by the IBM XL and gcc compiler. The difference in the total number of functions in the two profiles is due to the difference in heuristics and phase ordering employed by the IBM XL and gcc compiler. The compiler options used while compiling AMG were:

```bash
mpirun -np 1 amg2006 -r 6 6 6 -printstats
```

We analyzed the loops in the top 5 hot functions to assess the inherent TLP in AMG. For instance, the loops in the hottest function `hypre_BoomerAMGRelax` (coverage of 20.8%) have the code as of

```
for (i = 0; i < n; i++) {
  if (A_diag_data[A_diag_i[i]] != zero) {
    res = f_data[i];
    for (jj = A_diag_i[i]+1; jj < A_diag_i[i+1]; jj++) {
      ii = A_diag_j[jj];
      res -= A_diag_data[jj] * Vtemp_data[ii];
    }
    for (jj = A_offd_i[i]; jj < A_offd_i[i+1]; jj++) {
      ii = A_offd_j[jj];
      res -= A_offd_data[jj] * Vext_data[ii];
    }
    u_data[i] *= one_minus_weight;
    u_data[i] += relax_weight * res / A_diag_data[A_diag_i[i]];
  }
}
```

5.1 AMG

AMG is an algebraic multigrid solver for linear systems arising from problems on unstructured grids [44]. The driver provides builds linear systems for various 3D problems. The code is written in ISO standard C. The purpose of the benchmark, as mentioned by LLNL, is to test single CPU performance and scaling efficiency. AMG is part of the Sequoia benchmark suite [45] from LLNL.

The compiler options used while compiling AMG were:

```bash
-02 -DTIMER_USE_MPI -DHYPRE_NO_GLOBAL_PARTITION
```

Subsequently, we ran the binary with the following (default) options:

```bash
mpirun -np 1 amg2006 -r 6 6 6 -printstats
```

The function-level coverage profile of AMG on POWER5 and Xeon is shown in Figure 2. The total number of functions – the range of the x-axis in each profile – reported for each application correspond to the dynamically executed functions. This need not be equal to the number of functions in the source code. The latter can be, in part, ascribed to the following: (a) a function may not be executed for a given input data set and (b) functions may be inlined the compiler. The difference in the total number of functions in the two profiles is due to the difference in heuristics and phase ordering employed by the IBM XL and gcc compiler. For instance, the IBM XL and gfortran/gcc compilers employ different function inlining heuristics. This directly affects the total number of dynamically executed functions. Note that the two profiles shown in Figure 2 are very similar. This trend holds for all the applications we studied, which serves as an empirical evidence the results and analysis presented in this paper are not an artifact of the underlying architecture or a particular compiler. Due to space limitations, we are unable to include the coverage profile of all the applications.

```c
for (i = 0; i < n; i++) {
  if (A_diag_data[A_diag_i[i]] != zero) {
    res = f_data[i];
    for (jj = A_diag_i[i]+1; jj < A_diag_i[i+1]; jj++) {
      ii = A_diag_j[jj];
      res -= A_diag_data[jj] * Vtemp_data[ii];
    }
    for (jj = A_offd_i[i]; jj < A_offd_i[i+1]; jj++) {
      ii = A_offd_j[jj];
      res -= A_offd_data[jj] * Vext_data[ii];
    }
    u_data[i] *= one_minus_weight;
    u_data[i] += relax_weight * res / A_diag_data[A_diag_i[i]];
  }
}
```
the loop shown above (taken from file hypre_BoomerAMG Relax\_c, line number 188).

From the snippet we note that the loop is a DOALL loop, subject to privatization of variables such as ii, jj and res. The presence of a subscripted subscript (the read from the array A\_diag\_data) and/or a conditional does not necessarily inhibit the parallelization of the loop! Likewise, the hot loops in the function hypre_BoomerAMG BuildCoarseOperator (coverage of 18.4\%), at line numbers 541, 748, 1121 and 1393 in the file par\_rap\_c are DOALLs. On further analysis we note that the hot loops in other functions such as hypre_BoomerAMG BuildInterp (coverage of 18.1\%) and hypre_CSRMatrix Matvec (coverage of 11.5\%) are also DOALLs. Overall, more than 75\% of the total coverage belongs to the IP category.

Non-DOALL loops in AMG have a conditional dependence between the different iterations. For example, let us consider the following loop (taken from the file par\_interp\_c, line number 236):

```
for (i=0; i < num\_cols\_A\_offd; i++) {
    for (j=A\_ext\_i[i]; j < A\_ext\_i[i+1]; j++) {
        k = A\_ext\_j[j];
        if (k > col_1 && k < col_n) {
            A\_ext\_data[index++] = A\_ext\_data[j];
            A\_ext\_j[index] = k - col_1;
            A\_ext\_data[index++] = A\_ext\_data[j];
        } else {
            kc = hypre\_BinarySearch(col\_map\_offd,k,num\_cols\_A\_offd);
            if (kc > -1) {
                A\_ext\_j[index] = kc-1;
                A\_ext\_data[index++] = A\_ext\_data[j];
            } else {
                A\_ext\_j[index] = index;
            }
        } 
    }
}
```

From the code snippet, we observe that the conditional increment of the variable index may induce a dependence between the (respective) writes to the array’s A\_ext\_j and A\_ext\_data in different iterations of the loop. Further, the variable index is neither an induction variable nor can it be privatized. Due to this, the loop is classified as a non-DOALL loop. However, the upper bound of the aforementioned run time option is zero! Assuming that the default options are representative of the general case, we argue that such non-DOALL loops do not impact the parallel performance significantly.

5.2 Crystalmk

Crystalmk is a single CPU, C program intended to be an optimization and SIMD compiler challenge [46] and is part of the Sequoia benchmark suite [45] from LLNL. It consists of selected small portions of a large material strength package; however, the performance of this very set dominates the performance of the full package.

Based on our analysis of the function-level coverage profile of Crystalmk we note that the function Crystal\_Cholesky accounts for the largest coverage – 37.17\% on the POWER5 – of all the functions. Let us consider the main loop of the function Crystal\_Cholesky, taken from file Crystal\_Cholesky\_c, line number 33.

```
for (i = 0; i < nSlip; i++) {
    for (k=0; k<i; k++)
        fdot += a[i][k] * a[k][i];
    a[i][i] = a[i][i] - fdot;
    for (k=i+1; k<nSlip; k++)
        for (j=i; j<k; j++)
            a[i][j] += a[i][k] * a[k][j];
    for (j=i+1; j<nSlip; j++)
        fdot = 0.0;
    for (k=0; k<i; k++)
        for (j=i+1; j<nSlip; j++)
            a[i][j] += a[i][k] * a[k][j];
    a[i][i] = a[i][i] - fdot;
}
```

From the code snippet we note that the outer loop L1 is a non-DOALL loop. For example, the array element a[2][1] is written to in iteration 1 and is then read in iteration 2 of the loop L1. However, the inner loops L2 and L3 are DOALLs. Loop L2 can be parallelized using OpenMP-type reduction of the variable fdot, whereas loop L3 can be parallelized via privatization of the variable fdot. The key to parallelization of loop L3 is the exploitation of the condition k < i in the header of the loops L4 and L5. This guarantees that there is no dependence between the iterations of the loop L3 (see Figure 3 for the memory access pattern).

![Figure 3. Memory access pattern for iteration i of loop L3](image)

In Figure 3, the matrix represents the array a. The shaded blocks correspond to the elements of a written in the first iteration of the loop L3. Let us consider the blue colored block which corresponds to a[1][3]. The blue arrows represent the set of elements of the array a read for computing fdot which is then subtracted from a[1][3]. As j increases, the dashed arrow – which corresponds to a[k][3] in loop L4 – shifts to the right, whereas the solid arrow – which corresponds to a[1][k] in Loop L4 – remains “fixed”. Similarly, in the case of the red colored block which corresponds to a[3][1], the dashed arrow moves downwards and the solid arrow remains “fixed”. On analysis, we note the read and writes in loops L4 and L5 do not introduce a loop-carried dependence between the iterations of loop L3.

Based on run-time analysis of the loop above, we find that the value of the variable nSlip is 12. Given this, the number of iterations of loop L3, on an average, is 3.5. Hence, we argue that if the number of processors is less than 6, then it is more profitable to exploit TLP at the level of loop L3. Our recommendation is based on the fact that the L3 is a DOALL loop. If the number of processors is more than 5 then TLP and speculative thread-level parallelism may be exploited at the levels of loops L3 and L1 respectively.

On the other hand, the hot loop in the function Crystal\_div (which has a coverage of 21.1\% on POWER5) is a DOALL loop (the loop is shown below). Interestingly, the library function pow accounts for a coverage of 13.3\% on the POWER5! This suggests that lack of parallelization of such library routines would limit the performance gain achievable via parallelization of only the source code.

```
for (n = 0; n < nSlip; n++)
    for (k = 0; k < nSlip; k++)
        for (m = 0; m < nSlip; m++)
            for (l = 0; l < nSlip; l++)
                matrix[n][m] = (matrix[n][m] + matrix[k][m]) * bor\_array[n];
```

5.3 IRSmk

IRSmk [47] is part of the Sequoia benchmark suite [45] from LLNL. The purpose of the benchmark, as stated by LLNL, is to assess the scaling efficiency and single processor performance.
assessment. We compiled the benchmark using the gcc-4.1.2 compiler and with following options: -c -O3 -pg.

The benchmark code comes along with three input data sets viz., irsmk-input_25, irsmk-input_25 and irsmk-input_100. We used all the three input data sets for our experiments. The binary was executed using the ./IRSmk command. From the figure we observe that the first function (rmatmult3) accounts for over 99% of the total execution time. The only loop in the function mentioned above, taken from file rmatmult3.c, line number 79 (shown below), has a coverage of 99% on both the systems.

```c
L1: do 4010,ib=1,before
b[i] = dbl[i] * xdbl[i] + dbc[i] * xdbc[i] + dbr[i] * xdbr[i] +
                 ds[i] * xds[i] + dts[i] * xts[i] + dtr[i] * xtr[i] +
                 ds[i] * xds[i] + dts[i] * xts[i] + dtr[i] * xtr[i] +
                 ds[i] * xds[i] + dts[i] * xts[i] + dtr[i] * xtr[i] +
                 ds[i] * xds[i] + dts[i] * xts[i] + dtr[i] * xtr[i] +
                 ds[i] * xds[i] + dts[i] * xts[i] + dtr[i] * xtr[i] +
                 ds[i] * xds[i] + dts[i] * xts[i] + dtr[i] * xtr[i] +
     i=i i+j j*j p+k k*k p;
L2: do 4010,ib=1,before
nin4=ib-after
nout4=nout3+after
nout3=nout2+after
nout2=nout1+after
nout1=nout1+atn
nin4=nin3+atb
nin3=nin2+atb
nin2=nin1+atb
nin1=nin1+after
4010 continue
else
zout(2,j,nout2) = r + s
zout(2,j,nout3) = r - s
zout(2,j,nout1) = r + s
...) zout(2,j,nout4) = r - s
4000 continue
endif
```

On analyzing the code snippet we note that the outer loop is a DOALL loop, subject to privatization of variables such as kk, jj, ii and i. Thus, based on code analysis, we classify IRSmk under the IP category.

Based on run-time analysis, we find that the iteration count of each loop in the triply nested loop is 100. Table 3 reports our recommended loop nesting level for multithreading based on the number of processors. Arguably, loop L3 can also be multithreaded if the number of processors is > 10^4. However, this may not be profitable due to small amount of computation in the loop body.

<table>
<thead>
<tr>
<th># of processors</th>
<th>Loop-level</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;100</td>
<td>L1 only</td>
</tr>
<tr>
<td>100 &lt; &amp; ≤ 10^4</td>
<td>L1 and L2</td>
</tr>
</tbody>
</table>

Table 3. Granularity of multithreading for IRSmk

5.4 CPMD

The CPMD code is a plane wave/pseudopotential implementation of Density Functional Theory, particularly designed for ab-initio molecular dynamics (MD) [12]. We compiled the source code using the gfortran-4.1.2 compiler with the following options:

-02 -fcray-pointer -fsecond underscore -pg

and using the following libraries
-llamf77mpi -llmpi -llam -lpthread -llblas -llapack

The CPMD code internally invokes routines from the LAPACK package [33]. We used version 3.1.1. We ran the binary on a Xeon-based 4 processor system. We used both the input data sets provided with the distribution and used the LAM/MPI [32] for running the binary on the 4-processor system.

On analyzing the function-level coverage profiles we note that the coverage of the hottest function is 31.28% and 41.91% for input data sets 1 and 2 respectively. The loops in the hottest subroutine fftstp are of the type shown on the right hand side gfft.f, line number 70).

On analyzing the code snippet we note that loops L3 and L5 are DOALLs, subject to privatization of variables such as x and a. Likewise, loops L2 and L4 are also DOALLs, subject to IVE (induction variable elimination) [38] and atn > 3× after. The latter stems from the following:

1. nout4 = nout1 + 3× after (1)
2. nout3 = nout1 + 2× after (2)
3. nout2 = nout1 + after (3)

2. The variable nout1 increases monotonically with increasing value of ib.

From above and the fact that nout1 is incremented by atn in each iteration, we conclude that the writes to the array zout do not alias if atn > 3× after. Next, let us analyze the outermost loop L1. We observe that along the then as well as the else branch of the conditional:

- The initialization of nout1: nout1=ia-atan, is the same.
- The variables nout1, nout2, nout3 and nout4 are computed in the same fashion.
- The lower and upper bounds of loops L2 and L3 are the same as that of the loops L4 and L5.

In light of the above, the condition for no aliasing of writes to the array zout from L1 perspective is the same as that for loops L2 and L4: atn > 3× after. This condition can be used as a basis for loop versioning whereby a parallelized version of the outermost loop can be invoked at run-time subject to the satisfiability of the above condition.

The inability of a compiler to determine the satisfiability of the condition mentioned above may suggest to include the coverage of the outermost loop, minus the coverage of inner parallel loops, under the PFP category. However, on further program analysis, we observe that the function fftstp is called in the parallel loop at line 53 in the file mlfft.f. The loop is parallelized using OpenMP pragmas in the original source code. Therefore, the coverage of the function fftstp should be counted as part of the coverage of parallel regions of CPMD. Similarly, the functions fftpre and fftrot, which have coverages of 13.81% and 7.83%, are called from the above loop. Overall, our analysis shows that more than
75% of the total coverage is inherently parallel. Further, the upper bound of the loop at line 53 in the file mlitff.t.f is equal to the number of processors (NCUPS). Thus, in the current case, nested multithreading is unwarranted.

Lastly, the function zazzzero (coverage of 12.45%) consists of a loop wherein the elements of an array are set to zero. Thus, parallelization of the library routine memset can help in obtaining better performance.

5.5 POP

POP is an ocean modeling code, written in Fortran90, developed at Los Alamos National Lab. Prior to building the binary, we installed LAM/MPI [32], version 7.1.4 and the netcdf library [40], version 3.6.2. Then, we compiled POP using the IBM xlf90 compiler with -O3 -qsave -qmaxmem=131072 -q64 -qsuffix=f=f90 -qfree=f90 options and used mpi77 for linking. We ran POP on a single processor using the ./pop command and using a real dataset.

On analyzing the function-level coverage profile we see that the maximum coverage of an individual function – the function state in the module state_mod – is 30.28%. Unlike the hot functions in the benchmarks analyzed so far, state does not contain any loops! state consists of a 3-way and a 4-way select statements. Conceivably, the function can be parallelized via multipath execution [2] in conjunction with hardware/software support for termination of a wrongly executed path. Akin to the methodology followed for analysis of CPMD, we traced the calling context of state to explore TLP at higher level of abstraction.

<table>
<thead>
<tr>
<th>filename</th>
<th>line number</th>
<th>Caller function</th>
<th>Called inside a DOALL loop?</th>
</tr>
</thead>
<tbody>
<tr>
<td>advection.f90:1406,1413,1463,1470</td>
<td>advt</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>baroclinic.f90:574</td>
<td>baroclinic_correct_adjust</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>hmix.f90:926</td>
<td>hdiffu_aniso</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>hmix.f90:716,801,1663</td>
<td>hdiffu_gm</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>initial.f90:706,710</td>
<td>int js</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>step_mod.f90:1405,1409</td>
<td>step</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>vertical_mix.f90:1472,1474,1519</td>
<td>convad</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>vmodule_const.f90:213,218</td>
<td>vmodule CONST</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>vmodule_kjp.f90:1019,1825</td>
<td>bldgpath</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>vmodule_kjp.f90:1835,1977</td>
<td>dlmix</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Table 4. Calling context of the function state

Table 4 details the calling context of state and reports whether it is called inside a DOALL loop, parallelized in the original source code using OpenMP pragmas. For example, the call to state in initial.f90 is shown below:

```c
omp parallel do private(block, k, this_block) 
do block = 1,blocks_clinic
   this_block = get_block(blocks_clinic(block),block)
   do k=1,km
      call state(k,k,TRACD(:,1,curtime,block), &
      TRACD(:,2,curtime,block), &
      TRACD(:,3,curtime,block), &
      this_block, &
      hmix->hmix(1,1,1,curtime,block))
   enddo
enddo
```

From Table 4 we note that 5 (out of 26) contexts correspond to the IP category. Next, we traced second, third and so on levels of the calling context. For example, state is called by the function advt in advection.f90, advt is called by the function tracer_update. The latter is called inside a DOALL loop in the function baroclinic_driver. Thus, the first five calling contexts correspond to the IP category. Likewise, state is called by the function vmix_coeffs_const in vmix_const.f90, vmix_coeffs_const is called by the function vmix_coeffs which is in turn called inside a DOALL loop in the function baroclinic_driver. Based on the above analysis, we find that 22 contexts correspond to the IP category.

The second hottest function hdiffu_aniso (coverage of 13.09%) in the module hmix_aniso consists of mostly DOALL loops – at line numbers 691, 718 and 922 in the file hmix_aniso.f90. Similarly, most of the loops in the function hdiffu_gm (coverage of 12.19%) in the module hmix_gm are DOALLs, e.g., the outermost loop at line number 1232 in the file hmix_gm.f90 (the code of the loop – 194 lines – could not be included owing to space limitations) is a DOALL loop. Based on analyzing the top 10 hot functions, we conclude that more than 75% of the total coverage of POP is inherently parallel.

Lastly, the code makes extensive use of Fortran90 intrinsics as shown above (taken from the file hmix_gm.f90 line number 1137). The above computation involves matrix-scalar and matrix-matrix multiplication. Each intrinsic is “unfolded” into nested loops by the front end. The resulting loops can be executed in parallel as there does not exist any dependence between them. The coverage corresponding to such intrinsics corresponds to the PP category.

5.6 UMT2K

The UMT benchmark is a 3D, deterministic, multigrid, photon transport code for unstructured meshes. UMT 1.2, referred to as UMT2K for clarity, includes features that are commonly found in large LLNL applications.

We compiled UMT2K using the IBM xlf90 and xlc compilers with the -O3 -qsave -pg options. For linking, we used mpi77 and mpicc (of the LAM/MPI distribution). Then, we ran the binary on a POWER5-based system using the following command: ./umt2k -procs 1.

On analyzing the function-level coverage profile we note that the function (snawp3d) accounts for >90% of the total coverage. The function consists of 9 outermost loops (at line numbers 221, 226, 235, 275, 299, 306, 316, 356 and 553). On analysis we note that the loops at lines 235, 316 and 356 are not parallel and the rest are DOALLs. However, the inner loops in the three non-DOALL loops are DOALL loops.

For instance, the loop at line 373 in the file anawp3d.f is a DOALL loop and is inside the loop at line number 356. Likewise, the other inner loops inside this outermost loop are DOALL loops. Based on this, we ascribe the coverage of the outermost loop, minus the coverage of the inner DOALL loops, to the PP category.

As mentioned earlier, the profitability of speculative parallelization of non-DOALL outermost loops is subject to, say (but not limited to), the dependence properties such as the minimum dependence distance [5]. For the outermost loop at line number 356, the minimum dependence distance is very small. Consequently, exploitation of TLP at the inner loop level may be more profitable in the current context.

5.7 RF-CTH

CTH is a code used to explore the effects of strong shock waves on a variety of materials using many different models.
We compiled RF-CTH using the IBM xlf and xlc compilers and executed it on a POWER5 machine with small1 and small2 input data sets. The datasets were provided along with the source code distribution. The run is done in two steps: first, the input is processed using rfctgen and then the processed input is fed to the binary rfcth. Next, we present the evaluation of available TLP in the latter.

We analyzed the top 25 hot functions which account for 90% of the total coverage. Akin to other benchmarks, the inner loops in these functions are DOALLs. For illustration, the largest (in terms of lines of code) loop, subject to privatization of the scalar variables such as rpl1, rml1 and pl1. There is no loop-carried dependence based on the write to the arrays such as uavl, pavl and uxpavl. Beyond parallelization, higher speedup can be achieved by optimizing each iteration of the loop. For instance, the loops we analyzed, more 65% of the total coverage of erfays is inherently parallel.

The coverage reported above is an upper bound on the speedup achievable via vanilla multithreading. However, in practice, we find that it is not profitable to multithread many DOALL loops. This is due to their low coverage per invocation. In such cases, the performance gain achieved via multithreaded execution is offset by the threading overhead. For example, the function erfays has a coverage of 1.5% and is called 3755500 times. Given this and the configuration listed in Table 2, the run time of the function is approximately 8.7K cycles on an average. For simplicity, assuming uniform distribution of the run time between the different iterations of the loop, the run time each loop spans for < 800 cycles which makes multithreaded execution of such loops non-profitable. This highlights the need for exploiting TLP at higher levels akin to CPMD and POP.

5.8 SPPM

SPPM is a simplified version of PPM, the Piecewise-Parabolic method. It contains a nonlinear Riemann solver and a careful computation of the Courant time step limit. We compiled SPPM using the IBM xlf compiler with the -O3 -qarch=pwr4+nosave -qfixed=132 -qmaxmem=-1 -qautodbl=dbl4 -pg options and ran the binary on POWER5.

```fortran
DO 6030 NN=1,NXWB
   IF (IGM.GE.30) VZ(1,JJ)=PZERO
   VX(1,JJ)=PZERO
   IF (IDIOXB(IAMBLK).EQ.3) THEN
      VX(1,JJ)=-VX(3,JJ)
   ENDIF
   IF (IDIOXB(IAMBLK).EQ.1) THEN
      VX(1,JJ)=VX(2,JJ)
   ENDIF
   VX(2,JJ)=PZERO
   IF (IDIOXB(IAMBLK).EQ.0) THEN
      VX(1,JJ)=-VX(3,JJ)
   ENDIF
   VX(1,JJ)=-VX(3,JJ)
   VX(2,JJ)=PZERO
DO 6040 NN=1,NXWT
   IF (Y(JJ).GE.YSWIN(1,NN+10) .AND.
    & Z(KPLANE).LT.ZSWIN(2,NN)) THEN
      VX(1,JJ)=VX(2,JJ)
   ENDIF
   VX(1,JJ)=VX(2,JJ)
   VX(2,JJ)=PZERO
```

On analyzing the code snippet, we observe that there does not exist aliasing between the writes to the array VX in the different iterations. Also, the variable IFLG is local to each iteration. Thus, the coverage of this loop, minus the coverage of the inner parallel loops, is classified under the PP category.

All the loops in the functions convc (coverage of 11.8%) and elag (coverage of 7%) are of the type as the loop in the code snippet shown above and are DOALL loops. Overall, based on the loops we analyzed, more 65% of the total coverage of erfays is inherently parallel.

The coverage reported above is an upper bound on the speedup achievable via vanilla multithreading. However, in practice, we find that it is not profitable to multithread many DOALL loops. This is due to their low coverage per invocation. In such cases, the performance gain achieved via multithreaded execution is offset by the threading overhead. For example, the function erfays has a coverage of 1.5% and is called 3755500 times. Given this and the configuration listed in Table 2, the run time of the function is approximately 8.7K cycles on an average. For simplicity, assuming uniform distribution of the run time between the different iterations of the loop, the run time each loop spans for < 800 cycles which makes multithreaded execution of such loops non-profitable. This highlights the need for exploiting TLP at higher levels akin to CPMD and POP.

5.8 SPPM

SPPM is a simplified version of PPM, the Piecewise-Parabolic method. It contains a nonlinear Riemann solver and a careful computation of the Courant time step limit. We compiled SPPM using the IBM xlf compiler with the -O3 -qarch=pwr4+nosave -qfixed=132 -qmaxmem=-1 -qautodbl=dbl4 -pg options and ran the binary on POWER5.

```fortran
DO 6030 JJ=1,JMAX
   IF (ISHEM(LT 0) .THEN
      VX(JJ)=PZERO
   ELSEIF (ISHEM(LE 0) .THEN
      VX(JJ)=PZERO
   ENDIF
   ELSEIF (ISHEM(LT 1) .THEN
      VX(JJ)=PZERO
   ELSEIF (ISHEM(LE 1) .THEN
      VX(JJ)=PZERO
   ENDIF
   VX(1,JJ)=VX(2,JJ)
DO 6040 CONTINUE
```

We analyzed the top 6 hot functions of SPPM which account for a coverage of 94.19%. The hottest function sppm has a coverage of 30.61%. There are 12 outermost loops in this function and all of them are DOALLs. For illustration, the largest (in terms of lines of code) loop in the function sppm is shown above (taken from sppm.f, line number 703). On analyzing the code snippet, we note that the loop is a DOALL loop. Likewise, the multi-way loop [43] at line 132 in the file erfay.f, line number 1551.
writes to the array uxavl, statements 1 and 3, can be eliminated as the corresponding values are overwritten by the write to uxavl in statement 5. The writes to the array pavl in statements 2 and 4 can be memory-renamed [38], e.g., statement 2 can be rewritten as:

\[ \text{TEMP} = \text{max} (\text{smallp}, (\text{pavl}(i) - \text{thynq} * \text{dpuil})) \]

and then replace the reads to pavl(i) between statements 2 and 4 by the scalar TEMP. The write to pavl in statement 6 cannot be eliminated as the array pavl is not local to the loop.

The function difuze, at line 1135 in the file appm.f and with a coverage of 14.87%, consists of a singly nested DOALL loop. Likewise, the function interf, at line 1725 in the file appm.f and with a coverage of 13.37%, consists of a single nested DOALL loop. Overall, we note that more than 55% of the total coverage belongs to the IP category. On further analysis, we find that the intrinsics\(^3\) _vrec_GR and _vsqrt_GR account for a coverage of 16.72% and 13.37% respectively. This suggests that further parallelization of SPPM is subject to the parallelization of the library calls mentioned above.

5.9 HYCOM

HYCOM is a Hybrid Coordinate Ocean Model developed from MICOM (Miami Isopycnic Coordinate Ocean Model) and NLOM (Navy Layered Ocean Model) by a Consortium of LANL, NRL and University of Miami [24]. We compiled the source code using the IBM xlf compiler with the following options: -O3 -pg -qmaxdata:0x80000000/dsa -qstrict -qtune=pwr5 -qcache=auto -qPhillsize=92000 -q64 -qfixidx -qrealsize=8 -qintszie=4. Subsequently, We ran HYCOM, using the following command ./hycom.single on a POWERS5 processor.

We analyzed the top 15 functions which account for a coverage of 80%. Let us first consider the hottest function momtum. On analysis we find that all the outermost loops in the function are DOALL loops. As a matter of fact, majority of them are parallelized using the OpenMP pragmas in the original source code. Examples include the DOALL loop in momtum.f:459 where most of the array accesses are not aliased and would benefit from scalar privatization.

Outermost loops in the second hottest function .mxkppaij (coverage of 9.3%) have dependence distance of 1 and are therefore classified under the MS category (recall that the systems listed in Table 2 do not have support for data value speculation (DVS)). In other functions, the outermost loops are parallelized in the original source code using OpenMP pragmas or are inherently parallel otherwise.

Lastly, analyzing the function-level coverage profile we note that the library calls ..mod_advm, ..MOD_advm, ..atan2 and ..exp account for 5.7%, 5.4% and 4.3% of the total execution time respectively. This suggests that parallelization of the above library calls bear a large potential for speeding up HYCOM.

5.10 Sweep3d

Sweep3d represents the heart of a real ASCI application. It solves a 1-group time-independent discrete ordinates (Sn) 3D cartesian (XYZ) geometry neutron transport problem.

We compiled SPPM using the IBM xlf compiler with the -O3 -qhot -qarch=pwr5 -pg options and ran the binary on POWER5. On analyzing the function-level coverage profile, we note that more than 55% of the total coverage belongs to the (outermost) contain the loop above could not be parallelized due to the presence of function calls.

Although the loop at line number 353 is already parallelized using OpenMP pragmas, it should not be “disregarded” for analysis while evaluating the available parallelism. This stems from the need for exploiting nested TLP which in turn is driven by the increasing number of cores on a chip [25]. For example, let us consider the loop at line number 416 shown below. The loop is inside the already parallelized loop discussed above.

From the code snippet we note that the writes to the arrays phi, phi0 and phi0b do not induce a loop-carried dependence. The loop is a DOALL loop subject to scalar privatization and application of IVE on the fjixed. Overall, > 75% of the total coverage is inherently parallel.\(^4\)

\[ \text{DO PARALLEL} \]

\[ i = 1, \ldots, pref \]

\[ \text{ci} = \text{si} * \text{phi}(i) \]

\[ \text{dl} = (\text{dxi}(i,j) + \text{ci} * \text{cj} + \text{ck}) \]

\[ \text{ti} = 1.0 / \text{dl} \]

\[ \text{ql} = (\text{phi}(i) + \text{phi0}(i) * \text{phi}(i,j,lk,mi) + \text{phi0b}(i,j,mi)) \]

\[ \text{phi}(i,j)=\text{ti} \]

\[ \text{ti} = \frac{2.0d0*\text{phi}(i) - \text{phi0}}{\text{ti} = \frac{2.0d0*\text{phi}(i) + \text{phi0b}(i,j,mi)}{fjixed = 0} \]

\[ \text{i} = 1! \]

\[ \text{continue} \]

\[ \text{if} \left( \text{ti} \right) \{ \text{ti} = 0.0d0 \} \]

\[ \text{ti} = 0.0d0 \]

\[ \text{phi}(i,j) = \text{ti} \]

\[ \text{endif} \]

\[ \text{if} \left( \text{ti} \right) \{ \text{ti} = 0.0d0 \} \]

\[ \text{ti} = 0.0d0 \]

\[ \text{phi}(i,j) = \text{ti} \]

\[ \text{endif} \]

\[ \text{if} \left( \text{ti} \right) \{ \text{ti} = 0.0d0 \} \]

\[ \text{ti} = 0.0d0 \]

\[ \text{phi}(i,j) = \text{ti} \]

\[ \text{endif} \]

5.11 Parallelization Spectroscopy Summary

In this subsection, we summarize the spectroscopic analysis of the parallelization of the production HPC codes we studied (listed in Table 1). We report the \( L(T) \) metric as it is representative of the performance potential of a technique under consideration.

From Table 5 and the detailed case-by-case analysis presented earlier in this section we make the following conclusions:

- Existing transformations for program parallelization are “sufficient” for exploiting most of the TLP (from coverage standpoint) available in production HPC codes, if compilers and tools were capable of detecting when to apply these techniques. The remaining TLP could not be extracted using the existing techniques because of, but not limited to, the presence I/O as

\(^3\)The intrinsics are mapped on to optimized library calls.

\(^4\)The function calls in the outer loops account for less 2% of the total coverage!
Table 5. Summary of parallelization spectroscopy

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Reduction</th>
<th>Privatization</th>
<th>Loop Translations</th>
<th>Symbolic Analysis</th>
<th>Call-site Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMGe</td>
<td>✗</td>
<td>1.0</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>CrystalMk</td>
<td>0.52</td>
<td>0.81</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>HSImk</td>
<td>✗</td>
<td>1.0</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>CPMD</td>
<td>✗</td>
<td>0.91</td>
<td>✗</td>
<td>0.91</td>
<td>0.91</td>
</tr>
<tr>
<td>POP</td>
<td>✗</td>
<td>0.46</td>
<td>✗</td>
<td>✗</td>
<td>0.54</td>
</tr>
<tr>
<td>UMT2K</td>
<td>✗</td>
<td>0.67</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>RF-CTH</td>
<td>✗</td>
<td>0.72</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>SPPM</td>
<td>✗</td>
<td>1.0</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>HYCOM</td>
<td>✗</td>
<td>0.69</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Sweep3d</td>
<td>✗</td>
<td>1.0</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

in the case of RF-CTH or due to may-dependences as in the case of UMT2K or due to dependences with a very small dependence distance. A relatively small value of $S_{\text{cov}}(L,T)$ in the case of HYCOM can be attributed to high (15%) coverage of library calls. Although other techniques such as thread-level speculation (TLS) can be employed for parallelizing HPC codes beyond what can be done using the existing techniques, the speedup achievable via such techniques would be small, as evidenced from the high (close to a maximum value of 1.0) of $S_{\text{cov}}(L, T)$. This is akin to the results reported by Bova et al. for an Euler flow code [8], i.e., most of the TLP in a HPC code can be harnessed in a non-speculative fashion (via OpenMP/MPI directives) and with very little source code alteration.

There is a critical need to develop richer program semantics to guide the compiler in determining which program transformations to apply and for run-time parallelization. This would minimize the sensitivity of program parallelization with respect to the strength of dependence analysis of the particular compiler used. For example, augmenting the existing set of OpenMP directives to support run-time dependence checks can enable parallelization of the hot loop in CPMD.

Better expressivity of the inherent TLP at the programming language level is required to assist the compiler. Recent efforts to this end are exemplified by support for explicit modeling of iteration spaces in the Chapel [10] programming language.

Support for feedback to the user is required, akin to the technique proposed by Wu et al. in [58], to assist algorithmic or application-level transformation for program parallelization.

From Table 5 we note that, for the applications we studied, no loop transformations such as loop peeling, loop permutation were required to enable thread-level parallel execution. For instance, loop permutation is not warranted to parallelize the triply nested loop shown in subsection 5.3. Likewise, loop distribution is not required to enable parallelize the multi-way loop [43] shown in subsection 5.9. Of course, this need not be true for all the HPC codes. Further, such loop transformations can potentially assist in achieving higher level of TLP and and improved performance on different architectures. For example, let us consider a doubly nested 9GALL loop wherein the outer loop has 4 iterations and the inner loop has 10,000 iterations. Given an octal core machine, the outer loop cannot be parallelized into 8 threads. In order to alleviate this limitation, the loops can be interchanged, which would enable 8-way parallelization of the outer loop in the transformed loop nest. In a similar vein, loop tiling [57] can be employed to exploit temporal and spatial locality (as applicable) thereby improving the overall multithreaded performance.

6. Related Work

Weinberg et al. proposed a methodology to obtain architecture-neutral characterizations of the spatial and temporal locality exhibited by the memory access patterns of HPC applications [56]. Chevereau et al. presented a comparative analysis of HPC workloads in [11]. In particular, they studied characteristics such as (a) instruction decomposition (floating-point and integer, loads, stores, branches and software prefetch instructions), (b) temporal and spatial locality, (c) sensitivity with respect to cache size cache associativity, (d) data sharing analysis and (e) efficacy of data prefetching. In [39], Nagarajan et al. presented a scheme for proactive fault tolerance for arbitrary MPI codes, wherein processes automatically migrate from “unhealthy” nodes to healthy ones. Their scheme leverages virtualization techniques combined with health monitoring and load-based migration.

In [8], Bova et al. describe their experiences converting an existing serial production code to a parallel code combining both MPI and OpenMP. The scope of the paper is restricted to a harbor response simulation code. Likewise, techniques for hybrid parallelization – based on MPI/OpenMP or MPI/Pthreads – of applications ranging from molecular dynamics [21], costal wave analysis [37], atmospheric research [35] and computational fluid dynamics [13] have been proposed. In [19], Gropp et al. described their performance tuning experiences with a 3-d unstructured grid Euler flow code from NASA. In [53], Verma et al. investigate the use of power management techniques for HPC applications on modern power-efficient servers with virtualization support. They showed that for HPC applications, working set size is a key parameter to take care of while placing applications on virtualized servers. None of the aforementioned works address evaluation of the available TLP in HPC applications and parallelization spectroscopy. We believe that our work is complimentary to the above.

Recently, Bridges et al. [9] and Zhong et al. [60] presented techniques for uncovering thread-level parallelism in sequential codes. Akin to our work, Zhong et al. explored the applicability of a set of transformations, such as, variable privatization, reduction variable expansion, speculative loop fission and speculative prematerialization, to parallelization of applications in the SPEC CPU benchmark suite. Contrary to our experimental methodology, their results are simulation-based and further, they assume a perfect memory system. Given that they address non-HPC codes, we believe that their work is complimentary to the work presented in this paper.

7. Conclusion

In this paper, we present a detailed measured analysis of the available thread-level parallelism in production codes. The codes are industrial or are widely used publicly available applications. The measurement was done on two different, viz., POWER5 and Xeon, architectures. Based on the analysis, we draw the following conclusions:

- First, our measurement and analysis shows that more than 75% of the total coverage is inherently parallel in the applications we studied. The corresponding program regions, loops in the current context, can be marked parallel using OpenMP [41] pragmas.
- Second, for applications such as POP, parallelism is available at higher levels, i.e., beyond the lowest level of a calling context. Therefore, higher level program analysis is necessary to assess the true coverage of the IP category.
- Third, the benchmarks listed in Table 1 do not use parallel packages (wherever applicable) for routines such as Cholesky factorization [1] or FFT [14]. This has two pitfalls: (i) it subjects the parallelization of these routines to the strength of dependence analysis of the specific compiler used and (ii) the code generated by the compiler does not measure up with the code of the hand-tuned packages. Thus, the use of these libraries is imperative from both performance and productivity perspective.
Fourth, as evident from Section 5, library routines account for a significant percentage of the total coverage in many benchmarks. From this we conclude that parallelization of libraries such as libc would assist in achieving better performance in many applications.

As future work, we plan to evaluate the $R_{\text{cost}}^M(T_j, T_k)$ metric for the transformations discussed in this paper. The efficacy of a given set of transformations depends on the order in which the transformations are applied. This - also referred to as the phase ordering problem - is well known to be NP-complete [16]. We also plan to study the cache performance of the applications during concurrent execution.

References
Exploiting Hidden Parallelism

Erik Altman
IBM Research

ABSTRACT

Many studies have found large amounts of parallelism are present even in integer apps like SPECint. A machine with unbounded resources and an oracle for branch prediction and memory disambiguation could execute hundreds or thousands of instructions per cycle. This talk will explore some ideas about finding and exploiting such parallelism.

BIO

Erik Altman is a Research Staff Member and Manager of the Dynamic Optimization Group at the IBM T.J. Watson Research Center. His research has explored dynamic binary translation and optimization, compilers, architecture, and tooling for massive multicore systems. He has authored or co-authored more than 30 conference and journal papers, and filed more than 30 patents. He was one of the originators of IBM's DAISY binary translation project, that allowed VLIW architectures to have high performance and achieve 100% binary compatibility with PowerPC. He was also one of the original architects of the Cell processor used in the Sony Playstation 3. He has been the program chair and/or general chair of the PACT, CASES, and P=ac2 Conferences and has served on numerous program committees. He has served as guest editor of IEEE Computer, the ACM Journal of Instruction Level Parallelism (JILP), the Springer Journal of Design Automation for Embedded Systems, and the IBM Journal of Research and Development. He is currently the Chair of ACM SIGMICRO.
1. INTRODUCTION

The rise of the chip multiprocessor (CMP) has marked the end of the road for monolithic cores, requiring the development of highly-threaded applications to take advantage of the many cores available in a CMP. A CMP optimized for such future workloads is focused on achieving the best area and power efficiencies, as this enables the integration of a larger number of cores within a single chip.

Unfortunately, cores that are efficient with respect to area and power are bound to be lacking in single-threaded performance relative to conventional monolithic cores. This creates a real performance backwards-compatibility headache for CPU designers; consumers expect performance of next-generation CPUs to remain competitive on legacy poorly-threaded workloads. For this reason, consumer CMPs have thus far been designed as a collection of monolithic cores. However, these designs sacrifice throughput performance to maintain performance on single-threaded workloads.

As an alternative to this approach, we propose the design of CMPs with a focus on future highly-threaded applications, treating the performance of legacy single-threaded code as a backwards-compatibility concern. In other words, the underlying hardware should make use of simple, efficient cores that enable the best performance on highly-threaded workloads. At the same time, several clever tricks can be used to maintain single-threaded performance without resorting to complex additional hardware. Specifically, our approach relies on the use of dynamic binary translation to transparently accelerate legacy applications on a throughput-oriented CMP.

The DBT86 dynamic binary translation research framework is designed from the ground up to explore new opportunities presented by CMPs. The availability of multiple cores presents a broad array of novel opportunities for a dynamic binary translator. In this paper, we explore how DBT86 can make use of additional cores for profiling, optimization, and execution of legacy single-threaded workloads.

Section 2 provides a brief tour of the DBT86 environment. Section 3 presents the DBT86 code generation strategy, which leverages idle cores to perform optimization. Section 4 presents CRISP, a performance-monitoring technique that enables rapid profiling within a CMP. Section 5 presents RASP, a technique for the dynamic extractions of speculative thread-level parallelism, while section 6 presents a rough sketch of the performance that could be achieved with a DBT86-style approach.

2. THE DBT86 ENVIRONMENT

DBT86 translates x86 machine code into RISC microcode, similar to Transmeta’s CMS [4]. The choice of x86 for this research platform is apt due to the abundance of legacy x86 software, much of which has been distributed in binary form without source code.

2.1 Instruction Set Architecture

The DBT86 system functions as a layer between x86 code and a microprocessor capable of executing a RISC microcode designed to efficiently implement x86. Most x86 instructions map to between one and three µops. Figure 1 demonstrates µops generated for some sample x86 instructions.

![Table: Converting x86 to RISC microcode](image)

Figure 1. Converting x86 to RISC microcode
Some of the advantages of generating microcode include access to microarchitectural scratch registers, hardware support for mapping indirect branch targets, and the ease of adding ISA extensions to support speculative threading. Without these features, dynamic binary translation to the same ISA incurs a slight performance overhead [3], though it should certainly be possible to apply many of these same techniques in a dynamic binary translator targeting x86 rather than microcode.

2.2 Runtime Environment

The DBT86 runtime environment is responsible for resource management, handling low-level exceptions, profiling running code, and selecting regions for optimization. The total memory footprint for the runtime environment is roughly 128MB, which includes DBT86 code and a 32MB translation cache for dynamically-generated code. The remaining space is consumed by auxiliary data structures which enable dynamic linking between translations and two-way mapping between x86 addresses and the resulting microcode, in addition to a variety of data structures supporting dynamic profiling and code generation.

![DBT86 block diagram](image)

The major components of the runtime system, depicted in Figure 2, are the code manager, profile manager, code generator, and interpreter. The code manager is responsible for storing and indexing translated code, as well as handling low-level exceptions. The profile manager is responsible for collecting profiling data and guiding optimization. To execute code, the runtime first queries the code manager for an existing translation. If no translation is found, the runtime invokes the code generator to create a translation. As a fallback, any obscure operation not supported by code generation is executed using the interpreter, although this does not occur frequently enough in the code that we have examined to be relevant to performance. When resources are exhausted, the code manager can query the profile manager for an inactive translation to return to the resource pool.

2.3 Runtime Data Structures

Entries in the microcode translation cache are allocated at the granularity of an instruction-cache line. The physical address corresponding to code in the microcode translation cache is known as a host address, while any address in the context of the original program is known as a guest address.

The basic container for translated code is a Translation, which may consist of multiple lines of translated code. The entry point for a translation may correspond to one or more guest addresses, each of which is allocated a unique Alias record. Alias records are hashed for easy lookup of Translation records using guest addresses. There is also a mapping with one entry per line in the microcode translation cache that references the containing Translation, which is used for purposes like exception handling and profiling.

A Translation may have multiple Fixup records, which enable linking between multiple translations. The Fixup structure contains the host address at which to apply the fixup as well as the guest address for the target. Fixup records are hashed based on guest address to allow fast patching of all references when committing or freeing a translation.

In order to detect when a translation becomes stale due to modification or unloading of the original guest code, each Translation has one or more Interval records associated with it. To allow for precision with a compact representation, DBT86 employs a sparse bitvector approach. Each Interval record contains the guest address of a 4KB page, with a bitvector specifying coverage at 32B granularity. Most translations require just a single Interval record. Interval records are hashed based on guest address. When a write occurs to a protected region of a code page, the corresponding translations can then be looked up efficiently and invalidated.

Mapping from the host address of the running translated code back to the corresponding guest
address involves the use of Map records. This mapping is used when processing profile data. Not every host instruction requires its own Map record; control-flow profiling only requires one entry per basic block. For compact storage, Map records are stored and allocated in a cache-line-sized unit known as a MapBlock. A Translation may have one or more MapBlock records.

Several additional data structures track profiling information. Edge profiles are stored in a counting filter. Any given edge \((a,b)\) results in several counters within a table being incremented, each determined by a unique hash function. When querying the profiled heat of an edge, the same hash functions are used to retrieve the counters. The minimum of these counter values determines the actual heat of the edge. Although this data structure can theoretically have false positives, we find that it works well in practice. In order to prevent counter overflow, each profiling event is accompanied by a slight decay of a few profiling counters.

A binary heap tracks the active translations based on heat, as determined by the number of profiling samples that occur within a translation. This structure allows for fast selection of the least-used translation when resources are exhausted, with logarithmic overhead for insertion, removal, and update. A smaller data structure collects profiles of non-return indirect branches for use in code generation.

3. CODE GENERATION

The code generation strategy in DBT86 differs in several significant ways from prior dynamic binary translation systems.

3.1 Initial Strategy

Transmeta’s CMS featured two modes of operation, interpretation and translation. In order to keep overheads low for run-once code, CMS would interpret code and only translate it if execution exceeded some threshold. The downside to this approach is that interpretation is very costly relative to the execution of translated code.

DBT86 takes a different approach to initial code generation. The first time that a new piece of code is encountered, DBT86 generates an unoptimized translation for a short section of code. This “quick-and-dirty” generation of unoptimized code is only marginally slower than interpretation; once an instruction has been decoded, it is a simple matter of a few shifts and or-operations to assemble the instruction and store it into an instruction buffer.

Based on an execution profile of start-up and shutdown for Microsoft Word 2002, a representative case of so-called run-once code, just 25% of the static instructions executed are only run once. The DBT86 approach of generating unoptimized code in the first pass is beneficial for these cases, as running unoptimized code is far more efficient than repeated interpretation. The difference in performance between unoptimized and optimized code (typically less than 2x) is marginal compared to the difference in performance between interpretation and execution of optimized code (at least 30x). These results are consistent with prior work on the topic [1][7].

After this initial code generation, DBT86 profiles the running code. As a consequence of the lower overhead of executing unoptimized code relative to interpretation, DBT86 can wait longer prior to generating optimized code. This allows DBT86 sufficient time to collect adequate profile data for large-scale optimization. Hu and Smith present several options for further reducing the initial translation costs [7], though such techniques for reducing the overheads of initial translation are largely orthogonal to the CMP-related opportunities that we seek to explore with DBT86.

3.2 Generating Optimized Code

Transmeta’s CMS was designed in the single-core era, where any dynamic binary translation work represented a direct overhead to program execution. This led to considerable efforts to keep the cost of optimization low. By contrast, the potential to perform optimization on otherwise-idle cores in a CMP can allow heavyweight optimizations to be considered that would once have been cost-prohibitive. In particular, speculative parallelization is a complex optimization that is only sensible for DBT86 to perform when there are idle cores available to execute the parallel code. In such a situation, the idle cores can also be used to perform the analysis and optimization necessary to produce this parallel code. Furthermore, when optimization takes place on otherwise-idle cores, only the lightweight initial translation represents an overhead to the running program.
Another consequence of generating optimized code on idle cores is that the optimization process does not interfere with the real-time behavior of the busy cores. Dynamic binary translation systems from the single-core era tend to focus on small, trace-based program regions. By contrast, the DBT86 framework for generating optimized code has features similar to a conventional compiler, including a control-flow graph (CFG) and static-single-assignment (SSA) representation of dependencies. DBT86 supports optimization of program regions as large as thousands of instructions, enabling complex optimizations like speculative parallelization of non-trivial loops. In summary, the availability of additional cores allows DBT86 to perform more complex optimizations than a conventional dynamic binary translator, while simultaneously presenting lower overhead to the running program.

4. REMOTE PROFILING

Dynamic binary translation relies on profiling to guide the optimization process. DBT86 uses profiling information to identify hotspots for optimization and to monitor the performance of optimized code. With conventional profiling, the profiling takes place on the same core that is running the code, representing a direct overhead to execution. With a given overhead per sample, there is a fundamental tradeoff between overhead and the speed at which samples may be collected. This tradeoff can be tuned by adjusting the sampling interval. To solve this problem, DBT86 uses a profiling scheme called Continuous Remote Interrupt-free Sample Profiling (CRISP).

The hardware model for DBT86 uses a simple variation on conventional performance-monitoring registers, a feature that is already present in most current CPUs. Unlike conventional CPUs, the DBT86 performance-monitoring registers can be accessed from other cores using memory-mapped IO. This involves minimal hardware changes, yet it allows otherwise-idle cores to profile the active cores without interrupting the running program.

Unless the on-chip bandwidth is saturated, there is no overhead whatsoever to the running program while performing profiling. Even when bandwidth is fully saturated, with a sample interval of 1500 clock cycles (assuming concurrent remote profiling by several cores), and requiring two cycles to transfer a cache line, profiling would still represent an overhead of just 0.1%. By contrast, assuming 3000 cycles to collect and process a profiling sample, conventional sample-based profiling would present a hefty 5% overhead at a far slower sampling interval of 60,000 cycles. In this manner, DBT86 can rapidly profile a running program, eliminating much of the latency normally required for profiling prior to optimization.

In an effort to reduce overheads, a conventional dynamic binary translator may only profile while running new code. By contrast, the overhead for profiling with CRISP is so low that it can be performed continuously as the program runs. This enables DBT86 to monitor the performance of optimized code and use runtime feedback to refine its optimizations.

5. SPECULATIVE PARALLELISM

Hardware support for speculation, in the form of a gated store buffer and shadow registers, is used by CMS to provide precise exceptions by enabling rollback to a consistent architectural state. Within a speculative region, CMS is free to reschedule instructions without regard for their original program ordering. This support for speculation also enables speculative optimizations within a thread [10]. More recently, transactional memory [6] has been proposed as a means of simplifying parallel programming. Hardware support for speculation allows transactional memory to run efficiently.

Like CMS, DBT86 uses hardware support for speculation to support precise exceptions. DBT86 goes one step further, however, using this same hardware support to exploit speculative parallelism in a CMP. In a process called Runtime Automatic Speculative Parallelization (RASP), DBT86 can run multiple iterations of a loop on different cores while preserving sequential semantics, thereby accelerating legacy sequential code.

A single hardware checkpoint supports both precise exceptions and RASP, meaning that speculative parallelization in DBT86 is largely making use of hardware structures that already exist to support dynamic binary translation. DBT86 does not provide any special-purpose hardware constructs to support RASP beyond this basic hardware support for speculation, which guarantees the correct ordering of memory references between speculative threads by rolling back and restarting any speculative thread that
has accessed stale data. In particular, RASP does not assume any hardware support for preserving register dependencies between speculative threads, synchronizing data dependencies between speculative threads, value prediction, or selective recovery from violations.

During dynamic binary translation, DBT86 attempts to transform loops into code that can run in parallel on multiple cores. RASP involves various code transformations to break loop-carried dependencies and to preserve register dependencies, as well as the use of CRISP feedback to iteratively tune the generated code. Unlike TLS compilers [2][5][8][9][11], DBT86 enables speculative parallelization of legacy code without recompilation.

We find that RASP can accelerate SPECint2006 benchmarks by an average of 49%, using a cluster of four cores within a CMP to run speculatively-parallelized code. Details of RASP will appear separately in a pending publication.

6. CONCLUSION

As demonstrated with Transmeta’s CMS, dynamic binary translation can greatly improve the performance of simple in-order cores by enhancing instruction scheduling. Although simple in-order cores do not offer compelling performance in comparison to monolithic out-of-order cores, lower area and power consumption make in-order cores a good choice for a throughput-oriented CMP. In addition to the optimization opportunities available to a conventional dynamic binary translation system, our work with DBT86 presents several novel ways to use CMP resources in combination with dynamic binary translation to accelerate programs.

In Figure 3, we project the performance potential of a DBT86-based CMP that makes use of both the conventional dynamic binary translation optimization opportunities used in systems like CMS as well as the opportunities we’ve explored with DBT86. The performance is normalized to a conventional CMP.

As a starting point for this evaluation, we begin with benchmarks that show that clock-for-clock, a simple in-order core like Intel’s Atom achieves about half the performance of Intel’s Core2 processors [12]. The reduced area and power consumption of the simpler core should make it possible to integrate 4x as many in a CMP, achieving double the performance on perfectly-parallel workloads.

![Figure 3. DBT86-based CMP performance projection](image)

We then factor in an estimated 30% performance boost that could be obtained using conventional optimizations like instruction scheduling during dynamic binary translation. Our research with DBT86 has not focused on these optimizations, as they have been extensively investigated in other work. For example, Transmeta [4] reports a 33% average benefit to reordering memory accesses during scheduling, facilitated by hardware support for alias detection. Hu and Smith [7] report an 18% benefit on SPECint2000 using dynamic binary translation to fuse micro-ops into macro-ops. As the combined benefit of just these mentioned techniques exceeds the value used for our estimate, we believe that our performance projection is conservative.

The benefit of conventional single-threaded optimizations is complementary to the 49% performance gain that RASP achieves by speculatively-parallelizing single-threaded workloads. With RASP added to the mix (the third bar in Figure 3), we project that performance parity can be achieved on sequential applications despite the use of simpler cores. On parallel workloads, by contrast, this same CMP would have a commanding performance advantage.

DBT86 can leverage the idle cores in a CMP to profile, optimize, and participate in the execution of a running program. When combined with conventional optimizations performed during dynamic binary translation, the techniques demonstrated in DBT86 permit future CMPs to excel at parallel workloads without sacrificing single-threaded performance.
REFERENCES


Abstract

JavaScript is emerging as the ubiquitous language of choice for web browser applications. These applications increasingly execute on embedded mobile devices, and thus demand responsiveness (i.e., short pause times for system activities, such as compilation and garbage collection). To deliver responsiveness, web browsers, such as Firefox, have adopted trace-based Just-In-Time (JIT) compilation. A trace-based JIT restricts the scope of compilation to a short hot path of instructions, limiting compilation time and space. Although the JavaScript limits applications to a single-thread, multicore embedded and general-purpose architectures are now widely available. This limitation presents an opportunity to reduce compiler pause times further by exploiting cores that the application is guaranteed not to use. While method-based concurrent JITs have proven useful for multi-threaded languages such as Java, trace-based JIT compilation for JavaScript offers new opportunities for concurrency.

This paper presents the design and implementation of a concurrent trace-based JIT that uses novel lock-free synchronization to trace, compile, install, and stitch traces on a separate core such that the interpreter essentially never needs to pause. Our evaluation shows that this design reduces the total, average, and maximum pause time by 89%, 97%, and 93%, respectively compared to the base single-threaded JIT system. Our design also improves throughput by 6% on average and up to 34%, because it delivers optimized application code faster. This design provides a better end-user experience by exploiting multicore hardware to improve responsiveness and throughput.

Categories and Subject Descriptors D.3.4 [Programming Languages]: Processors—Incremental compilers, code generation.

General Terms Design, Experimentation, Performance, Measurement

Keywords Just-In-Time Compilation, Multicore, Concurrency

1. Introduction

JavaScript is emerging as the scripting language of choice for client-side web browsers [10]. Client-side JavaScript applications initially performed simple HTML web page manipulations to aid server-side web applications, but they have since evolved to use asynchronous and XML features to perform sophisticated, interactive dynamic content manipulation on the client-side. This style of JavaScript programming is called AJAX (for Asynchronous JavaScript and XML). Companies, such as Google and Yahoo, are using it to implement interactive desktop applications such as mail, messaging, and collaborative spreadsheets, word processors, and calendars. Because Internet usage on mobile platforms
is growing rapidly, the performance of JavaScript is critical for both desktops and embedded mobile devices. To speed up the processing of JavaScript applications, many web browsers are adopting Just-In-Time (JIT) compilation, including Firefox TraceMonkey [5], Google V8 [11], and WebKit SFE [19].

Generating efficient machine code for dynamic languages, such as JavaScript, is more difficult than for statically typed languages. For dynamic languages, the compiler must generate code that correctly executes all possible runtime types. Gal et al. recently introduced a trace-based JIT compilation for dynamic languages to address this problem and to provide responsiveness (i.e., low compiler pause times and memory requirements) [7]. Responsiveness is critical, because JavaScript runs on client-side web browsers. Pause times induced by the JIT must be short enough not to disturb the end-user experience. Therefore, Gal et al.’s system interprets until it detects a hot path in a loop. The interpreter then traces, recording instructions and variable types along a hot path. The JIT then specializes the trace by type and translates it into native code in linear time. The JIT sacrifices code quality for linear compile times, rather than applying heavy weight optimizations. This trace-based JIT provides fast, lightweight compilation with a small memory footprint, which make it suitable for resource-constrained devices.

On the hardware side, multicore processors are prevailing in embedded and general purpose systems. The JavaScript language however lacks a thread model, and thus all JavaScript applications are single-threaded. This limitation provides the opportunity to perform the JIT and other VM services concurrently on another core, transparently to the application, since the application is guaranteed not to be using it. Unfortunately, state-of-the-art trace-based JIT compilers are sequential [7, 5, 18], and have not exploited concurrency to improve responsiveness.

In this paper, we present the design and implementation of a concurrent trace-based JIT compiler for JavaScript that combines responsiveness and throughput for JavaScript applications. We address the synchronization problem specific to the trace-based JIT compiler, and present novel lock-free synchronization mechanisms for wait-free communication between the interpreter and the compiler. Hence, the compiler runs concurrently with the interpreter reducing pause times to nearly zero.

Our mechanism piggybacks a single word, called the compiled state variable (CSV), on each trace, using it as a synchronization variable. Comparing with CSV synchronizes all of the compilation actions, including checking for the native code, preventing duplicate traces, and allowing the interpretation to proceed, without using any lock.

We introduce lock-free dynamic trace stitching in which the compiler patches new native code to the existing code. Dynamic trace stitching prevents the compiler from waiting for trace stitching while the interpreter is executing the native code, and reduces the potential overhead of returning from native code to the interpreter.

We implement our design in the open source TamarinTracing VM, and evaluate our implementation using the SunSpider JavaScript benchmark suite [20] on three different hardware platforms. The experiments show that our concurrent trace-based JIT implementation reduces the total pause time by 89%, the maximum pause time by 93%, and the average pause time by 97% on Linux. Moreover, the design improves the throughput by an average of 6%, with improvements up to 34%. Our concurrent trace-based JIT virtually eliminates compiler pause times and increases application throughput. Because tracing overlaps with compilation, the interpreter prepares the trace earlier for subsequent compilation, thus the JIT delivers the native code more quickly. This approach also opens up the possibility of increasing the code quality with compiler optimizations without sacrificing the application pause time.

2. Related Work

Gal et al. proposed splitting trace tree compilation steps into multiple pipeline stages to exploit parallelism [6]. This is the only work we can find seeking parallelism in the trace-based compilation. There are a total of 19 compilation pipeline stages, and each pipeline stage runs on a separate thread. Because of data dependency between each stage and the synchronization overhead, the authors failed to achieve any speedup in compilation time. We show having a parallel compiler thread operating on an independent trace provides more benefit than pipelining compilation stages. With proper synchronization mechanisms, our work successfully exploited parallelism in the trace-based JIT by allowing
tracing to happen concurrently with the compilation, even when only one compiler thread was used.

Kulkarni et al. explored maximizing throughput of background compilation by adjusting the CPU utilization level of the compiler thread [15]. This technique is useful when the number of application threads exceeds the number of physical processors and the compiler thread cannot fully utilize a processor resource. They conducted their evaluation on method-based compilation, though the same technique can be applied to trace-based compilation. However, because JavaScript is single-threaded, it is less likely that all the cores are fully utilized in today’s multicore hardware. Hence, the effect of adjusting CPU usage levels will not be as significant as it is in multi-threaded Java programs.

A number of previous efforts have sought to reduce compilation pause time in method-based JIT. SELF-93 VM introduced adaptive compilation strategies for minimizing application pause time [13]. When a method is invoked for the first time, the VM compiles it without optimizations using a light weight compiler. If method invocations exceed a threshold, the VM recompiles the method with more aggressive optimizations. While the SELF-93 VM provided reasonable responsiveness, it must pause the application thread for compilation when initially invoked.

Krintz et al. implemented profile-driven background compilation in the Jalapeño Virtual Machine (now called Jikes RVM) [14, 2]. In multiprocessor systems, a background compiler thread overlaps with application execution, which reduces compilation pause times. Jikes RVM also applied lazy compilation, where the JIT only compiles the method on demand within a class instead of compiling every method in a class at class loading time. When the method is invoked for the first time before the optimized code is ready, the VM pauses the application and run the baseline compiler.

These novel techniques have made adaptive compilation in method-based compilation practical in Java Virtual Machines, such as Sun HotSpot [16], IBM J9 [17], and Jikes RVM [1]. However, issues specific for trace-based JIT has not been successfully evaluated by any previous work.

3. Background

3.1 Dynamic Typing in JavaScript

JavaScript is a dynamically typed language. The type of every variable is inferred from its content dynamically. Furthermore, the type of JavaScript variables can change over time as the script executes. For example, a variable may hold an integer object at one time and later hold a string object. A consequence of dynamic typing is that operations need to be dispatched dynamically. While the degree of type stability in JavaScript is the subject of current studies, our experiences and empirical results indicate that JavaScript variables are type stable in most cases. This observation suggests that type-based specialization techniques pioneered in Smalltalk [4] and later used in Self [12] and Sun HotSpot [16] have the potential for tremendously improving JavaScript performance.

3.2 Trace-based JIT Compilation

Hotpath VM is the first trace-based JIT compilation introduced for Java applications in a resource-constrained environment [8]. The authors later explored trace-based JIT for dynamic languages, such as JavaScript [7].

The trace-based JIT compiles only frequently executed path in a loop. Figure 1 shows an example of how the interpreter identifies a hot path, and expands it. Initially, the interpreter executes the byte code instructions, and identifies the hot loop with backward branch profiling which operates as follows. When the execution reaches the backward branch, the interpreter assumes it a loop backedge and increments the counter associated with the branch target address. When the counter reaches a threshold, the interpreter enables tracing, and records each byte code instruction to a
trace buffer upon execution. When the control reaches back to the address where the tracing started, the interpreter stops tracing and the compiler compiles the trace to native code. As the interpreter is not doing an exact path profiling, the traced path may or may not be the real hot path. The first trace in a loop is called a trunk trace.

Instructions are guarded if they potentially diverge from the recorded path. If a guard is triggered, the native code side-exit back to the interpreter, and begin interpreting from the branch that caused the side-exit. The interpreter counts each side-exit to identify the frequent side-exit. When a side-exit is taken beyond a threshold, it means the loop contains another hot path, and the interpreter enables tracing from the side-exit point until it reaches the address of the trunk trace. This trace is called a branch trace. A branch trace is compiled and the code is stitched to the trunk trace at the side-exit instruction. As the interpreter finds more hot paths, the number of branch traces grows forming a trace tree.

Since the compilation granularity is a trace, which is smaller than a method, the total memory footprint of the JIT is smaller than that of method-based JITs. And because no control flow analysis is required, start-up compilation time is less than that of the method-based compilers. However, as optimization opportunities are limited, the final code quality may not be as good as code generated by method-based compilation. Therefore, trace compilation is suitable for embedded environments where resources are limited, or the initial JIT cost is far more important than the steady state performance.

4. Design and Implementation

4.1 Parallelism to Exploit

To design a proper synchronization mechanism to maximize the concurrency, we must understand what parallelisms can be exploited. Figure 2 explains an execution flow example of sequential and concurrent JIT. As the compilation phase is offloaded to a separate thread, the interpreter is responsive and making progress while compilation happens, as is common for generic concurrent JIT compilers. For trace-based JIT, tracing must precede the compilation phase. If tracing can happen concurrently with compilation, subsequent compilation may start earlier, and deliver the native code faster. Furthermore, more hot paths can be compiled during the execution. We can expect to achieve throughput improvements as well as a reduction in the pause time. The concurrent JIT also opens the possibility to do more aggressive optimizations without hurting pause time. The following sections explain how we designed the synchronization to achieve the parallelism shown in Figure 2.

4.2 Compiled State Variable

In the trace-based JIT compiler, the interpreter changes state at loop entry points. As shown in Figure 3, when the control flow reaches a loop entry point, the interpreter must identify four different states. First, if compiled native code exists for the loop, the interpreter calls it. The native code executes until the end of the loop or a side-exit is taken. Second, if the loop has never been traced and the loop is a hot loop, the interpreter executes byte code with tracing enabled. Identifying hot loop path is explained in Section 3 in detail. Third, if tracing is currently enabled at the loop header, the interpreter disables it and requests compilation. While compiling the trace, the interpreter continues to execute the program. Fourth, if the loop is cold, the interpreter
increments the associated counter and keeps on interpreting the byte codes.

Checking all these cases at a loop header requires a synchronization with the compiler thread. Otherwise, race conditions may cause overhead or incorrect execution. For example, the interpreter may make duplicate compilation requests, or trace the same loop multiple times. The simplest synchronization method is using a coarse-grained lock around the checking routine. However, the lock can easily be contended after the compilation request is made, especially with a short loop body, because the control reaches the loop header frequently. We could use a fine-grained lock for accessing each loop data structure. However, this is also infeasible because the native code for the loop can change as the trace tree grows, and holding a lock while executing the native code would stall the compiler too often.

To overcome these challenges, we design a lock-free synchronization technique using a compiled state variable (CSV). A word size CSV piggybacks on each loop data structure, and it is aligned not to cross the cache line. Thus, stores to it are atomic. The value of the CSV is defined as shown in Table 4.2. By following simple but efficient ways of incrementing the CSV value, the state check at the loop header can be done without any explicit synchronization. The initial value of CSV is zero, and only the interpreter increments 0 to 1 when it requests a compilation. As it is a local change, the interpreter sees the value 1 on the subsequent operations before the compiler sees the value 1. The compiler changes the value 1 to 2 after it registers the native code to the loop data structure. Thus, when the interpreter reads the value 2, it is guaranteed that the native code is ready to call. Therefore, the pause time for waiting is almost zero for both the interpreter and the compiler, maximizing the concurrency.

When the compiler makes a JIT request, the trace buffer is pushed to a queue before the CSV is incremented to 1. We use a simple synchronized FIFO queue for the JIT request, because it is normally not contended. However, a generic, concurrent, lock-free queue for one producer and consumer [9] could always replace this queue, but we think it would not affect performance.

4.3 Dynamic Trace Stitching

The trace-based JIT specializes types and paths, and injects guard instructions to verify the assumptions for the type and path of the trace. Guards trigger side-exit code if the assumption is not met, and returns the control back to the interpreter.

If two or more hot paths exist in a loop, the first hot path will be compiled normally, but the subsequent hot paths will frequently trigger guards. As explained in Section 3, the interpreter traces from the branch that caused the side-exit (branch trace), and compiles it. As more hot paths are revealed, trunk and branch traces form a trace tree. Recompiling the whole trace tree is good for the code quality, but the compilation time will grow quadratic if the whole trace tree is recompiled every time a new trace is attached to the tree. Also, this strategy would keep the trace buffer in memory for future recompilation, which is infeasible in memory constrained environments. Instead of recompiling the whole tree, we use trace stitching technique. Trace stitching is a technique that compiles the new branch trace only, and patches the side-exit to jump to the branch trace native code.

Branch patching modifies code that is produced by more than one trace. Hence, it is probable that interpreter is executing the native code at the same time that the compiler wants to patch it. Naive use of a lock around the native code will incur a significant pause time on both the interpreter and the compiler. Waiting becomes a problem if time spent in the native code grows large, reducing the overall concurrency. The compiler may also make a duplicate copy of the code instead of patching, or delay the patching until the native code exits to the interpreter. Either method has inefficiencies, and we propose lock-free dynamic trace stitching for the branch patching. The key factor of dynamic trace stitching is that a side-exit jump is a safe point where all variables are synchronized to the memory. We use each side-exit jump instruction as a placeholder for the patching. When the compiler generates the native code for the branch trace, both jumping to the previous side-exit target or jumping to the branch trace code does not change the program semantic. Therefore, if the patching is atomic, the compiler can patch the jump instruction directly without waiting for the interpreter. If the branch target operand is properly aligned, patching is done by a single store instruction. There is no harmful data race even without any lock. With these benign data races, the interpreter and the compiler run concurrently without pausing.
Table 1. Value of Compiled State Variable (CSV) at a loop header.

<table>
<thead>
<tr>
<th>Description</th>
<th>Action</th>
<th>CSV</th>
</tr>
</thead>
<tbody>
<tr>
<td>has native code</td>
<td>Call native code</td>
<td>2</td>
</tr>
<tr>
<td>compilation already requested</td>
<td>normal interpretation</td>
<td>1</td>
</tr>
<tr>
<td>Hot loop</td>
<td>Enable tracing</td>
<td>0</td>
</tr>
<tr>
<td>Cold loop</td>
<td>normal interpretation</td>
<td>0</td>
</tr>
<tr>
<td>Trace enabled</td>
<td>Disable tracing and request compilation</td>
<td>0 to 1</td>
</tr>
</tbody>
</table>

5. Preliminary Results

5.1 Experiments Setup

We evaluate our implementation on Intel Core 2 Quad processor 2.4GHz running Linux 2.6 kernel. We run SunSpider benchmark suite [20], which is widely used to test and compare the JavaScript engine on web browsers. By default, we ran 50 runs and averaged the results. For easy comparison, all graphs are presented so that the lower bar represents the better result.

5.2 SunSpider Benchmarks Characterization

The SunSpider benchmark suite is a set of JavaScript programs intended to test performance [20]. It is widely used to test and compare the JavaScript engine on web browsers, such as Firefox SpiderMonkey, Adobe ActionScript, and Google V8. Table 2 characterizes the benchmarks running on original TamarinTracing VM.

5.3 Pause time reduction

We evaluate application pause time using total, average, and maximum pause time. Total pause time for running a benchmark is a good indicator of the application’s responsiveness, and the average reflects the end-user experience. Many small pauses are better than one big pause in terms of responsiveness [3]. We also compare maximum pause time, which is the most noticeable pause to the end-user, therefore we want it to be as low as possible.

Figure 4 demonstrates that our concurrent JIT implementation reduces both maximum and total pause time significantly. The y-axis is the pause time normalized to the pause time in the sequential JIT. A value of 1.0 means that the pause time is the same, and 0.1 means the pause time is reduced by 90%. Tics at the top of each bar shows 95% confidence interval.

Geometric mean shows that we reduced the total pause time by 89% and 93% for the maximum pause time, showing a huge improvement in responsiveness. Furthermore, the average pause has reduced by 97% of the sequential JIT, which shows the implementation successfully avoided long pauses.

The concurrent JIT was more successful on longer compilation time per trace. crypto-md5 has the highest per trace compilation time, compiling six traces for 25% of the execution time. It also achieves the best reduction in pause time, with 99% for all three metrics.

5.4 Throughput improvements

Figure 5 shows the speedup for each configuration. The first bar represents the sequential JIT, and the second bar shows the interpreter thread activity in the concurrent JIT. This thread activity includes the interpreter, native code, and pause time caused by compilation requests. The third bar shows the compile time of the compiler thread. The y-axis is the speedup normalized to the execution time of the sequential JIT. Hence, bar 2 less than 100% is the speedup. The concurrent JIT achieves 6% speedup on average, and achieves up to 34% on s3d-cube. The speedup in s3d-cube is due to increasing the number of compiled traces.

6. Conclusion

In this paper, we showed that even though JavaScript language itself is currently single-threaded, both its throughput and responsiveness can benefit from multiple cores with our concurrent JIT compiler. This improvement is achieved by running the JIT compiler concurrently with the interpreter. Our results show that most of the compile-time pauses can be eliminated, resulting in a total, average, and maximum reduction in pause time by 89%, 97%, and 93%, respectively. Moreover, the throughput is also increased by an average of 6%, with a maximum of 34%. This paper demonstrates a way to exploit multicore hardware to improve application performance and responsiveness by offloading system tasks.

References

[1] Alpern, B., Attanasio, D., Barton, J. J., Burke,
Table 2. Workload characterization of SunSpider benchmarks with sequential Tamarin JIT.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Bytecode (bytes)</th>
<th>Compiled Traces</th>
<th>Compilation (%)</th>
<th>Native (%)</th>
<th>Interpreter (%)</th>
<th>Runtime (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>access-binary-trees</td>
<td>697</td>
<td>37</td>
<td>5.4</td>
<td>89.1</td>
<td>5.5</td>
<td>74</td>
</tr>
<tr>
<td>access-fannkuch</td>
<td>823</td>
<td>49</td>
<td>2.4</td>
<td>94.2</td>
<td>3.3</td>
<td>117</td>
</tr>
<tr>
<td>access-nbody</td>
<td>2,202</td>
<td>27</td>
<td>3.5</td>
<td>91.6</td>
<td>4.9</td>
<td>144</td>
</tr>
<tr>
<td>access-nsieve</td>
<td>543</td>
<td>14</td>
<td>1.4</td>
<td>96.8</td>
<td>1.7</td>
<td>56</td>
</tr>
<tr>
<td>bitops-3bit-bits-in-byte</td>
<td>414</td>
<td>6</td>
<td>4.0</td>
<td>89.7</td>
<td>6.3</td>
<td>12</td>
</tr>
<tr>
<td>bitops-bits-in-byte</td>
<td>385</td>
<td>15</td>
<td>1.5</td>
<td>96.5</td>
<td>2.1</td>
<td>40</td>
</tr>
<tr>
<td>bitops-bitwise-and</td>
<td>264</td>
<td>3</td>
<td>0.2</td>
<td>99.4</td>
<td>0.4</td>
<td>179</td>
</tr>
<tr>
<td>bitops-nsieve-bits</td>
<td>586</td>
<td>11</td>
<td>1.4</td>
<td>96.6</td>
<td>2.0</td>
<td>50</td>
</tr>
<tr>
<td>controlflow-recursive</td>
<td>504</td>
<td>35</td>
<td>8.3</td>
<td>84.5</td>
<td>7.3</td>
<td>28</td>
</tr>
<tr>
<td>crypto-aes</td>
<td>7,004</td>
<td>158</td>
<td>11.4</td>
<td>63.2</td>
<td>25.4</td>
<td>150</td>
</tr>
<tr>
<td>crypto-md5</td>
<td>5,470</td>
<td>6</td>
<td>24.6</td>
<td>17.4</td>
<td>58.0</td>
<td>120</td>
</tr>
<tr>
<td>crypto-sha1</td>
<td>3,236</td>
<td>26</td>
<td>9.2</td>
<td>52.8</td>
<td>38.0</td>
<td>31</td>
</tr>
<tr>
<td>math-cordic</td>
<td>832</td>
<td>9</td>
<td>1.8</td>
<td>95.0</td>
<td>3.1</td>
<td>32</td>
</tr>
<tr>
<td>math-partial-sums</td>
<td>758</td>
<td>11</td>
<td>1.3</td>
<td>93.2</td>
<td>5.5</td>
<td>41</td>
</tr>
<tr>
<td>math-spectral-norm</td>
<td>841</td>
<td>35</td>
<td>7.8</td>
<td>78.3</td>
<td>13.9</td>
<td>36</td>
</tr>
<tr>
<td>s3d-cube</td>
<td>4,918</td>
<td>188</td>
<td>8.4</td>
<td>41.6</td>
<td>50.0</td>
<td>155</td>
</tr>
<tr>
<td>s3d-morph</td>
<td>573</td>
<td>14</td>
<td>1.5</td>
<td>95.9</td>
<td>2.6</td>
<td>81</td>
</tr>
<tr>
<td>s3d-raytrace</td>
<td>7,289</td>
<td>147</td>
<td>9.3</td>
<td>68.1</td>
<td>22.6</td>
<td>170</td>
</tr>
<tr>
<td>string-fasta</td>
<td>1,426</td>
<td>22</td>
<td>1.9</td>
<td>95.6</td>
<td>2.5</td>
<td>141</td>
</tr>
<tr>
<td>string-validate-input</td>
<td>1,511</td>
<td>28</td>
<td>1.4</td>
<td>96.0</td>
<td>2.6</td>
<td>261</td>
</tr>
</tbody>
</table>

Figure 4. Pause time ratios of concurrent vs. sequential JITs.


Figure 5. Execution time improvements. Average time is break down in compilation, native code, and interpretation.


Exploring Practical Benefits of Asymmetric Multicore Processors

Jon Hourd, Chaofei Fan, Jiasi Zeng, Qiang(Scott) Zhang
Micah J Best, Alexandra Fedorova, Craig Mustard
{jlh4, cfa18, jza48, qsz, mbest, fedorova, cam14}@sfu.ca
Simon Fraser University
Vancouver Canada

Abstract—Asymmetric multicore processors (AMP) are built of cores that expose the same ISA but differ in performance, complexity, and power consumption. A typical AMP might consist of a plenty of slow, small and simple cores and a handful of fast, large and complex cores. AMPs have been proposed as a more energy efficient alternative to symmetric multicore processors. They are particularly interesting in their potential to mitigate Amdahl’s law for parallel program with sequential phases. While a parallel phase of the code runs on plentiful slow cores enjoying low energy per instruction, the sequential phase can run on the fast core, enjoying high single-thread performance of that core. As a result, performance per unit of energy is maximized. In this paper we evaluate the effects of accelerating sequential phases of parallel applications on an AMP. Using a synthetic workload generator and an efficient asymmetry-aware user-level scheduler, we explore how the workload’s properties determine the speedup that the workload will experience on an AMP system. Such an evaluation has been performed before only analytically; experimental studies have been limited to a small number of workloads. Our study is the first to experimentally explore benefits on AMP systems for a wide range of workloads.

I. INTRODUCTION

Asymmetric multicore processors consist of several cores exposing a single ISA but varying in performance [1], [4], [5], [6], [10], [11]. AMP systems are envisioned to be built of many simple slow cores and a few fast and powerful cores. Faster cores are more expensive in terms of power and chip area than slow cores, but at the same time they can offer better performance to sequential workloads that cannot take advantage of many slow cores. AMP systems have been proposed as a more energy-efficient alternative to symmetric multicore processors (SMP) for workloads with mixed parallelism. Workloads that consist of both sequential and parallel code can benefit from AMPs. Parallel code can be assigned to run on plentiful slow cores,享受 low energy per instruction, while sequential code can be assigned to run on fast cores, using more energy per instruction but enjoying much better performance than if they were assigned to slow cores.

In fact, recent work from Intel demonstrated performance gains of up to 50% on AMPs relative to SMPs that used the same amount of power [1]. Recent work by Hill and Marty [3] concluded that AMPs can offer performance significantly better than SMPs for applications whose sequential region is as small as 5%. Unfortunately, prior work evaluating the potential of AMP processors focused either on a small set of applications [1] or performed a purely analytical evaluation [3]. The question of how performance improvements derived from AMP architectures are determined by the properties of the workloads in real experimental conditions has not been fully addressed. Our work addresses this question.

We have created a synthetic workload generator that produces workloads with varying degrees of parallelism and varying patterns and durations of sequential phases. We also developed a user-level scheduler inside Cascade that is aware of the underlying system’s asymmetry and the parallel-to-sequential phase changes in the application. The scheduler assigns the sequential phases to the fast core while letting the parallel phases run on slow cores. As an experimental platform we use a 16-core AMD Opteron system where the cores can be configured to run at varying speeds using Dynamic Voltage and Frequency Scaling (DVFS).

While theoretical analysis of AMP systems indicated their promising potential, these benefits may not necessarily translate to real workloads due to the overhead of thread migrations. A thread must be migrated from the slow to the fast core when the workload enters a sequential phase. The migration overhead has two components: the overhead of rescheduling the thread on a new core and the overhead associated with the loss of cache state accumulated on the core where the thread ran before the migration. In our experiments we attempt
to capture both effects. We use the actual user-level scheduler that migrates the application’s thread to the fast core upon detecting a sequential phase, and we vary the frequency of parallel/sequential phase changes to gauge the effect of migration frequency on performance. We use workloads with various memory working set sizes and access patterns to capture the effects on caching. Although the caching effect has not been evaluated comprehensively (this is a goal for future work), our chosen workloads were constructed to resemble the properties of real applications. For the workloads used in our experiments, our results indicate that AMP systems deliver the expected theoretical potential, with the exception of workloads that exhibit very frequent switches between sequential and parallel phases.

The rest of this paper is organized as follows: Section 2 introduces the synthetic workload generator. Section 3 discusses theoretical analysis. Section 4 describes the experiment setup. Section 5 presents the experiment results.

II. THE SYNTHETIC WORKLOAD GENERATOR

To generate the workloads for our study, we used the Cascade parallel programming framework [2]. Cascade is a new parallel programming framework for complex systems. With Cascade, the programmer explicitly structures her C++ program as a collection of independent units of computation, or tasks. Cascade allows users to create graphs of computation tasks that are then scheduled and executed on a CPU by the Cascade runtime system. Figure 1 depicts a structure typical of the Cascade program we created for our experiments. The boxes represent the tasks (computational kernels), arrows represent dependencies. For instance, arrows going from tasks B, C, and D to task E indicate that task E may not run until tasks B, C, and D have completed. We use the graph structure depicted in Figure 1 to generate the workloads for our study. In particular, we focus on two aspects of the program: the structure of the graph and the type of computation performed by the tasks. All graphs start with a single task (A) to simulate a sequential phase. Once A finishes, several tasks start simultaneously (B, C, and D) to simulate a parallel phase. B, C, and D perform the same work so that they start and end at roughly the same time. Once B, C, and D finish, the next sequential phase (E) is executed. The last phase of all graphs is a sequential phase (I).

While the structures of our generated graphs are similar to the graph shown in Figure 1, they vary as follows:

1. The number of sequential phases can be varied according to the desired phase change frequency. The number of parallel phases is one fewer than the number of sequential phases.
2. The number of parallel tasks in each parallel phase can also be varied. For our purpose all parallel phases have the same number of parallel tasks.
3. The total computational workload of the entire graph can be precisely specified.
4. We can also specify the percentage of code executed in sequential phases.

Once a percentage of code executed by sequential phases is specified, the corresponding amount of the total workload is distributed equally to each sequential task so that the execution time for each sequential phase is roughly the same. The same method is applied to parallel phases so that all parallel computational tasks (e.g., B, C, D, F, G, and H in Figure 1) have roughly the same execution time.

In our initial experiments, all computational tasks execute an identical C++ function that consists of four algorithms, each taking roughly the same time to complete: (1) $I_c$, a CPU-intensive integer based pseudo-LZW algorithm; (2) $I_s$, a CPU-intensive integer based memory array shuffle algorithm; (3) $F_m$, a floating point Mandelbrot fractal generating algorithm (also CPU-intensive); (4) $F_r$, a memory-bound floating point matrix row reduction algorithm.

III. THEORETICAL ANALYSIS

Amdahl’s Law states that the speedup is the original execution time divided by the enhanced execution time. Following the method used by Hill and Marty [3], we use Amdahl’s Law to obtain a formula to predict a program’s performance speedup when its serial and parallel portions and processor performance are known:

$$\text{ExecutionTime} = \frac{f}{\text{perf}(s)} + \frac{(1-f)}{\text{perf}(p) \times x}$$
\[
\text{Speedup} = \frac{\text{ExecutionTime(Original)}}{\text{ExecutionTime(Enhanced)}}
\]

where \( f \) is the percent of code in sequential phases, \( \text{perf}(s) \) is the performance of serial core with frequency \( s \), \( \text{perf}(p) \) is the performance of parallel cores with frequency \( p \), \( x \) is the number of cores used in parallel phase. \( \text{perf}(x) \) is a function that predicts the performance of a core with frequency \( x \). For simplicity, we assume that it is proportional to the frequency. This formula assumes that parallel portions are entirely parallelizable and that there is no switching overhead. Both of these assumptions are to simplify the model and not necessarily expected to hold in a practice.

Using this formula, we generate the expected speedup of parallel applications on three systems: (1) SMP_16: a symmetric multicore system with 16 cores, (2) SMP_4: a symmetric multicore system with four cores, where each core runs at 2 times the frequency of each core in SMP_16, and (3) AMP_13: an asymmetric multicore system consisting of one “fast” core (of the speed similar to cores on the AMP_4 system) and 12 “slow” cores (of the speed similar to cores on the SMP_16 system).

The system configurations were constructed to have roughly the same power budget. The power requirements of a processing unit are generally accepted to be a function of the frequency of operation \cite{1}. For a doubling of clock speed, a corresponding quadrupling in power consumption is expected \cite{3}. Thus, a processor running at frequency \( x \) will consume four times less power than the processor running at frequency \( 2x \). Therefore, one core running at speed \( 2x \) is power-equivalent to four cores running at speed \( x \). As such, the three systems shown above will consume roughly the same power.

Figure 2 shows that using our execution time formula, we determine that the AMP system will outperform the SMP_4 system for all but completely sequential programs and it will outperform the SMP_16 system for programs with sequential region greater than 4%.

The results presented in Figure 2 are theoretical and they mimic those reported earlier by Hill and Marty \cite{3}. In the following sections we present the experimental results to evaluate how close they are to these theoretical predictions.

**IV. EXPERIMENTAL SETUP**

**A. Experiment Platform**

We used a Dell-Poweredge-R905 as our experimental system. The machine has 4 chips (AMD Opteron 8356 Barcelona) with 4 cores per chip. Each core has a private 256KB L2 cache and 2MB L3 victim cache that is shared among cores on the same chip. Our system is equipped with 64GB of 667MHz DDR, and it runs Linux 2.6.25 kernel with the Gentoo distribution.

This system supports DVFS for frequency scaling on a per core basis. The available frequency of AMD Opteron 8356 is from 1.15GHz to 2.3GHz. By varying the core frequency and turning off unused cores, we created three configurations with the same power budget as shown in Table 1.

<table>
<thead>
<tr>
<th>Number of Cores</th>
<th>Frequencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMP_4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>4×2.3GHz</td>
</tr>
<tr>
<td>SMP_16</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>16×1.15GHz</td>
</tr>
<tr>
<td>AMP_13</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>1×2.3GHz</td>
</tr>
<tr>
<td></td>
<td>+ 12×1.15GHz</td>
</tr>
</tbody>
</table>

**TABLE I EXPERIMENTAL CONFIGURATION**

Our user-level scheduler assigns tasks (recall Figure 1) to threads at runtime. Upon initialization, the scheduler creates as many threads as there are cores and binds each thread to a core. When the task graph begins to run, tasks are assigned to threads. On symmetric configurations, scheduling is purely demand-driven: a newly available task is assigned to any free thread. On an AMP configuration, one thread is bound to the fast core and is called the fast thread; other threads are bound to slow cores and are called slow threads. When there is only one runnable task, Cascade assigns it to the fast thread. When there are multiple runnable tasks, they are assigned to slow threads. Although this scheduling policy does not utilize the fast core during the parallel phase, it is a reasonable approximation of a realistic AMP-aware scheduler. Figure 3 demonstrates one example of workload assignment during runtime: each thread is assigned to one core; sequential parts are always executed on thread 0, which is a fast thread.
while parallel parts are executed in parallel on other slow threads.

B. Workloads

We varied several parameters in our graph generator to generate a task graph that could capture major characterizations of real applications.

**Iterations**: This parameter represents the number of computational tasks of the whole graph, in other words, the execution time of the program. By setting $iterations = 1$, there will be $10^7$ computational tasks, each consisting of four C++ algorithms.

**Phase change**: This parameter defines how many sequential and parallel phases there are in the graph representing the computation. A graph always starts and ends with a sequential phase. By setting $phase \_change = 2$, there will be two sequential phases and one parallel phase.

**Parallel width**: This parameter defines how many parallel tasks are there in the parallel phase. By setting $parallel \_width = 4$, there will be four parallel tasks in the parallel phase.

**Sequential percentage**: This parameter defines the portion of code that is sequential. By setting $sequential \_percentage = 50$, 50% of the graph will be executed in sequential phases and 50% will be in the remaining parallel phases.

Setting $iterations = 10$, $phase \_change = 4$, $parallel \_width = 3$, $sequential \_percentage = 20$ will produce the same graph as in Figure 1. Each sequential task will have $\frac{10 \times 10^7 \times 20%}{3}$ algorithmic iterations, while each parallel task will have $\frac{10 \times 10^7 \times 80%}{2 \times 3}$ algorithmic iterations.

For each experimental configuration, we configure the graph such that the parallel width is equal to the number of cores available in the parallel phase, which corresponds to the way users often configure the threading level in their applications.

V. Experimental Results

In the first experiment we set the number of iterations to 100 and the $phase \_changes$ parameter to 5. Figure 4 shows the speedup for workloads with sequential percentage ranging from 0% to 100% (with 5% increment) on SMP_16 and AMP_13 relative to SMP_4. Comparing these results to the theoretical results in Figure 2 we see that the actual experimental results closely follow the theoretical results with all data on average within 1% range of the analytically derived values. When the workload is purely parallel, SMP_16 outperforms SMP_4 by a factor of 2 approximately, as seen in the theoretical graph. With the increase of sequential code fraction, the fast core in SMP_4 begins to show its power: SMP_4 outperforms SMP_16 beyond the sequential fraction of 15%. Most importantly, AMP_13 almost always outperforms SMP_4 and SMP_16. This is simply because the single fast core speeds up the sequential phases while the remaining slow cores are able to efficiently execute the parallel phases. Only when the sequential code fraction is below 5% does SMP_16 outperform AMP_13 since SMP_16 is better able to utilize a large number of cores for highly parallel workloads.
To experiment with shorter tasks (and thus more frequent phase changes), we reduced the number of total iterations by setting $\text{iterations} = 10$ and left the number of phase changes set to five. In this case, the pattern of task graph is the same as in the previous test and the only difference is the length of each task (1/10 of that in previous task graph). The results shown in Figure 5 demonstrate that when the tasks are shorter, the effect of the overhead comes into play. The speedup of AMP\_13 is on average 3.5% within the range of theoretical results, and the speedup for SMP\_16 is on average within 1.9% of theoretical results.

To further investigate the effect of phase changes, we measured the slowdown for each configuration when phase change increased from five to fifteen while keeping the number of iterations equal to ten (Figure 7). SMP\_16 and AMP\_13 suffered more performance degradation than SMP\_4 and the slowdown appeared to decrease as sequential percentage increased. This indicates that scheduling overhead was the reason behind poor performance. When switching between parallel and sequential phases, there is scheduling overhead associated with updating the scheduler’s internal queues, handling interprocessor interrupts as well as migrating the thread’s state architectural state to the fast core. Since the synthetic workloads on SMP\_16 and AMP\_13 have a greater parallel width than SMP\_4, the overhead of task assignment was larger and this caused a greater slowdown. As the sequential code fraction increases, the size of each sequential task becomes larger, and so the overhead of scheduling is relatively small. In prior work we evaluated the efficiency of the Cascade scheduler [2] and found that it was rather efficient, so we conjecture that the overhead is not due to the implementation of the scheduler, but is inherent to any system that would be required to switch threads at such a high frequency.
VI. CONCLUSIONS AND FUTURE WORK

In this paper we have evaluated the practical potential of AMP processors by analyzing how the performance benefits delivered by these systems are determined by the properties of the workload. We create synthetic workloads to simulate real applications and use DVFS technique to model AMP processors on conventional multicore processors. Our results demonstrate that AMP systems can deliver their theoretically predicted performance potential unless the changes between parallel and sequential phases are extremely frequent.

As part of future work we would like to further investigate the overhead behind thread migrations, perhaps deriving an analytical model for this overhead based on the architectural parameters of the system and the properties of the workload. The effects of migration on cache performance in the context of AMP systems must also be investigated further.

Our synthetic workloads aim at simulating parallel behavior of applications with a fine granularity. But assumptions about the synthetic workloads, i.e., computing-bound with consistent pattern, may not be a good reflection of real applications. More diversified workloads with various parallel width and percentage should be tested more systematically. To improve the reliability of our synthetic workload generator, further investigation on the behavior of real applications will also be needed.

Scheduling is another future area for investigation. Since we didn’t fully utilize fast cores, migrating parallel tasks to fast cores when they are idle may achieve significantly better performance in parallel phases. To further optimize the performance of parallel phase, more sophisticated scheduling algorithms [11] may be introduced. While several schedulers for AMP systems have proposed in prior work [5], [7], [8], they have primarily addressed the ability of these systems to address instruction-level parallelism in the workload. Only one work addressed the design of an asymmetry-aware operating system scheduler that caters to the changes in parallel/sequential phases of the applications [9]. It would be interesting to validate our results with that scheduler, and to evaluate the difference in the overhead resulting from the user-level and kernel-level implementations.

REFERENCES


60
Hybrid Operand Communication for Dataflow Processors

Dong Li  Behnam Robatmili  Sibi Govindan  Doug Burger  Steve Keckler
University of Texas at Austin, Computer Science Department
{dongli, beroy, sibi, dburger, skeckler}@cs.utexas.edu

Abstract

One way to exploit more ILP and improve the performance of a single threaded application is to speculatively execute multiple code regions on multiple cores. In such a system, communication of operands among in-flight instructions can be power intensive, especially in superscalar processors where all result tags are broadcast to a small number of consumers through a multi-entry CAM. Token-based point-to-point communication of operands in dataflow architectures is highly efficient when each produced token has only one consumer, but inefficient when there are many consumers due to the construction of software fanout trees. This paper evaluates a compiler-assisted hybrid instruction communication model that combines token instruction communication with statically assigned broadcast tags. Each fixed-size block of code is given a small number of architectural broadcast identifiers, which the compiler can assign to producers that have many consumers. Producers with few consumers rely on point-to-point communication through tokens. Selecting the mechanism statically by the compiler relieves the hardware from categorizing instructions at runtime. At the same time, the compiler can categorize instructions better than dynamic selection does because the compiler analyzes a larger range of instructions. The results show that this compiler-assisted hybrid token/broadcast model requires only eight architectural broadcasts per block, enabling highly efficient CAMs. This hybrid model reduces instruction communication energy by 28% compared to a strictly token-based dataflow model (and by over 2.7X compared to a hybrid model without compiler support), while simultaneously increasing performance by 8% on average across the SPECINT and EEMBC benchmarks, running as single threads on 16 composed, dual-issue EDGE cores.

1 Introduction

Improvement of the single thread performance highly relies on the amount of ILP that can be exploited. Conventional superscalar processor can not scale well because of the complexity and power consumption of large-issue-width and huge-instruction-window processor. One way to solve this problem is to partition the code into regions, and execute multiple regions speculatively on multiple cores. This method increases both the issue width and the instruction window size dramatically, thus more ILP can be extracted. In such a system, communicating operands between instructions is one of the performance bottlenecks. In addition, communicating operands between instructions is a major source of energy consumption in modern processors. A wide variety of operand communication mechanisms have been employed by different architectures. For example in superscalar processors, to wake up all consumer instructions of a completing instruction, physical register tags are broadcast to power-hungry Content Addressable Memories (CAMs), and operands are obtained from a complex bypass network or by a register file with many ports.

A mechanism commonly used for operand communication in dataflow architectures is point-to-point communication, which we will refer to as “tokens” in this paper. Tokens are highly efficient when a producing instruction has a single consumer; the operand is directly routed to the consumer, often just requiring a random-access write into the consumer’s reservation station. If the producer has many consumers, however, dataflow implementations typically build an inefficient software fanout tree of operand-propagating instructions (that we call move instructions).

These two mechanisms are efficient under different scenarios: broadcasts should be used when there are many consumers currently in flight (meaning they are in the instruction window), tokens should be used when there are few consumers, and registers should be used to hold values when the consumers are not yet present in the instruction window.

Several approaches [3, 4, 6, 9, 10] have proposed hybrid schemes which dynamically combine broadcasts and tokens to reduce the energy consumed by the operand bypass. These approaches achieve significant energy consumption compared to superscalar architectures. In addition, because of their dynamic nature, these approaches can adapt to the window size and program characteristics without changing the ISA. On the other hand, these approaches use some additional hardware structures and keep track of various mechanisms at runtime.

The best communication mechanism for an instruction depends on the dependence patterns between that instruction and the group of consumer instructions currently in the instruction window. This information can be calculated statically at com-
pile time and conveyed to the microarchitecture through unused bit in the ISA.

Using this observation, this paper evaluates a compiler-assisted hybrid instruction communication mechanism that augments a token-based instruction communication model with a small number of architecturally exposed broadcasts within the instruction window. A narrow CAM allows high-fanout instructions to send their operands to their multiple consumers, but only unissued instructions waiting for an architecturally specified broadcast actually perform the CAM matches. The other instructions in the instruction window do not participate in the tag matching, thus saving energy. All other instructions, which have low-fanout, rely on the point-to-point token communication model. The determination of which instructions use tokens and which use broadcasts is made statically by the compiler and is communicated to the hardware via the ISA. As a result, this method does not require instruction dependence detection and instruction categorization at runtime. However, this approach requires ISA support and may not automatically adapt to microarchitectural components such as window size.

Our experimental vehicle is TFlex [7], a composable multicore processor, which implements an EDGE ISA [12]. We extend the existing token-based communication mechanism of TFlex with this hybrid approach and evaluate the benefits both in terms of performance and energy. On a composed 16-core TFlex system (running in the single-threaded mode), the proposed compiler-assisted hybrid shows a modest performance boost and significant energy savings over the token-only baseline (which has no static broadcast support). Across the SPECINT2K and EEMBC benchmarks, using only eight architectural broadcasts per block, performance increases by 8% on average. Energy savings are more significant, however, with a 28% lower energy consumption in operand communication compared to the token-only baseline. This energy saving translates to a factor of 2.7 lower than a similar hybrid policy implementation without full compiler support.

2 System Overview

TFlex is a composable lightweight processor in which all microarchitectural structures, including the register file, instruction window, predictors, and L1 caches are distributed across a set of cores [7]. Distributed protocols implement instruction fetch, execute, commit, and misprediction recovery without centralized logic.

TFlex implements an EDGE ISA which supports block-atomic execution. Thus, fetch, completion, and commit protocols operate on blocks rather than individual instructions. The compiler [14] breaks the program into single-entry, predicated blocks of instructions. At runtime, each block is allocated to one core and is fetched into the instruction queue of that core. The union of all blocks running simultaneously on distributed cores constructs a large contiguous window of instructions. Inter-block communication for long dependences occur through distributed register files using a lightweight communication network [7].

Register and memory communication is used for inter-block communication. Within blocks, instructions run in dataflow order. A point-to-point bypass network performs producer-consumer direct communication using tokens. When an instruction executes, the address of its target is used to directly index the instruction queue.

In this dataflow representation, each instruction explicitly encodes its target instructions in the same block using the offsets of the target instructions from the beginning of the block. For each instruction, its offset from the beginning of its block is the instruction ID of that instruction. An example of the initial intermediate code and its converted dataflow representation are shown in Figures 1(a) and 1(b), respectively. Instruction i adds values a and b and sends the output to operand\(_1\) and operand\(_2\) of instructions j and k, respectively. Instruction j subtracts that value from another value d, and sends the output to operand\(_2\) of instruction k. Finally, instruction k stores the value computed by instruction i at the address computed by instruction j.

The aforesaid dataflow encoding eliminates the need for an operand broadcast network. When an instruction executes, the address of its target is used to directly index the instruction queue. Because of this direct point-to-point communication, the instruction queue has a simple 128-entry SRAM structure instead of large, power-hungry CAM structures used for instruction queues in superscalar processors. Figure 2 illustrates instruction encoding used by the EDGE ISA. Because the maximum block size is 128 instructions, each instruction ID in the target field of a instruction requires seven bits. The target field also requires two bits to encode the type of the target because each instruction can have three possible inputs including operand\(_1\), operand\(_2\) and predicate.

![Figure 1: A baseline code example.](image)

![Figure 2: TFlex Instruction Encoding.](image)
Although token based point-to-point communication is very power-efficient for low-fanout instructions but similar to other dataflow machines, it may not be very performance-efficient when running high-fanout instructions since the token needs to travel through the fanout tree to reach all the targets.

3 Hybrid Operand Communication Mechanism

This section proposes an approach for hybrid operand communication with compiler assistance. The goal of the new approach is to achieve higher performance and energy efficiency by allowing the compiler to choose best communication mechanism for each instruction during the compilation phase. The section discusses the implementation of the new approach, which consists of three parts: (1) heuristics to decide the operand communication mechanism during compilation; (2) ISA support for encoding the compiler decision, broadcast tags or point-to-point tokens; and (3) microarchitectural support for the hybrid communication mechanism. This section concludes with a discussion of design parameters and power trade-offs and performance implications of the proposed approach.

3.1 Overview

Since each block of code is mapped to one core, the hybrid mechanism explained in this section is used to optimize the communication between instructions running within each core. This means that no point-to-point or broadcast operand crosses core boundaries. For cross-core (i.e. cross-block) communication, TFlex uses registers and memory [11], which are beyond the scope of this article. Of course extending hybrid communication to cross-core communication is an interesting area and can be considered future work of this work.

Different from dynamic hybrid models, the compiler-assisted hybrid model relies on the ISA to convey information about point-to-point and broadcast instructions into the microarchitecture. The involvement of the ISA leads provides some opportunities for the compiler while causing some challenges at the same time. Assuming a fixed instruction size, using tokens can lead to construction of fanout move trees and manifests itself at runtime in form of extra power consumption and execution delay. On the other hand, categorizing many instructions as broadcast instructions requires the hardware to use a wide CAM in the broadcast bypass network, which can become a major energy bottleneck. The main role of the compiler is to pick the right mixture of the tokens and broadcast such that the total energy consumed by the move trees and the broadcast network becomes as small as possible. In addition, this mixture should guarantee an operand delivery delay close to the one achieved using the fastest operand delivery method (i.e. the broadcast network). One challenge, however, is to find enough number of unused bits in the ISA to encode broadcast data and convey it to the microarchitecture.

3.2 Broadcast Tag Assignment and Instruction Encoding

One primary step in designing the hybrid communication model is to find a method to distinguish between low- and high-fanout instructions. In the compiler-assisted hybrid communication approach, the compiler detects the high-fanout instructions and encodes information about their targets via the ISA. In this subsection, we first give an overview of the phases of the TFlex compiler. Then we explain the algorithm for detecting high-fanout instructions and the encoding information inserted by the compiled in the broadcast sender and receiver instructions.

The original TFlex compiler [14] generates blocks containing instructions in dataflow format by combining basic blocks using if-conversion, predication, unrolling, tail duplication, and head duplication. In each block, all control dependencies are converted to data dependencies using predicate instructions. As a result, all intra-block dependencies are data dependencies, and each instruction directly specifies its consumers using a 7-bit instruction identifier. Each instruction can encode up to two target instructions in the same block. During block formation, the compiler identifies and marks the instructions that have more than two targets. Later, the compiler adds move fanout trees for those high-fanout instructions during the code generation phase.

The modified compiler for the hybrid model needs to accomplish two additional tasks, selecting the instructions to perform the broadcast, and assigning static broadcast tags to the selected instructions. The compiler lists all instructions with more than one target and sorts them based on the number of targets. Starting from the beginning of the list, the compilers assigns each instruction in the list a tag called broadcast identifier (BCID) out of a fixed number of BCIDs. For producers and consumers send or receive BCIDs needs to be encoded inside each instruction. Therefore, the total number of available BCIDs is restricted by the number of unused bits available in the ISA. Assuming there are at most MaxBCID BCIDs available, then the first MaxBCID high-fanout instructions in the list are assigned a BCID.

After the broadcast sender instructions are detected and BCIDs are assigned, the compiler encodes the broadcast information inside the sender and receiver instructions. Figure 3 illustrates the ISA extension using a sample encoding for MaxBCID equal to eight. Each sender contains a broadcast bit, bit B in the figure, enabling broadcast send for that instruction. The compiler also encodes the BCID of each sender inside both the sender and the receiver instructions of that sender. For the sender, the target bits are replaced by the three send BCID bits and two broadcast type bits. Each receiver can encode up
to two BCIDs with six bits, and so it can receive its operands from two possible senders. Although this encoding uses two BCIDs for each receiver instruction, the statistics show that a very small percentage of instructions may receive broadcasts from two senders. For the other instructions that are not receiver of any broadcast instructions, the compiler assigns the receive BCIDs to 0, which disables the broadcast receiving mechanism for those instructions.

Figure 4 illustrates a sample program (except for stores, the first operand of each instruction is the destination), its equivalent dataflow representation, and its equivalent hybrid token/broadcast representation generated by the modified compiler. In the original dataflow shown code in Figure 4(b), instruction \(i_1\) can only encode two of its three targets. Therefore, the compiler inserts a move instruction, instruction \(i_{1a}\), to generate the fanout tree for that instruction. For the hybrid communication model shown in Figure 4(c), the compiler assigns a BCID (BCID of 1 in this example) to \(i_1\), the instruction with high fanout, and eliminates the move instruction. The compiler also encodes the broadcast information into the \(i_1\) and its consuming instructions (instructions \(i_2, i_3\) and \(i_4\)). The compiler use tokens for the remaining low-fanout instructions. For example, instruction \(i_3\) has only one target (instruction \(i_2\)) so \(i_3\) still uses token-based communication. In the next subsection, we explain how these fields are used during the instruction execution and what additional optimizations are possible in the proposed hardware implementation.

### 3.3 Microarchitectural Support

To implement the broadcast communication mechanism in the TFlex substrate, a small CAM array is used to store the receive BCIDs of broadcast receiver instructions in the instruction queue. When instructions are fetched, the receive BCIDs are stored in a CAM array called BCCAM. Figure 5 illustrates the instruction queue of a single TFlex core when running the broadcast instruction \(i_1\) in the sample code shown in Figure 4(c). When the broadcast instruction executes the broadcast signal, bit \(B\) in Figure 3 is detected, then the sender BCID (value 001 in this example) is sent to be compared against all the potential broadcast receiver instructions. Notice that only a subset of instructions in the instruction queue are broadcast receivers and the rest of them need no BCID comparison. Among all receiving instructions, the tag comparison will match only for the CAM entries corresponding to the receivers of the current broadcast sender (instructions \(i_2, i_3\) and \(i_4\) in this example). Each matching entry of the BCCAM will generate a write-enable signal to enable a write to the operand of the corresponding receiver instruction in the RAM-based instruction queue. The broadcast type field of the sender instruction (operand1 in this example) is used to select the column corresponding to the receivers’ operand, and finally all the receiver operands of the selected type are written simultaneously into the instruction window.

It is worth noting that tag delivery and operand delivery do not happen at the same cycle. Similar to superscalar operand delivery networks, the tag of the executing sender instruction is first delivered at the right time, which is one cycle before instruction execution completes. At the next cycle, when instruction result is ready, the result of the instruction is written simultaneously into all waiting operands in the instruction window.

Figure 6 illustrates a sample circuit implementation for the compare logic in each BCCAM entry. The CAM tag size is three bits which represents a \(MaxBCID\) parameter of eight. In this circuit, the compare logic is disabled if one of the following conditions is true:

- If the instruction corresponding to the CAM entry has been previously issued.
If the receiver BCID of the instruction corresponding to the CAM entry is not valid, which means the instruction is not a broadcast receiver. For example instruction $i_5$ in the example shown in Figures 5 and 4.

If the executed instruction is not a broadcast sender.

This hybrid broadcast model is more energy-efficient than the instruction communication model in superscalar processors for several reasons. First, because of the $MaxBCID$ limit on the maximum number of broadcast senders, the size of the broadcast tag, which equals to the width of the CAM, could be reduced from $\log(InstructionQueueSize)$ to $\log(MaxBCID)$. A broadcast consumes significantly less energy because it drives a much narrower CAM structure. Second, only a small portion of bypasses are selected to be broadcast and the majority of them use the token mechanism, since the compiler only selects a portion of instructions to perform broadcasts. Third, only a portion of instructions in the instruction queue are broadcast receivers and perform BCID comparison during each broadcast. Both of these design aspects are controlled by the $MaxBCID$ parameter. This parameter directly controls the total number of broadcast senders in the block. On the other hand, as we increase the $MaxBCID$ parameter, the number of active broadcast targets is likely to increase, but the average number of broadcast targets per broadcast is likely to shrink.

Different values of $MaxBCID$ represent different design points in a hybrid broadcast/token communication mechanism. $MaxBCID$ of zero represents a pure token-based communication mechanism and fanout trees using move instructions. $MaxBCID$ of 128 means every instruction with fanout larger than one will be a broadcast sender. In other words, the compiler does not analyze any global fanout distribution to select right communication mechanism for each instruction. Instead, all fanout instruction in each block use broadcast operation. This model is close to a TFlex implementation of a dynamic hybrid point-to-point/broadcast communication model [6]. It is worth mentioning that even with $MaxBCID$ equal to 128, there are still many instructions with just one target and those instructions still use token-based communication. As we vary the $MaxBCID$ form zero to 128, more fanout trees are eliminated, and more broadcasts are added to the system. By choosing an appropriate value for this parameter, the compiler is able to minimize total power consumed by fanout trees and broadcasts while achieving a decent speedup in performance as a result of using broadcasts for high-fanout instructions.

Figure 5: Execution of a broadcast instruction in the IQ.

Figure 6: Compare logic of BC CAM entries.

4 Evaluation and Results

In this section we evaluate the energy consumption and performance of the compiler-assisted hybrid operand communication model. We first describe the experimental methodology followed by statistics about the distribution of broadcast producers and consumers. This distribution data will indicate the fraction of all instructions in the window that have a high fan-out value. The distribution also suggests the minimum $MaxBCID$ and $BCCAM$ bit-width needed for assigning broadcast tags to all of those high-fanout instructions. Then, we report performance results and power breakdown of fanout trees or broadcast instructions for different $MaxBCID$ values. These results show
that by intelligently picking a subset of high-fanout instructions for broadcast, the compiler is able to reduce the total power significantly without losing much performance than if it picked all high-fanout instructions.

The results show that this compiler-assisted hybrid model consumes significantly lower power than the pure broadcast mechanism used by superscalar processors. With this hybrid communication model, we explore the full design space ranging from a very power efficient token-based dataflow communication model to a high-performance broadcast model similar to that used in superscalar machines. The results show that the compiler assistance is more reliable than dynamically choosing the right operand communication mechanism for each instruction. Given the compiler assistance, not only are we able to achieve a higher energy efficiency than pure dataflow, but at the same time we are also able to achieve better performance in this design space.

4.1 Methodology

We augment the TFlex simulator [7] with the support for the hybrid communication model explained in the previous section. In addition we modify the TFlex compiler to detect high-fanout instructions and to encode broadcast identifiers in those instructions and their targets. Each TFlex core is a dual-issue, out-of-order core with a 128-instruction window. Table 1 shows the microarchitectural parameters of each TFlex core. The energy consumed by move instructions during the dispatch and issue phases is already incorporated into original TFlex power models [7]. We augment the baseline TFlex models with the power consumed in the BCCAM entries, modeled using CACTI 4.1 [5], when tag comparisons are made during a broadcast.

The results presented in this section are generated using runs on several SPEC INT [2] and EEMBC [1] benchmarks running on 16 TFlex cores. We use seven integer SPEC benchmarks with the reference (large) dataset simulated with single SimPoints [13]. The SPEC FP benchmarks achieve very high performance when running on TFlex, so the speedups are less important and interesting to this work. We also use 28 EEMBC benchmarks which are small kernels with various characteristics. We test each benchmark varying the MaxBCID from 0 to 128 to measure the effect of that parameter on different aspects of the design.

### Table 1: Single Core TFlex Microarchitecture Parameters [7]

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Supply</td>
<td>Partitioned 8KB I-cache (1-cycle hit); Local/Gshare Tournament predictor (8K+256 bits, 3 cycle latency) with speculative updates; Num. entries: Local: 64(L1) + 128(L2), Global: 512, Choice: 512, RAS: 16, CTB: 16, BTB: 128, Byte: 256.</td>
</tr>
<tr>
<td>Execution</td>
<td>Out-of-order execution, RAM structured 128-entry issue window, dual-issue (up to two INT and one FP) or single issue.</td>
</tr>
<tr>
<td>Data Supply</td>
<td>Partitioned 8KB D-cache (2-cycle hit, 2-way set-associative, 1-read port and 1-write port); 44-entry LSQ bank; 4MB decoupled S-NUCA L2 cache [8] (8-way set-associative, LRU-replacement); L2-hit latency varies from 5 cycles to 27 cycles depending on memory address; average (unloaded) main memory latency is 150 cycles.</td>
</tr>
<tr>
<td>Simulation</td>
<td>Execution-driven simulator validated to be within 7% of real system measurement</td>
</tr>
</tbody>
</table>

4.2 Distribution of Producers and Operands

Figure 7 shows the average cumulative distribution of the number of producers and the operands for different fanout values for SPEC INT benchmarks. The cumulative distribution of producers converges much faster that the one of operands does, which indicates a small percentage of producers corresponds to a large number of operands. For example, for fanouts larger than four, only 8% of producers produce 40% of all operands. It indicates that performing broadcasts on a small amount of producers could improve operand delivery for a large number of operands. The information shown in this graph is largely independent from the microarchitecture and reflects the operand communication behaviors of the programs. To choose the right mechanism for each producer, one also must consider the hardware implementation of each mechanism. This graph shows that 78% of all instructions have fanout equal or less than two.

For these instructions, given the TFlex microarchitecture, it is preferred to use efficient token-based communication. For the rest of instructions, finding the right breakdown of instructions between broadcasts and move trees also depends on the cost of each of these mechanisms.

Figure 8 shows the breakdown ratio of broadcast producers, instructions sending direct tokens, and the move instructions to all instructions for the SPEC benchmarks when using the compiler-assisted model proposed in this paper. The number of broadcast instructions (producers) increases dramatically for smaller MaxBCID values, but levels off as the MaxBCIDs parameter approaches 32. At the same time, the ratio of move instructions decreases from 35% to 5%. As a result, the total number of instructions drops to 79%. This observation indicates that the compiler can detect most of the high-fanout dependences inside a block and replace the software fanout tree by using only up to 32 broadcasts. The data shown in Figure 8 also indicates that even with the unlimited number of broadcasts, at most 25% of the instructions use broadcast communication and the rest of them use tokens for communicating. This is almost one fourth of the number of broadcasts used by a superscalar machine because in a superscalar machine all instructions must use the broadcast mechanism. Another observation is that the total number of instructions decreases 15% with only 8 broadcasts, which indicates that a small number of broadcasts could give us most of the benefits of unlimited broadcasts.
4.3 Energy Tradeoff

Figure 9 illustrates the energy breakdown into executed move and broadcast instructions for a variety of MaxBCID values on the SPEC benchmarks. The energy values are normalized to the total energy consumed by move instructions when instructions communicate only using tokens (MaxBCID = 0). When only using tokens, all energy overheads are caused by the move instructions. Allowing one or two broadcast instructions in each block, MaxBCIDs of 1 and 2, we observe a sharp reduction in the energy consumed by move instructions. As discussed in the previous section, the compiler chooses the instructions with highest fanout first when assigning BCIDs. Consequently, high number of move instructions are removed for small MaxBCIDs which results in significant reduction in the energy consumed by move instructions. For these MaxBCIDs values, the energy consumed by broadcast instructions is very low.

As we increase the total number of broadcast instructions, the energy consumed by broadcast instructions increases dramatically and fewer move instructions are removed. As a result, at some point, the broadcast energy becomes dominant. For high numbers of MaxBCID, the broadcast energy is orders of magnitude larger than the energy consumed by move instructions. The key observation in this graph is that for MaxBCID equal to 4 and 8, in which only 4 to 8 instruction broadcast in each block, the total energy consumed by moves and broadcast is minimum. For these MaxBCIDs, the total energy is about 28% lower than the energy consumed by a fully dataflow machine (MaxBCID = 0) and about 2.7x lower than when MaxBCID is equal to 128. These results show that the compiler is able to achieve a better trade-off in terms of power breakdown by selecting a critical subset of high-fanout instructions in each block. We also note that for MaxBCIDs larger than 32, the energy consumed by move instructions is at a minimum and does not change. In an ideal setup where the overhead of broadcast is ignored, these points give us the best possible energy savings. This energy is four time lower than the total energy consumed when using MaxBCID equal to 8, which is the point with the lowest total power. The energy breakdown chart for EEMBC benchmarks is similar to SPEC benchmarks except that MaxBCID of 4 results in lower total power consumption than MaxBCID of 8.

Figure 9 also shows the lower bound energy consumption values derived using an analytical model. This analytical model
gives us the best communication mechanism for each producer in an ideal environment. In order to choose the best communication mechanism for each instruction, the analytical model measures the energy consumption of a single move instruction and that of broadcast CAMs of different bit widths. The energy consumption of software fanout tree mainly comes from several operations, such as writing/reading move instructions in the instruction-queue, writing/reading operands in the operand buffer, generating control signals, driving the interconnection wires which includes the activities on the wire networks when fetching, decoding, executing of the move instruction and transmitting the operand. On the other side, the energy consumption of the broadcast operations mainly comes from driving the CAM structure, the tag-matching and writing the operands in the operand buffer. The energy consumed by each of these operations is modeled and evaluated with CACTI4.1 [5] and the power model in the TFloat simulator [7], and used by the analytical model. For a specific MaxBCID x, the analytical model estimates the lower bound of energy consumption of the hybrid communication model assuming an ideal situation in which that there are unlimited number of broadcast tags and each broadcast consumes as little energy as a broadcast using a CAM width logx. Based on this assumption, the analytical model finds the break even point between moves and broadcast instructions in which the total energy consumed by broadcasts is the same as the total energy consumed by moves.

As can be seen in Figure 9, for small or large values of MaxBCID, the real total power consumed by moves and broadcasts is significantly more than the ideal energy estimated by the analytical model. This difference seems to be minimum when MaxBCID equals 8, which the total consumed power is very close to the optimum power at this point. Table 2 reports the percentage of broadcast producer instructions for different BCIDs achieved using ideal analytical model and compiler-assisted approach. With small MaxBCIDs, the large difference between real energy and ideal energy is because there is not enough tags to encode more broadcasts. On the other hand, when using large MaxBCIDs the more than enough number of broadcasts are encoded, which increases the energy consumption. Finally, with MaxBCID of eight, the percentage of broadcast is very close to that achieved using the ideal analytical model.

We also measured the total energy consumption of the while processor (including SDRAMs and L2 caches) with variable MaxBCID. The compiler-assisted hybrid communication model achieves 6% and 10% total energy saving for SPEC INT and EEMBC benchmarks, respectively. The energy reduction mainly comes from two aspects: (1) replacing software fanout trees with broadcasts which reduces the energy of instruction communication; (2) reducing the total number of instructions, so there are fewer number of I-Cache access (and misses) and less overhead for executing the move instructions.

### 4.4 Performance Improvement

In terms of performance, full broadcast has the potential to achieve highest performance. The reasons are that there is only one cycle latency between the broadcast instructions with its consumers, while communicating the operands though move tree results in more than one cycle latency. However, large number of broadcast causes large amount of energy consumption.
There is an important tradeoff between the performance and the energy efficiency when using viable value of MaxBCID. This subsection evaluates the performance improvement for different parameters. The key observation from the evaluation is that 8 broadcasts per-block could be the best tradeoff between the performance and energy efficiency. It achieves most of the speedup reached by the unlimited broadcast, at the same time, it saves most of the energy as discussed in last subsection.

Figure 10 shows the average performance improvement over TFlex cores with no broadcast support (MaxBCID = 0) for the SPEC and EEMBC benchmarks. The average speedup reaches its maximum as MaxBCID reaches 32 and remains almost unchanged for larger values. As shown in Figure 8, with MaxBCID equal to 32, most of high-fanout instructions are encoded. The speedup achieved using MaxBCID of 32 is about 8% for SPEC benchmarks. Again, for the EEMBC benchmarks MaxBCID of 32 achieves very close to the best speedup, which is about 14%. On average, the EEMBC benchmarks gain higher speedup using the hybrid approach, which might be because of larger block sizes in EEMBC applications, which provide more opportunity for broadcast instructions. Most EEMBC benchmarks consist of parallel loops, whereas the SPEC benchmarks have a mixture of small function bodies and loops. In addition, the more complex control flow in SPEC benchmarks results in relatively smaller blocks.

Figure 11 shows the performance improvement over TFlex cores with no broadcast support (MaxBCID = 0) for individual SPEC benchmarks. The general trend for most benchmarks is similar. We do not include the individual EEMBC benchmarks here because we notice similar trends in EEMBC too. For gcc, the trend of speedups is not similar to other benchmarks for some MaxBCID values. We attribute this to the high misprediction rate in the memory dependence predictors used in the load/store queues.

Although MaxBCID of 32 achieves the highest speedup, but Figure 9 shows it may not be the most power-efficient design point compared to the power-efficiency of full dataflow communication. When designing for power-efficiency, one can choose MaxBCID of 8 to achieve the lowest total power, while still achieving a decent performance gain. Using MaxBCID of 8 the speedup achieved is about 5% and 10% for SPEC and
EEMBC benchmarks, respectively, and the power is reduced by 28%.

5 Conclusions

This paper proposes a compiler-assisted hybrid operand communication model. Instead of using dynamic hardware-based pointer chasing, this method relies on the compiler to categorize instructions for token or broadcast operations. In this model, the compiler took a simple approach: broadcasts were used for operands that had many consumers, and dataflow tokens were used for operands that had few consumers. The compiler can analyze the program in a bigger range to select the best operand communication mechanism for each instruction. At the same time, the block-atomic EDGE model made it simple to perform that analysis in the compiler, and allocate a number of architecturally exposed broadcasts to each instruction block. By limiting the number of broadcasts, the CAMs searching for broadcast IDs can be kept narrow, and only those instructions that have not yet issued and that actually need a broadcast operand need to be performing CAM matches. This approach is quite effective at reducing energy; with eight broadcast IDs per block, 28% of the instruction communication energy is eliminated by eliminating many move instructions (approximately 55% of them), and performance is improved by 8% on average due to lower issue contention, reduced critical path height, and fewer total blocks executed. In addition, the results show that the power savings achieved using this model are close to the minimum possible power savings using a near-ideal operand delivery model.

References

Hierarchical Control Prediction: Support for Aggressive Predication

Hadi Esmaeilzadeh
Department of Computer Sciences
The University of Texas at Austin
hadi@cs.utexas.edu

Doug Burger
Microsoft Research
One Microsoft Way, Redmond, WA 98052
dburger@microsoft.com

Abstract

Predication of control edges has the potential advantages of improving fetch bandwidth and reducing branch mispredictions. However, heavily predicated code in out-of-order processors can lose significant performance by deferring resolution of the predicates until they are executed, whereas in non-predicated code those control arcs would have remained as branches, and would be resolved immediately in the fetch stage when they are predicted. Although predicate prediction can address this problem, three problems arise when trying to predict aggressively predicated code that contains multi-path hyperblocks: (1) how to maintain a high bandwidth of branch prediction to keep the instruction window full without having the predicate predictions interfere and without increasing the branch mispredictions, (2) how to determine which predicates in the multi-path hyperblocks should be predicted, and (3) how to achieve high predicate prediction accuracies without centralizing all prediction information in a single location. To solve these problems, this paper proposes a speculation architecture called hierarchical control prediction (HCP). In HCP, the control flow speculation is partitioned into two levels. In parallel with the branch predictor, which identifies the coarse-grain execution path by predicting the next hyperblock entry, HPC identifies and predicts the chain of predicate instructions along the predicted path of execution within each hyperblock, using encoded static path approximations in each branch instruction and local per-predicate histories to achieve high accuracies. Using a 16-core composable EDGE processor as the evaluation platform, this study shows that hierarchical control prediction can address these issues comprehensively, accelerating single-threaded execution by 19% compared to no predicate prediction, and thus achieving half of the 38% performance gain that ideal predicate prediction would attain.

1. Introduction

Predication offers several benefits, including linearized control flow for high-bandwidth instruction fetch and potentially reduced branch mispredictions. However, predicates are typically evaluated at execution time, which can cause large performance losses compared to correctly predicted branches. Even though previous research has shown that predication can improve performance significantly [1]–[4], superscalar architectures have applied only limited hammock predication due to the complexity and negative performance effects of combining predication with dynamic scheduling in an out-of-order environment [1], [5]. Predicate prediction can mitigate the performance losses associated with execution-time evaluation of control flow arcs, and has been evaluated for conventional superscalar architectures. However, previous predicate prediction research [6], [7] typically assumes that the predicate is only applied to a small number of statically identified hammock branches that are anticipated to be difficult to predict.

Due to power constraints, multiple researchers are exploring execution models that accelerate single threads across multiple lightweight processor cores [8], [12]–[14], with the prediction and fetch functions distributed across those cores. One example is composable processor architectures [8], which use EDGE ISAs [13] to support in-flight execution of multiple predicated hyperblocks. Each block produces one branch result which determines the next block to execute. Within a block, all control is determined by predicates produced by test instructions. Since these microarchitectures are designed to support large windows of execution, they require both good prediction accuracy and high fetch bandwidth. These requirements make heavily predicated code a challenge: if no predicate prediction occurs, large performance losses due to inhibited parallelism result. If predicates are predicted and all predictions are serialized, distributed and high-bandwidth fetching becomes throttled. Non-serialized predictions, however, have the potential to produce high rates of mispredictions in a distributed microarchitecture, since much correlation information is lost. Finally, in heavily predicated blocks, there are many potential paths through the block, and so determining which predicates to predict is important for predictor bandwidth and performance.

This paper studies a number of strategies for addressing these predicate prediction challenges. For brevity, we call the overall approach Hierarchical Control Prediction (HCP). HCP predicts branches independently from predicates, allowing multiple instruction blocks to be quickly predicted and launched in flight. Once in flight, each block’s predicates are predicted. Thus, many in-flight blocks may be predicting their predicates in parallel. Within each block, HCP addresses the issue of which predicates to predict by first predicting all non-predicated test instructions, and then following the predication chain for each, predicting each predicated test instruction until the end of each chain is reached. This approach avoids predicting predicates down paths that are not predicted to be valid. We show that by tagging each branch with a three-bit exit code in the compiler, and choosing the codes for all of a block’s branches to approximate the predicate path through the block to each branch, the overall branch (next-block) prediction accuracy can be kept high without requiring a serialization of all predicate predictions. Finally we show that by coupling an OGEHL predicate predictor with a local
identifies the correct path of execution among the different paths constructing the fetched hyperblock. In addition to leveraging the benefits of predicate prediction without incurring the cost of extra data dependences, this scheme improves the total control flow speculation accuracy and allows the processor to better utilize its fetch bandwidth (see Figure 10(a)).

Figure 2(a) shows a sample C code containing a number of basic blocks. The compiler if-converts all the conditional statements and forms a large hyperblock comprising six different paths guarded by four predicates. Figure 2(b) shows the resulting hyperblock in the intermediate representation. In the figure, \( \text{add}_t <p1> \) means that the \( \text{add} \) operation is predicated on the \( p1 \) predicate with the true polarity. Similarly, \( \text{sub}_f <p0> \) indicates that the \( \text{sub} \) operation is predicated on \( p0 \) with the false polarity. The six paths in the hyperblock and the dependences between the predicates guarding these paths are illustrated in Figure 2(c). To correctly speculate the execution path in the hyperblock, the predicate predictor needs to always speculate the output of the \( p0 \) and \( p1 \). Nonetheless, strictly one of \( p2 \) or \( p3 \) needs to be predicted to avoid the execution of two exclusive paths at the same time. We propose the chain prediction strategy to address this issue.

2.1. Chain Predicate Prediction

In chain predicate prediction, all the predicates are predicted while being dispatched to the reservation stations. The reservation station is augmented with one bit which stores the speculative value of the predicate instructions. If the predicate instruction is not guarded by another predicate, is marked ready-to-issue after being predicted. Predicates \( p0 \) and \( p1 \) in Figure 2 are unguarded predicates that will be ready-to-issue speculatively once dispatched. On the other hand, the guarded predicates are predicted at dispatch time yet not marked as ready. The guarded predicates will wait for a matching predicate to enable them. In the example, predicates \( p2 \) and \( p3 \) are both predicted but are not marked ready. Given \( p1 \) is predicted as \( \text{true} \), only \( p2 \) will be issued speculatively. With the proposed chain predicate prediction scheme, only one of the exclusive paths originating from \( p1 \) is executed speculatively. The speculatively issued predicate instructions send the predicted values to the dependent instructions, enabling those with the matching polarity.

After sending the speculative value to the dependent instructions, the predicted predicate instruction is preserved in the reservation station waiting for its operands to arrive. Once the operands arrive, the predicate instruction is marked ready for the second time. The predicate instruction is executed for the second time with the correct non-speculative values and the result is compared to the speculative value stored in the reservation station. If the result is different from the speculative value, a misprediction signal is raised to flush the pipeline.

A hyperblock is a set of predicated basic blocks in which control may only enter from the top, but may exit from one or more locations [9].
2.2. First Level of Hierarchy: Branch Prediction

2.2.1. Exit Prediction. In parallel with the predicate predictor, which speculatively identifies the correct execution path among the predicated paths forming the fetched hyperblocks, the branch predictor determines the coarse-grain execution path of the program. Unlike conventional branch predictors, which predict the taken/non-taken direction of each branch, the branch predictor in HCP predicts the exit of hyperblocks. The exit of the hyperblock is the ID of the branch that will be taken after the resolution of all the predicates in the hyperblock. In Figure 2, the branch IDs are binary values in the square brackets surrounded by a box. Each branch ID identifies a branch instruction in the hyperblock and is encoded in the opcode of the branch instruction. In the example, given that p1 and p2 are both resolve with true value, the taken branch will be the branch with the ID equal to b10. The exit predictor predicts the ID of the branch which will be taken after the resolution of the predicates in the hyperblock. The branch predictor determines the branch target by using the predicted branch ID to access the branch target buffer. The predicted target is the entry of another hyperblock, which will be fetched next. In HCP, the branch/exit predictor can run ahead without waiting even for the branch instruction to be fetched or the predicates to resolve. This approach decouples the branch prediction from predicate prediction and makes it possible for them to run in parallel without interfering with or waiting for each other (see Figure 1).

Design and implementation of a next block predictor is presented in [8], which comprises two main components: the exit predictor, which predicts the ID of the branch that will be taken out of the hyperblock and the target predictor, which predicts the address of the next hyperblock based on the predicted branch ID. The exit predictor is a hybrid Alpha 21264-like tournament predictor composed of one two-level local, one global, and one choice predictor. The exit predictor uses local and global exit histories built from branch IDs assigned statically to each branch in the hyperblock. The branch IDs are used to construct the local and the global histories in the exit predictor instead of taken/non-taken bits as used in conventional branch predictors. In this implementation, the branch ID is a three-bit value encoded in the branch instruction opcode and assigned based on program order. This simple branch ID assignment does not encode any correlation information in the branch IDs that are used to construct the global and local history information. We propose path-based branch ID assignment to enhance the quality of information captured in the history registers.

2.2.2. Path-Based Branch ID Assignment. We describe the path-based branch ID assignment procedure by two examples. Figure 4 illustrates two predicate trees in which the nodes are predicate instructions and the edges indicate dependences between the predicate instructions. Hollow-headed arrows represent the true polarity and the solid-headed arrows represent the false polarity.
hyperblock exits and the IDs assigned by the compiler to
the corresponding branches. The branch ID assignment in
Figure 4(a) encodes the entire predicate path that leads to
the corresponding hyperblock exits. For example, the b101
branch/exit will be taken if p0, p4, and p5 resolve to
true, false, true, respectively. Figure 4(b) shows the
case where the branch ID assignment can not encode the
entire predicate path because of the presence of a control-

independent point in the tree. In this case, the branch IDs
partially encode the predicate path leading to the hyperblock
exits. Even though the encoding is partial it still contains some
of the correlation information. With path-based branch ID
assignment, the compiler assigns the branch IDs as described
above, enhancing the quality of information collected in the
global history registers of the branch and predicate predictors.
The results presented in Section 3 shows that compile-time
path-based branch ID assignment considerably improves the
accuracy of the run-time predicate prediction.

3. Second Level of Hierarchy: Speculating Predic-
tated Paths

Before diving into the design of the predicate predictor, the
second level of prediction hierarchy in HCP, we present a
general distributed/clustered execution model. The proposed
predicate predictor is designed and implemented considering
the challenges and issues of decentralized prediction, while
having partial correlation information available at the
prediction sites (cores/clusters). One of the contributions of
this work is constructing global history information by using
compile-time generated ISA tags (branch IDs) as well as
the local information available at each prediction site.
The design objective is to achieve a high degree of accuracy, while
minimizing the communication among the prediction sites.
The design tradeoff is between the prediction accuracy and the
amount of correlation information communicated.

It is important to notice that the mechanisms and ap-
proaches presented in this paper (HCP hierarchical speculation
scheme, chain predicate prediction, and path-based branch ID
assignment) are not specific to any architecture. However, the
high-bandwidth fetch offered by predication can be exploited
more effectively in clustered or distributed architectures [8],
[10]–[15]. These architectures provide enough fetch-dispatch-
execute-execution resources to mitigate resource contention caused by
false-path instructions. This section describes the speculation
of predicated paths in a larger context that includes both
distributed and conventional architectures.

Based on the chain prediction approach, predicate prediction
is performed in the dispatch stage. As depicted in Figure 5,
the proposed predictor design assumes that each core/cluster
dispatches the instructions mapped to that core/cluster. Thus,
each core/cluster is augmented with an independent predicate
predictor. At the beginning of each fetch cycle, the core/cluster
that predicts the exit branch ID broadcasts the predicted branch
ID to all the cores/clusters that will execute the instructions in
the hyperblock to be fetched. In addition, it is assumed that
each instruction in the hyperblock is always mapped to the
same core/cluster on which was mapped in previous iterations.

Figure 3: (a) 16-core TFlex processor. (b) Components and structure of a single TFlex core. (c) Internal organization of the next-block
predictor.

Figure 5: Distributed execution of a hyperblock. In all the iterations,
the predicate instructions are mapped to the same core. Each core is
augmented with a predicate predictor.
3.1. Base Predictor

We use a GEometric History Length (GEHL) [16] branch predictor as the base prediction algorithm because, compared to the other state-of-the-art predictors, GEHL delivers high accuracy, requires less state, and has lower complexity. As depicted in Figure 6, the GEHL predictor uses multiple prediction tables to generate a prediction. Each entry in the tables is a signed saturating counter. The core-local prediction table is the extra table indexed by CLPHR, which augments the GEHL predictor to improve the prediction accuracy.

(see Figure 5). With this general assumptions, this section discusses the design and implementation of a distributed predicate predictor that speculates on the correct path of execution in the fetched hyperblocks and functions as the second level of hierarchy in the HCP scheme.

3.2. Constructing Global History Information

In this section, different approaches of constructing the global predicate history are presented. The goal is to achieve a high degree of accuracy by exploiting the correlation between the predicates, while minimizing the communication among the cores/clusters.

3.2.1. Core-Local Predicate History Register (CLPHR).

The first approach, Core-Local Predicate History Register, implements one extreme of the design space. CLPHR restricts each predictor to only use the local information available at the prediction site. With this approach, the predictors do not communicate any information to one another. Each core/cluster owns its exclusive global history register, which only tracks the predicate instructions mapped to that core/cluster. The MPKI results for this approach are presented in the second column of Table 1. The CLPHR approach is fairly accurate because the dependent instruction are usually mapped to the same core/cluster. Therefore, dependent predicate instructions are also mapped to the same core/cluster. CLPHR exploits the correlation between these instructions. In addition, the base GEHL predictor uses long histories, which to some extent makes the predictor robust to information loss. In fact this approach trades prediction accuracy for no communication. The advantage of the CLPHR approach is that it requires no communication.

3.2.2. Global ID History Register (GIHR).

The next approach, Global ID History Register (GIHR), implements another extreme of the design space. This approach uses the branch IDs predicted by the block exit predictor to construct the global history register. As discussed before, the block exit predictor uses the same approach to construct its own global history register. In this approach, the branch IDs are assigned statically at compile-time based on program order without encoding any predicate path information. To examine this approach, we only used three bits in the branch instruction opcode. Using three bits implies that each hyperblock can only have eight exit points or branch instructions. To construct the GIHR global history register at each predicate prediction site (dispatch stage of each core/cluster), the exit predictor broadcasts the predicted branch ID to all of the cores/clusters that will execute the hyperblock. The broadcast of the predicted branch ID can be combined with the fetch command that is sent regardless to all the cores/clusters to initiate the

We use the best reported GEHL predictor in [16] with eight tables and a 125-bit global history register in each core/cluster. With this organization, the total state of each predictor is 11 Kbytes. This large configuration is used to evaluate various schemes of constructing global history without incurring accuracy loss due to the small storage budget. After identifying the best strategy to construct the global history, the predictors are sized down such that they deliver about the same accuracy, but with a smaller state.
fetch/dispatch process. The GIHR trades limited broadcast communication for better accuracy. However, as the results in the third column of Table 1 show, on average compared to CLPHR, GIHR incurs 0.84 more mispredictions per thousand instructions. The low accuracy is due to the low quality of the information captured in the global history register. The branch IDs do not encode any predicate path information, which severely reduces the quality of the information in GIHR.

3.2.3. GIHR ⊕ CLPHR. To take advantage of both approaches, we have combined the GIHR and CLPHR approaches by augmenting the GEHL predictor with and extra prediction table. As illustrated in Figure 6, the GEHL predictor is augmented with another table indexed by the CLPHR instead of the main global history register, which is GIHR in this case. To generate the prediction, the value retrieved from this extra table is added to the values retrieved from the other tables. As presented in Table 1 columns four through six, this compound approach is examined using various sizes for CLPHR. The results suggest that by adding the extra table indexed by a 10-bit CLPHR, the augmented GIHR method incurs 0.05 fewer MPKI than the CLPHR method. This approach trades extra space for better prediction accuracy. The core-local predicate history register (CLPHR) captures the correlation between the predicates and compensates for the lack of correlation information in the global ID history register (GIHR) resulting in improved prediction accuracy.

3.2.4. GPHR.all ⊕ CLPHR. The GPHR/all approach implements another point in the design space. In GPHR/all, all the prediction sites (cores/clusters) use the same global history register, which contains all the predicate outputs. In this approach when a core makes a prediction, it broadcasts the value to all the predictors to update their global history register. By using this approach, every prediction site can use all the information available about the predicates. Because of the large number of broadcast messages and the complexity of sequencing the predicate values, this approach is not practical in a distributed architecture, however, it can be adopted for conventional architectures. This approach trades broadcast communication for better correlation information in the global history registers. The results in the seventh column of Table 1 show that without CLPHR augmentation, GPHR/all approach performs worse than both CLPHR and compound GIHR. The lower accuracy is due to the pollution of the global history register. As discussed earlier, it is often the case that only one of the predicated path is executed. That is, only a subset of the predicates that determine the correct path should be included in the global history. Augmentation with a core-local prediction table compensates for the pollution by trading space for better accuracy (column eight).

3.2.5. GPHR.exit ⊕ CLPHR. The branch IDs, which are used to construct the global history, do not encode any correlation information. By using the proposed compile-time path-based branch ID assignment, the global history constructed from these branch IDs contains the correlation information between predicates. We refer to this method that uses the path-based branch IDs to construct the global history information as GPHR.exit. This approach trades limited communication for better accuracy. As the results in Table 1 column nine and column ten show, GPHR.exit outperforms all of the previous approaches. Augmenting GPHR.exit with the core-local prediction table also improves the prediction accuracy by 0.18 MPKI. This approach takes advantage of compile-time statically assigned path-based branch IDs to improve the run-time dynamic predicate prediction accuracy.

The best prediction accuracy is achieved when the global history register is constructed from branch IDs assigned statically based on the predicate path leading to the branches. Using path-based branch IDs outperforms the GPHR/all approach, which stores all the predicate values in the global history register. Since hyperblocks consist of multiple paths of execution from which only one or two gets executed, including all the predicates in the global history information decreases prediction accuracy.

3.3. Sizing the Predictor

Starting with the 11K predictor, we alter the number and width of the table entries to find a predictor with a smaller size but comparable accuracy. The tradeoff is between the storage budget allocated to the predictor and its accuracy. Sizing down the tables increases the chance of destructive aliasing and reduces the accuracy of the predictor. By examining different sizes and widths in an ad hoc manner, a predictor with comparable accuracy is found. The chosen predictor, which is referred to by the 2K predictor, comprises 17.5 Kbits or
2.1875 KBytes of state. The 2K predictor is only 0.17 MPKI less accurate than the 11K predictor when used in the 16-core configuration, the common case for a distributed architecture (see Figure 7). It also delivers acceptable accuracy with other configurations. The results in Figure 7 show that running with a smaller number of cores results in less storage, more aliasing, and reduced accuracy when employing the 2K predictor. On the contrary, the accuracy of the 11K predictor does not significantly change by reducing the number of cores. It even performs slightly better with the 1-core configuration. In this case, the CLPHR captures all the correlation information while constructive aliasing improves the quality of the information stored in the tables. Furthermore, the 11K was sized and optimized for a single core configuration, while the 2K predictor is designed for a 16-core configuration.

4. Experimental Results

4.1. TFlex Composable Lightweight Processor

This section evaluates the hierarchical control flow prediction scheme using the TFlex composable lightweight processor [8], which implements an EDGE ISA [13]. EDGE ISAs are low-overhead, fully predicated instruction set architectures [17]. By supporting block-based execution model, EDGE ISAs eliminate the need for the reorder buffer in the micro-architecture. The register renaming stage is also eliminated through supporting direct instruction communication. These characteristics alleviate two major overheads of predication, pipeline stalls because of fetching and allocating false-path instructions in the reorder buffer and multiple register definitions along multiple if-converted control paths. Furthermore, distributed architectures similar to TFlex provide a high fetch bandwidth and large number of micro-architectural resources that considerably reduce resource contention caused by false-path instructions.

The TFlex processor is composed of distributed lightweight yet full-fledged processors that each can run threads individually or form a more powerful logical processor by aggregating with other cores. The operating system can run 16 threads each on one core, allocate all of the 16 cores to one thread, or assign a various number of cores to different threads. To provide this capability, all micro-architectural structures are distributed across the chip, which includes instruction and data caches, branch predictor, register file, instruction window, and execution units (see Figure 3). Distributed fetch, execute, commit, flush, and misspeculation recovery protocols allow these structures to perform as a concrete logical processor.

To evaluate HCP, the TFlex cycle accurate simulator [8] is augmented with the compound GPHR.exit predicate prediction approach. The results are reported for eight integer SPEC2000 benchmarks simulated with the reference (large) data set using single simpoints [18]. The baseline is heavily predicated code, which is compiled with the –Omax flag using the compilation techniques presented in [19].

4.2. Performance Impact of HCP

4.2.1. Chain Predicate Prediction. Figure 8 shows achievable speedup limit with perfect chain prediction. As shown in Figure 8 perfect prediction without delaying the prediction of the guarded predicates is only slightly better than the perfect chain prediction. This slight difference results from the fact that unguarded predicates are predicted and passed down very early to the dependent instructions. On the other hand, dependent guarded predicates are also usually mapped close to or on the same core as the guarding predicates.

4.2.2. Speedup. Figure 9(a) shows the speedup over the baseline (hyperblock execution without predicate prediction) along with the achievable performance with perfect prediction. As shown in Figure 8, employing high-degree of predication with no predicate prediction improves the performance by 22%; however, because of the extra data dependences introduced by predication, it does not benefit from the 38% performance improvement achievable by perfect predicate prediction. The 2K predictor performs almost the same as the 11K predictor in the 16-core configuration and achieves a 19% speedup, half of the potential performance improvement.

4TFlex architectural parameters: 
- Instruction Supply: Partitioned 8KB I-cache (1-cycle hit); Local/Gshare Tournament predictor (8K+256 bits, 3 cycle latency) with speculative updates; Local: 64(L1) + 128(L2), Global: 512, Choice: 512, RAS: 16, CBT: 16, BTB: 128, Btype: 256.
- Execution Out-of-order execution, RAM structured 128-entry issue window, dual-issue (up to two INT and one FP).
- Data Supply Partitioned 8KB D-cache (2-cycle hit, 2-way set-associative, 1-read port and 1-write port); 64-entry LSQ bank; 4MB decoupled S-NUCA L2 cache (8-way set-associative, LRU-replacement); L2-hit latency varies from 5 cycles to 27 cycles depending on memory address; average (unloaded) main memory latency is 150 cycles.
By comparing the graphs in Figure 9(a) and Figure 9(b), we see a direct correlation between the size of hyperblocks and achievable speedup. For instance, bzip, which has the largest hyperblocks, shows the highest potential for speedup, whereas perlbin, which has the smallest hyperblocks, has little potential for performance improvement. This suggests that the more aggressive the prediction, the more effective the hierarchical control prediction. It is also noticeable that path-based branch ID assignment improves the speedup from 15% to 19% as shown by the two rightmost bars in Figure 9(a).

Among the benchmarks, bzip, crafty, gcc, and parser achieve a speedup that is more than half of the potential speedup. This improvement is the result of the high prediction accuracy achieved by the interplay between the compiler and the microarchitecture (path-based branch ID assignment). Even though perlbin and vortex do not provide a high potential speedup from predicate prediction, the achieved performance is very close to the limit. The relatively low accuracy of the predictor for the gzip and twolf benchmarks results in a low performance benefit compared to the achievable limit. Nevertheless, both of the benchmarks benefit from predicate prediction.

### 4.2.3. Throttling the Speculation

The confidence threshold can be used to throttle the speculation in the cases where the predictor is not confident about the prediction. In 16-core configuration, the only benchmark that benefits from confidence-based speculation throttling is twolf, which suffers from high misprediction rate. As depicted in Figure 10(a) in the 16-core configuration, despite the fact that confidence estimation reduces the number of predicate mispredictions, the total number of mispredictions is not reduced significantly. The predicate predictor only contributes to a relatively low fraction of overall mispredictions, and therefore the predicate misprediction reduction does not manifest as the reduction in the total number of mispredictions. Furthermore, speculation throttling suppresses a fraction of correct predictions, which exacerbates the results.

As shown in Figure 10(b), with 2-core configuration the performance improvement potential is less than the potential level in the 16-core configuration. There are fewer resources available to take advantage of the parallelism exposed by predicate prediction, which eliminates the extra data dependencies introduced by if-conversion. In this configuration, the misprediction penalty is more severe because of the reduced fetch and execution bandwidth. The increased misprediction rate (see Figure 7) due to the reduced amount of storage and increased degree of destructive aliasing exacerbates the performance loss. The crafty and twolf benchmarks, which already suffer from relatively low prediction accuracy, lose performance that can be mitigated by speculation throttling using a high confidence threshold. The gzip and perlbin benchmarks achieve performance improvement with higher confidence levels. The other benchmarks do not benefit from confidence estimation for similar reasons described for the 16-
core configuration. Nonetheless, in the 2-core configuration, the distributed predicate prediction scheme improves performance by 4%, which is half of the potential speedup.

4.2.4. Issue Width and HCP. Figure 11 shows the limit and the achieved speedup with different issue widths (number of aggregated cores). The larger the issue width (the amount of micro-architectural resources), the higher the speedup achieved by employing HCP. Larger window configurations with more micro-architectural resources can better exploit the parallelism exposed by predicate prediction. As Figure 11 shows, on average, HCP consistently achieves half of the potential speedup.

5. Related Work

Various studies show the effectiveness of predication in dynamically scheduled architectures. Pnevmatikatos and Sohi [1] evaluate the effect of predicated execution on instruction level parallelism in the presence of branch prediction. They show that predication can increase the effective size of basic blocks, which provides more opportunity for both the compiler and the dynamic hardware to extract fine-grained parallelism. The results also suggest that full predication can considerably increase the number of dynamic instructions executed between two branch mispredictions, which improves throughput. Chang et al. [3] employ predication to reduce the number of branch mispredictions by eliminating hard-to-predict branches. In this approach, profiling is used to identify the hard-to-predict hammock branches. In this approach, if-conversion is applied only in a limited form. This technique reduces the number of mispredictions; however, because of the limited application of if-conversion, this approach does not take full advantage of predication. Mahlke et al. [2] study full and partial predication and show that significant performance improvement is achievable by hyperblock formation or even partial predication in a relatively wide-issue machine (8-wide). To balance prediction and control flow, August et al. [4] study where and when predication should be employed. They conclude that due to resource contention caused by the execution of false-path instructions, if-conversion should be applied selectively based on detailed analysis of the dynamic program behavior and the availability of micro-architectural resources. Even though the prior art suggests that predication can improve performance significantly, predication is only employed in a limited form (hammock predication) in out-of-order processors due to the lack of techniques that can alleviate the overheads of heavy predication in the dynamically scheduled architectures.

Different approaches have been proposed to cope with the overheads of predication [1], [4]–[7], [20], from which we focus on predicate prediction schemes [6], [20] and wish branches [20]. Chuang et al. [6] propose predicate prediction for out-of-order processors to alleviate the problem of multiple register definitions along the if-converted control paths. They reverse all the if-conversions by predicting the predicates, which reduces the effectiveness of predication penalty. To preserve the benefit of predication, this method utilizes a replay mechanism that makes the predicate misprediction penalty less than the branch misprediction. To preserve the benefit of predication on hard-to-predict branches, Quiñones et al. [7] propose selective predicate prediction that predicts predicates selectively based on the estimated confidence of prediction. This approach maintains 84% of the if-converted hard-to-predict branches (predicates) and outperforms the original predicate prediction approach [6]. Kim et al. [20] adopt a different approach to use predication for hard-to-predict branches. In their approach, the compiler preserves the branch instruction in the form of wish branch/jump in the binary. This way, the branch predictor, which is augmented with a confidence estimation mechanism, can dynamically decide whether to fetch the predicated code or predict the branch. It may be beneficial to combine predicate prediction with wish branches.

The previous predicate prediction research typically assumes that the predication is highly restrained and only applied to hammock branches that are difficult to predict. Within such hyperblocks, there are only two execution paths that are guarded by a single predicate. This type of limited hammock predication does not offer the full benefits of predication. The prior predicate prediction research does not provide accommodations for speculatively identifying the correct path within the hyperblocks comprising multiple predicated paths (more than two). The proposed HCP scheme and chain predicate prediction addresses these problems. This paper also studied the effective approaches of constructing global history using path-based branch IDs and proposed a highly accurate predicate predictor design, which can be used in both conventional and distributed architectures.

6. Conclusions

Prediction of predicates is essential for good performance on out-of-order architectures running aggressively predicated code, in which many predicated paths can be fetched and in flight concurrently. Choosing which predicates to predict, and how to predict them well, is a challenge when code is a succession of predicates punctuated by branches, only some of which should be predicted. This challenge is particularly acute in EDGE architectures, in which aggressive predication is a crucial part of the execution model. Hierarchical control
prediction picks control points in the code (branches) and at instruction dispatch predicts a single path through each predicated region. With this approach, multiple predicate prediction chains can be dispatched in parallel across distinct control regions. Essential to HCP is finding the right information to form good history vectors locally. In this paper, we evaluated the use of static tags in individual branches to support effective distributed predicate prediction with small (2KB) tables in each core.

The results show a 19% speedup (4% of which is due to compile-time path-based branch ID assignment) over predicated code with no prediction, which is half of the 38% performance increase, which is theoretically achievable with perfect prediction. More importantly, this prediction scheme operates with fully distributed control, and small numbers of lightweight control messages necessary to orchestrate correct global execution. Additionally, we show that throttling predictions based on confidence estimation turns out to be unimportant for a large-window (16-core) configuration. Since there are almost always branch mispredictions in flight anyway, the small additional reduction in mispredictions from throttling low-confidence predicate predictions does not improve performance. On smaller-window configurations, of course, confidence estimation remains important to determine which predicates should be predicted to maximize performance while reducing misprediction penalties.

In the long term, effective predicate prediction may enable more aggressive predication, in which any branch that is hard to predict (and which does not contain a function call down any frequently traversed paths), is predicated. With more aggressive conversion of hard-to-predict branches, confidence estimation should be important for even large-window configurations, and may enable a reduction in misprediction rates above and beyond what the best current branch predictors (i.e., L-TAGE, GEHL, and perceptrons) are able to achieve. Dataflow architectures that rely on complete predication are unlikely to compete with superscalar processors unless many of the control flow arcs are predicted. HCP is one approach for addressing that challenge.

References

Tolerating Delinquent Loads with Speculative Execution

Chuck (Chengyan) Zhao, J. Gregory Steffan, Cristiana Amza, and Allan Kielstra
Dept. of Electrical and Computer Engineering
University of Toronto
{czhao,steffan,amza}@eecg.toronto.edu
IBM Toronto Laboratory
IBM Corporation
kielstra@ca.ibm.com

ABSTRACT
With processor vendors pursuing multicore products, often at the expense of the complexity and aggressiveness of individual processors, we are motivated to explore ways that compilers can instead support more aggressive execution. In this paper we propose support for fine-grain compiler-based checkpointing that operates at the level of individual variables, potentially providing low-overhead software-only support for speculative execution. By exploiting this checkpointing support to improve the performance of sequential programs, we investigate the potential for using speculative execution to tolerate the latency of delinquent loads that frequently miss in the second-level (last level on-chip) cache. We propose both data and control speculation methods for hiding delinquent load latency. We develop a theoretical timing model for speculative execution that can yield up to 50% relative speedup. Our initial testing using synthetic benchmarks strongly supports this model.

1. INTRODUCTION
While today’s computer hardware is characterized by the abundance of processor cores in multicore chips, the individual processors themselves are generally not much more aggressively speculative or out-of-order than previous designs. Instead the primary technique to cope with mounting latency to off-chip memory is multithreading, such as Intel’s Hyperthreading and SUN’s multithreaded Niagara processor: in these designs the long latency of an off-chip load miss can be tolerated by executing another thread for the duration of the miss. However, there is a dearth of threaded software—especially for desktop computing—which will limit the impact of solutions that depend on multithreading alone.

Prefetching is also a well-studied technique for addressing memory latency, via both hardware and compiler techniques. However, prefetching for irregular data accesses can be difficult, since irregular data accesses are difficult to predict and since there is a close trade-off between tolerating latency and increasing overhead and traffic. This environment underlines the importance of selective compiler techniques for tolerating memory latency.

One way to be more selective is to focus on delinquent loads (DLs) [8, 36]. A DL is a particular memory load in a program that frequently misses in a cache—typically the last-level cache on-chip. In other words, for many applications a small number of DLs contribute a large fraction of all last-level cache load misses. Hence DLs, should they be reasonably persistent across target architectures, may be a good focal point for compiler optimization.

1.1 Tolerating DLs with Compiler-BasedCheckpointing
We propose a software-only method for checkpointing program execution that is implemented in a compiler. In particular, our transformations implement checkpointing at the level of individual variables, as opposed to previous work that checkpoints entire ranges of memory or entire objects. The intuition is that such fine-grain checkpointing can (i) provide many opportunities for optimizations that reduce redundancy and increase efficiency, and (ii) facilitate uses of checkpointing that demand minimal overhead, such as tolerating DL latency. We propose two methods of tolerating DL latency that exploit compiler-based fine-grain checkpointing to implement software-only control and data speculation. We evaluate the performance potential through both theoretical analysis and synthetic benchmark testing on
real machines.

1.2 Contributions
We make the following contributions in this paper:

- we implement a software-only checkpointing framework that leverages on compiler analysis and targets aggressive overhead reduction;
- we propose control and data speculative compiler transformations that will overlap with DLs;
- we propose a theoretical performance model of relative speedups and implement synthetic benchmarks whose evaluation results strongly support the modeling.

2. RELATED WORK
Our techniques are based on a wide spectrum of existing work in related areas, including prefetching [8, 24], multithreading [11, 38], checkpointing [12, 19], speculation [9, 13] and identifying DLs.

Panait at el. [36] investigated techniques to identify DLs statically. They examine code at the assembler level, categorize memory load instructions into various groups, and calculate a final weight based on profiling information obtained through training. They single out 10% of data loads that generate 90% of all cache misses. However, their approach is based on short-distance predictable memory behaviors. Thus their scheme is applicable only in isolating level-1 DLs. In addition, the identified DLs are memory locations in assembly format, which is non-trivial to map to source locations. Zhao at el. [55] introduced a lightweight and online runtime methodology to identify DLs. They observe that bursty online profiling and mini simulation of short memory traces can largely represent the underlying memory behaviors. Their simulation provides 61% overall accuracy with only 14% extra runtime overhead. However, they also introduce a 57% false positive ratio, a prohibitive number for any speculative compiler adopting their technique. We identify DLs through a speculative program simulation based on PIN [4, 23, 35]. It can be configured to deal with artificially many levels of cache and is capable of identifying DLs at any designated cache level. It provides service to map loadPCs back to source program locations, which is particularly useful to enable compiler optimizations.

3. COMPILER-BASED FINE-GRANULAR CHECKPOINTING
Checkpointing [12, 19, 21, 40, 48, 49] is the process of taking a snapshot of program execution so that execution can rewind to that snapshot later if desired. Checkpointing has a wide range of uses and includes both hardware and software implementations. While proposed hardware-based solutions [2, 30] can perform well, they have yet to be adopted broadly in commercial systems. Software-only checkpointing solutions [19, 21, 37, 49] are therefore more immediately practical, although their inherent overheads can be prohibitive. In contrast with past work on coarse-granularity checkpointing based on copying large memory regions or cloning objects, in this section we propose a relatively lightweight compiler-based approach to checkpointing that operates at the level of individual variables.

Overview Figure 1 presents a high-level overview of our checkpointing system. The system takes as input a C-based program, with annotations that indicate where a checkpoint region begins and ends, as well as code that decides whether the checkpoint should be committed or rewound. Our checkpointing transformations and optimizations are implemented as passes in the SUIF [3, 14] compiler, which outputs transformed C code that can then be compiled to target a number of platforms (currently x86 via gcc and POWER via IBM’s xlc compilers). This source-to-source approach allows us to capitalize on all of the optimizations of the back-end compilers.

Undo-Log vs Write-Buffer The most important design decision in a checkpointing scheme is the approach to buffering: whether it will be based on write-buffer [16, 26] or alternatively an undo-log [18, 31]. A write-buffer approach buffers all writes from main memory, and therefore requires that the write-buffer be searched on every read. Should the check-
point commit, the write-buffer must be committed to main memory; should the checkpoint fail, the write-buffer can simply be discarded. Hence for a write-buffer approach the checkpointed code proceeds more slowly, but with the benefit that parallel threads of execution can be effectively checkpointed and isolated (e.g., for some forms of optimistic transactional memory [16, 28]). An undo-log approach maintains a buffer of previous values of modified memory locations, and allows the checkpointed code to otherwise read or write main memory directly. Should the checkpoint commit, the undo-log is simply discarded; should the checkpoint fail, the undo-log must be used to rewind main memory. Hence for an undo-log approach the checkpointed code can proceed much more quickly than a write-buffer approach. For this work, since we are considering only a single thread of execution with focus on performance, we proceed with an undo-log approach.

Base Transformation Given that we implement an undo-log based approach, the base pass of the checkpointing framework is to precede all writes with code to backup the write location into the undo-log. As illustrated in Figure 2(a), within the specified checkpoint region the variables x, y, and z are all modified and preceded with a backup() call. The backup() call takes as arguments a pointer to the variable to be backed up and its size in bytes. Figure 3 illustrates our initial design of an undo-log, where we have divided the undo-log into two structures: (i) a data buffer which is essentially a concatenation of all backed-up data values of arbitrary sizes; and (ii) a meta-data buffer which stores the length and starting address of each element. As an example, Figure 3(b) shows the contents of an undo-log after three backup() calls. When a checkpoint commits, we simply move the data and meta buffer pointers back to the start of each buffer; when a checkpoint must be rewound, we use the meta buffer to walk through the data buffer, writing each data element back to main memory. In future work we will more thoroughly investigate possibilities and trade-offs in the implementation of the undo-log.

Optimizations Our base transformation for fine-grain checkpointing provides significant opportunities for optimizations. Given the initial code shown in Figure 2(a), we can perform several optimizations. For example, as illustrated in Figure 2(b) a hoisting pass which will hoist the backup of any variable written unconditionally within a loop to the outside of that loop (variable z in the example); note that such hoisting would not be performed by a normal hoisting pass since the write to the variable is not necessarily loop invariant. Note also that we do not hoist variable y in the example since it is only conditionally modified—whether to hoist such cases is a trade-off that will be studied in future work. A second optimization is to aggregate backup() calls for variables which are adjacent in memory, potentially rearranging the layout of the variables to ensure that they are adjacent.1 Aggregation reduces the overhead of managing adjacent variables individually (variables z and x in the example). We have implemented an ini-

1Note that for a source-to-source transformation this isn’t necessarily a safe optimization as the back-end compiler may further rearrange the variable layout—an implementation in a single unified compiler would not have this problem.
lining pass so that a `backup()` is not actually implemented as a procedure call but instead consists only of the bare instructions for performing the backup. In future work, we will also investigate redundancy optimizations to remove redundant and unnecessary `backup()` calls.

4. DELINQUENT LOADS IDENTIFICATION AND PERSISTENCE

**DL Identification** We identify DLs by profiling second-level (L2) cache misses using a cache simulator based on PIN [23] that we developed for this work. The PIN framework identifies each memory-access instruction from the application and directs them to a software cache model. Within each memory access, the software model captures necessary access signatures (read vs. write, effective memory address, length of data, etc.) and performs efficient cache simulation. The software cache model is easily configurable when dealing with various cache configurations, including the total levels of cache, cache size, cache-line size, degree of associativity, replacement policy, etc.

One compelling feature of this infrastructure is that, when a benchmark is compiled with debug information, it allows us to directly associate load and store instructions with their corresponding source code location. Hence the simulator can reliably map each load instruction that is responsible for a large fraction of L2 cache misses back to the offending source code location. In this paper, we will consider a particular load instruction to be a delinquent load if it is responsible for greater than 10% of all L2 cache misses for a program. We will also refer to the actual percentage of L2 cache misses as the *significance* of that delinquent load (i.e., a load that is responsible for all of a program’s L2 cache misses would have a significance of 100%).

**DL Persistence** We use SPEC2000INT [10] benchmarks, compiled with compilers of various vendors (gcc and icc), versions (gcc 3.4.4, 4.0.4, 4.1.2, 4.2.4, 4.3.2, and icc 9.1), and optimization levels (O0, O2 and O3) to study DL locations and properties. We configure the cache simulator with 2-level cache that covers a large variations of cache size, cache line size and degree of associativity.

Our initial investigation of all SPEC2000INT C benchmarks found that only a subset of the applications contain DLs. Within that subset, the DLs have the following persistent properties:

- the DLs are persistent across various L2 cache configurations (size, line size, ways of associativity), as long as the working set doesn’t entirely fit into the L2 cache;
- the DLs are persistent across different compilers, including vendors, versions and optimization levels;
- the DLs are persistent across inputs (training or reference).

Interested readers should refer to [54] for further detail.

5. TOLERATING DELINQUENT LOADS WITH SPECULATIVE EXECUTION

In this section we propose two techniques that leverage compiler-based fine-grain checkpointing to tolerate DLs, namely data and control speculation. For a single-threaded speculation, we must make a prediction about the resulting value of a DL and execute code that uses that prediction to make progress rather than waiting for the DL result value from off-chip. This approach exploits the parallelism provided by a wide-issue superscalar processor that can execute instructions with memory access in parallel. Ideally the latency of the DL is hidden when the prediction is correct, but execution can rewind and re-execute using the correct DL value should the prediction be incorrect.

5.1 Overview

Figure 4(a) illustrates the challenge presented by a DL: the L2 miss latency for a DL can be lengthy, and the computation that follows the DL (`work()`)
likely depends on the DL’s result value (x). Figure 4(b) provides an overview of how to tolerate a DL by overlapping the DL miss latency with speculative execution of the subsequent code using a predicted value (v). The DL is scheduled as early as possible, followed by the generation of a predicted value (v).

The computation proceeds using the predicted value (work(v)), with that computation being checkpointed to support execution rewind. When the computation is complete, we compare the predicted value with the actual value, and if they are equal then we can commit the checkpoint (as shown in Figure 4(b)). Ideally such a successful prediction and speculation will result in a performance gain relative to the non-speculative original code. Should the value be mispredicted, as illustrated in Figure 4(c), then we must rewind the checkpoint and re-perform the computation with the correct result value of the DL (work(x)). The combined overheads of checkpointing as well as rewinding and retrying the computation can result in a performance loss relative to the original code.

5.2 Data Speculation

The first method of tolerating DL latency that we evaluate is data speculation (DS) where we predict the result value of the DL and use it to continue execution speculatively, as illustrated in Figure 5. After issuing the DL as early as possible (1), predicting the DL’s data value (2), starting the checkpoint (3), and speculatively executing based on that predicted value (4), we then attempt to commit the speculation. The commit process first checks whether the prediction was correct (5): if so then the checkpoint is committed (6), otherwise the checkpoint is rewound (7) and the computation is re-executed using the correct DL result value (8).

5.3 Control Speculation

Whenever the result value of a DL is used solely within a conditional control statement, as shown in Figure 6, we have an interesting opportunity: rather than predicting the exact result value of the DL we can instead merely predict the boolean result of the conditional—which ideally will more easily be an accurate prediction than predicting the exact result value. We call this form of speculation control speculation (CS), which is essentially a special-case of data speculation. The speculative compiler transformations of tolerating control speculation are given in Figure 6.

Modern processors perform branch prediction and speculatively execute instructions beyond the branch—however this speculation is limited to the size and aggressiveness of the processor’s issue window. With compiler-based control speculation we can ideally speculate more deeply, allowing greater opportunity for tolerating all of the latency of a DL.

6. PERFORMANCE

In this section, we give both a theoretical performance model and a practical evaluation of the proposed speculative techniques on real machines. We first present a mathematical analysis of the implicit DL memory overlapping model and give theoretical predictions of potential performance benefits. We show that the theoretical model predicts approximately 50% relative speedup. We then ap-
apply this model on synthetic benchmarks running on real machines and demonstrate that the relative performance gain of the synthetic benchmarks closely matches the theoretical prediction.

6.1 Theoretical Performance Modeling

Figure 7 illustrates the ideal timing model for overlapping execution with DLs. Figure 7(a) is the normal sequential model where the total execution time is the sum of both DL cycles and the work cycles whose continuation relies on the DL. This represents the conditions where the DL’s value is immediately needed to allow execution to proceed, thus the program stalls until the DL value returns. Under the overlapped model (Figure 7(b)), the program continues with the predicted value while the memory system is serving the DL. This resembles a level of memory-level parallelism though there is no explicit parallel thread needed to fetch the DL. Thus the total execution time is the maximum of the two. This model the cases when either the DL’s value not being immediately needed or the DL being used to make a predictable control-flow decision and therefore its precise value is less important.

Let $CL$ denote the cycles of a cache miss (DL) and let $C$ denote the cycles of work that overlaps with the DL, we have

$$T_{\text{sequential}} = CL + C$$

$$T_{\text{speculate}} = \max(CL, C)$$

Let $S$ denote the relative speedup of overlapping execution with DL, we give the definition of $S$

$$S = \frac{T_{\text{sequential}} - T_{\text{speculate}}}{T_{\text{sequential}}} = \frac{CL + C - \max(CL, C)}{CL + C}$$ (1)

Thus the ideal theoretical relative speedup for only overlapping with only a L1 cache is

$$S = \frac{CL_1 + C - \max(CL_1, C)}{CL_1 + C} = \begin{cases} \frac{C}{CL_1 + C}, & \text{if } C < CL_1 \\ \frac{CL_1}{CL_1 + C}, & \text{if } C \geq CL_1 \end{cases}$$

Similarly the ideal theoretical relative speedup for only overlapping with only a L2 cache is

$$S = \frac{CL_2 + C - \max(CL_2, C)}{CL_2 + C} = \begin{cases} \frac{C}{CL_2 + C}, & \text{if } C < CL_2 \\ \frac{CL_2}{CL_2 + C}, & \text{if } C \geq CL_2 \end{cases}$$

In addition, we obtain the theoretical relative speedup for overlapping with combined L1 and L2 cache by aggregating individual speedups.
\[
S = \begin{cases} 
\frac{C}{CL_1} + \frac{C}{CL_2}, & \text{if } 0 \leq C < CL_1 \\
\frac{CL_1}{CL_1} + \frac{C}{CL_2}, & \text{if } CL_1 \leq C < CL_2 \\
\frac{CL_1}{CL_1} + \frac{CL_2}{CL_2}, & \text{if } C \geq CL_2 
\end{cases}
\]

Figure 8 presents three theoretical relative speedup curves for overlapping with L1 cache only, with L2 cache only, and overlapping with combined L1-and-L2 cache respectively. It shows both the overall similarity and individual differences. For ease of comparison, we fix the L1 cache miss latency to 20 cycles (\(CL_1\)) and L2 cache miss latency to 500 cycles (\(CL_2\)).

The curve that overlaps with L1-only workload goes sharply to its peak from 0 to \(CL_1\) (20) cycles in the beginning. Since the L1-miss-and-L2-hit cycles are relatively short, the curve has only limited room to stretch before reaching its theoretical peak, which is predicted to be 50% when the overlapped cycles (\(C\)) equal to L1-miss-and-L2-hit cycles (\(CL_1\)). The curve that overlaps with L2-only work can be treated as horizontally scaling the L1 curve to match with L2-miss-and-memory-hit cycles (\(CL_2\)) and its theoretical performance upper-bound is also 50%. Given ideal workloads, the two theoretical speedups can further combine and generate an aggregated effect that can cross the 50% threshold, presented as the \(CL_2\)-centered triangle-like area in Figure 8.

### 6.2 Micro Benchmarks

We developed a set of synthetic benchmarks for real-machine evaluation. This includes linked list (linklist), binary search tree, B-tree, red-black tree, AVL tree, and hashtable, etc. They behave similarly in that accesses to dynamically allocated data structures result in frequent cache misses (DLs). We use linklist as the representative for this initial study. We make each node in the linklist larger than the cache line size on the machine it evaluates. To exacerbate the situation, we randomize the starting address of each node, which helps undermine the hardware prefetcher. By adjusting the number of nodes in the linklist, we achieve the effect of either polluting only the L1 cache (L1-DL), or polluting both L1-and-L2 caches (L1L2-DL) through a single linklist traversal. The empirical list size we use is 4K nodes for L1-DL and 2M nodes for L1L2-DL, respectively. We use \textit{RDTSC} [1, 50] for fine-grain time measurement.

The machine used for evaluating the benchmarks has a single-core 3.0GHz Pentium-IV CPU, with a 16KB 4-way set-associative L1 data cache, a 12KB 8-way set-associative L1 instruction cache, and a 512KB 8-way set-associative shared L2 cache. The cache line size is consistent at 64B. Each measurement data point is the arithmetic average of at least 5 independent runs.

![Figure 9: Relative speedup overlapping with L1-only DL on real machine](image)

### 6.3 Micro Benchmark Performance

Figure 9 shows the relative speedup of overlapping L1 DL using linklist. The workload to overlap with DL is a loop performing accumulation of integer adds (INTADDs, shown on the x-axis), while the y-axis gives the relative speedup. Figure 9 is very similar to the theoretical prediction of L1 speedup curve given in Figure 8. It reaches its maximum of 45% while overlapping roughly 70 INTADDs.

When performing testing on real machines, a workload that pollutes the L2 cache must already have the L1 cache polluted. It is difficult to obtain the performance figure with a workload that overlaps with only the L2 cache (L2 DL). We thus focus on workloads that overlaps with L1-and-L2 (L1-L2) DL.

Figure 10 shows the relative performance result when overlapping with L1-L2 DLs. In stage 1, the curve reaches around 35% speedup at roughly 70 INTADDs. This agrees with our own measurement given in Figure 9 and it is the effect of mostly overlapping L1 DL. In stage 2, the curve maintains stabilities over 35% with maximum reaching very close to the 50% theoretical peak. This closely matches the L1-and-L2 prediction given in Figure 8 where a wide range of 35%+ relative performance is expected after stage 1.

### 6.4 Challenge with Real-World Applications

We give theoretical predictions on performance analysis which overlaps with various levels of cache. We verify this claim with micro benchmarks that can reach very close to the theoretical peak. These results are obtained under ideal conditions that (i)
there is no need to do checkpointing because the workload has no global side effect (similar to a pure function), and (ii) there is no failed speculation because the involved predictor can yield 100% prediction accuracy. However, such ideal situations may not hold under non-synthetic benchmarks on real machines.

In the future, we plan to investigate the feasibility of applying the control and data speculation transformations we introduced in this paper to real-world applications (e.g., the DL-intensive applications in the SPEC2000INT suite). We expect some major challenges. First, even with all checkpointing optimizations enabled, checkpointing overhead is non trivial and can’t be ignored. Second, the branch or value prediction’s success rate plays an important role because failed predictions will directly translate into failed speculation and triggers the recovery and retry overhead. Third, the compiler needs to find enough work that can potentially overlap with the identified DL. Finally, the compiler needs to recognize an ideal sweet spot to terminate speculative overlapping.

6.5 Summary

We present theoretical performance analysis that models speculative execution overlapping with various levels of DLs. We predict the relative speedup will be around 50% through the model and verify it with real synthetic benchmarks that can reach very close to the theoretical peak. The results are obtained on real machines under ideal speculative conditions. The encouraging results motivate us to do further exploration using real-world applications.

7. CONCLUSIONS AND FUTURE WORK

In this paper we present our discovery that level-2 DLs from cache-miss intensive applications are persistent across a wide variety of cache architectures and input data sets. Motivated by this persistence, we present compiler transformations dealing with both control speculation and data speculation. We conduct theoretical performance modeling that predicts around 50% relative speedup. Our study using synthetic benchmarks strongly supports this claim.

We plan to investigate speculative execution that overlaps with DLs on real-world benchmark applications (e.g., the SPEC2000INT suite). The DL-persistent nature that exists in these applications provides an ideal granularity to further explore speculative execution that can be enabled through compiler transformations.

The emergence of hardware transactional memory [27] provides ideal hardware acceleration for fine-grain checkpointing. We plan to capitalize on the reduced overhead to use it to implement fine-grain speculative optimizations such as tolerating DL latency. We also plan to pursue alternative client optimizations for compiler-based fine-grain checkpointing such as debugging support, and possibly as part of an optimized software transactional memory (STM) [17, 41].

8. ACKNOWLEDGEMENTS

This work is funded by support from both IBM and NSERC. Chuck is supported by an IBM CAS Ph.D. fellowship since 2007.09. The authors would like to thank the anonymous reviewers for their feedback and insightful comments. The authors would also like to thank Mihai Burcea for the resourceful discussions during development.

9. REFERENCES


Programming Languages and Operating Systems, 1992.


