Proceedings of the

Fifth Workshop on Multithreaded Execution, Architecture and Compilation

MTEAC-5

December 1, 2001 -- Austin, Texas
In Association with the
34th International Symposium on Microarchitecture

MTEAC-5

WORKSHOP PROGRAM

Session I:

Message-Passing for the 21st Century: Integrating User-Level Networks with SMT, Mike Parker, Al Davis, Wilson Hsieh, *University of Utah*.

A Binary Translation System for Multithreaded Processors and its Preliminary Evaluation, *Kanemitsu Ootsu, Takashi Yokota, Takafumi Ono, Takanobu Bab, Utsonomiya*.

The Predictability of Computations that Produce Unpredictable Outcomes, Tor Aamodt, Andreas Moshovos, Paul Chow, *University of Toronto*.

Session II:

Hierarchical Multi-threading for Exploiting Parallelism at Multiple Granularities, Mohamed M. Zahran, Manoj Franklin, *University of Maryland*.

Basic Mechanisms of Thread Control for On-Chip-Memory Multi-threading Processor, Takanori Matsuzaki, Hiroshi Tomiyasu, Makoto Amamiya, *Kyushu University*.

Maximizing TLP with Loop-Parallelization on SMT, Diego Puppin (*Massachusetts Institute of Technology*), Dean Tullsen (*University of California, San Diego*).

Session III: Keynote Address

Speculative Multithreading: From Multiscalar to MSSP, Guri Sohi, *University of Wisconsin, Madison.*

Session IV:

Branch Prediction in a Speculative Dataflow Processor, Bradley C. Kuszmaul, Dana S. Henry, *Yale University*.

A Study of Compiler-Directed Multithreading for Embedded Applications, Anasua Bhowmik, Manoj Franklin, Quang Trinh, *University of Maryland*.

Prefetching in an Intelligent Memory Architecture Using a Helper Thread, Yan Solihin, Jaejin Lee, Josep Torrellas, *University of Illinois, Urbana-Champaign*.

Session V: Keynote Address

Multithreading for Latency, John P. Shen, *Intel Corporation*.

Message-Passing for the 21st Century: Integrating User-Level Networks with SMT

Mike Parker, Al Davis, Wilson Hsieh
School of Computing, University of Utah

Abstract

We describe a new architecture that improves message-passing performance, both for device I/O and for interprocessor communication. Our architecture integrates an SMT processor with a user-level network interface that can directly schedule threads on the processor. By allowing the network interface to directly initiate message handling code at user level, most of the OS-related overhead for handling interrupts and dispatching to user code is eliminated. By using an SMT processor, most of the latency of executing message handlers can be hidden. This paper presents measurements that show that the OS overheads for message-passing are significant, and briefly describes our architecture and the simulation environment that we are building to evaluate it.

1 Introduction

The same VLSI technology forces that are driving processor interconnect are also having an impact on I/O architectures. As clock frequencies increase, high capacitance processor and I/O buses cannot keep pace. These buses, on and off chip, are being replaced by point-to-point links. I/O interfaces are starting to look much more like message-passing networks, as is evidenced by recent standards such as InfiniBand[25] and Motorola's RapidIO[27]. Communication over these point-to-point I/O networks can be viewed as low-level message-passing, where queries are sent to devices and responses are received from devices. Since technology trends force the hardware to use point-to-point links, there is an interesting opportunity to expose communication directly to user-level software through a message-passing interface. By looking at this opportunity from a systems point of view (from user-level software down to the hardware), we anticipate that we can dramatically reduce the costs for both processor-to-processor and processor-to-IO message-passing.

Our architecture for addressing this problem consists of the following combination of ideas:

- An SMT processor allows the overhead of message handlers to be effectively hidden.
- A network interface that supports user-level access can be tightly coupled to the CPU to avoid the overhead and latency of slower I/O buses. In addition, the network interface can directly dispatch user-level threads on the SMT processor, which eliminates OS involvement in the common case
- A message cache buffers incoming messages so that they can be accessed quickly by the processor, and acts as a staging area for outgoing messages.
- A zero-copy message protocol allows messages to be delivered directly to user-space without copying.

Not all of these ideas are new. For example, previous research has explored the use of user-level network interfaces[3,9,11,13,18]. However, this specific combination of features is unique, in that it exposes interrupts directly to user-level programs. The important aspect of our architecture lies in its support for user-level messaging (for both interprocessor communication and I/O) in a general-purpose operating system with small modifications to an SMT processor.

The combination of features in our architecture should reduce message handling overheads dramatically, without requiring gang-scheduling or forcing a change of the message notification model that is seen by the user-level software. The SMT processor, originally targeted to hide message latency,

makes it possible to overlap computation and communication without adding a secondary communication processor. The combination of a zero-copy protocol, a message cache, and user-level access to the network interface allows user code to communicate without the overhead of OS involvement or data copying. Finally, our integration of the network interface (NI) with the SMT allow the NI to communicate message arrival events back to the target thread without most of the overhead an interrupt-style notification would incur.

2 Message Notification Costs

Figure 1 shows how a message send and receive may look from a single node point of view on a machine that uses a kernel-mode network interface and traditional interrupts for message arrival notification. Sends, receives, and notifications all make passes through operating system code. Since the operating system code is unlikely to reside in the cache, these system calls result in cache misses.

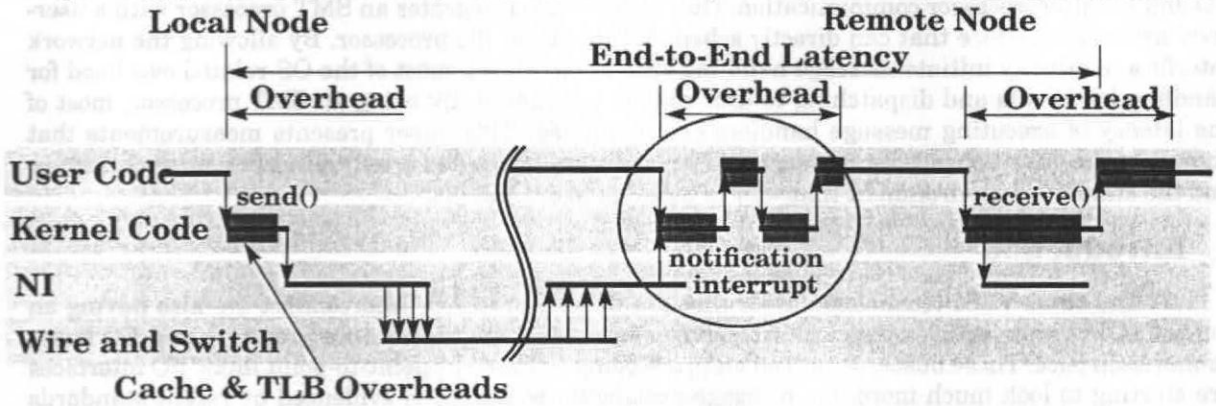


Figure 1: Anatomy of a message for a kernel-mode NI

User-level interfaces[3,9,11,13,18] and zero-copy protocols[5,7] significantly reduce the overhead of message sends and receives by eliminating operating system and copying overhead on the message send and receive sides. Notifications still have significant opportunity for optimization, as they remain the performance and scalability bottleneck in general multi-user environments. Polling for notifications consumes significant processor and memory resources, making them undesirable in a multiprogrammed system. Polling is especially poor for programs with irregular or unpredictable communication patterns. Interrupts in current architectures and operating systems are costly in terms of the number of processor cycles consumed to determine the cause of and handle the exception[12]. This makes them less than optimal for message notifications.

The components of an interrupt-style notification overhead include:

- processor pipeline flushing (due to the interrupt)
- serial instructions to get and save processor state
- cache and TLB misses to bring in OS code and data to determine the cause of the interrupt
- reading NI registers or data structures to determine which process should be notified
- posting the notification to the process via a signal or other such mechanism
- cache, TLB and context switch overhead to begin execution of the user-level notification (signal) handler
- a trap to return from the user-level handler back to the OS
- serial instructions to save processor state
- cache and TLB overhead to bring in OS code and data
- scheduler and context switch overhead to bring back in the original user process
- post kernel cache and TLB overhead to bring back in the user process's instructions and data

Using a refined version of Schaelicke's interrupt measurement work[22], we measured the overhead of servicing a network interrupt for a minimum sized packet. Under Solaris 2.5.1 on a 147-MHz Ultra 1, such an interrupt takes approximate 119 microseconds (17500 cycles) when user-level code is utilizing the entire L2 cache. The process of handling such an interrupt results in about 380 kernel-induced L2-cache misses. (Fewer misses may be observed in practice if the user-level code is not utilizing the entire L2 cache.) Assuming that each cache miss takes an average of approximately 270 ns to service[17], this accounts for about 103 microseconds or 87% of the interrupt processing time. The remaining 13% of the time is spent in flushing the pipeline after the interrupt and trap, carefully reading and saving critical processor state, querying the NI for information about the interrupt, and executing operating system code to determine how to deal with the interrupt. In addition to incurring the overhead of cache misses during an interrupt, the process that was running when the interrupt occurred could see up to another 380 L2 cache misses once it is re-scheduled after the interrupt to refill the cache with its working set.

L2-cache and TLB miss penalties unfortunately scale at memory speeds, as opposed to processor speeds. As a result, these overheads will become even more important as the memory gap widens. Optimizations to the OS and signalling system can reduce this overhead. However, reducing the number of cache misses and other overheads to get the OS penalty down below a few microseconds does not seem plausible. To make frequent notifications acceptable, the operating system's involvement must be significantly reduced or eliminated.

3 Architecture

Notifications only become the bottleneck when the rest of the message-passing system is appropriately tuned. This section describes the system architecture, showing how it is optimized for efficient messaging, describes how notifications are delivered to the user-level process without kernel involvement, and walks through the path a one-way message takes though this architecture. Figure 2 shows a block-level view of the architecture.

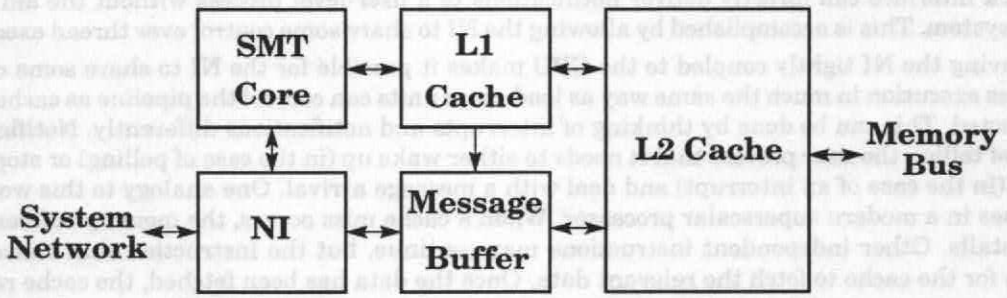


Figure 2: Block-level diagram of our architecture

3.1 Components of the Architecture

Using an SMT processor allows communication-related threads to run parallel to computation-based threads. Much of the overhead required to process sends, receives, and message arrival notification can be hidden by overlapping these functions with computation. Previous work has dealt with overhead by providing external communication processors[14,21]. These extra processors add to the overall latency and complicate the message-passing mechanism due to the additional overhead of communication between the computation and communication processor. The simultaneous nature of the SMT processor makes it possible to provide this overlap without requiring the use of an extra communications processor.

The message buffer acts as both a staging area for outgoing messages as well as a cache for incoming messages[23]. Messages may be composed directly in the message buffer for user-program written (PIO-style) transfers, or fetched from user memory for DMA-style transfers. Buffering outgo-

ing messages in the message buffer allows send data to be prefetched from memory and buffered before going out on the network. This buffering reduces the probability that the network will need to be stalled, tying up network resources, while waiting for outgoing message data to be supplied by the local memory subsystem.

Incoming messages are placed in the message buffer. The message buffer acts as a cache for incoming message data. As a message arrives, the message buffer invalidates corresponding cache lines in the L1 and L2 caches. Misses in the L1 cache result in concurrent lookups in both the L2 cache and the message buffer. In this way, the message buffer is similar to a victim cache to the L2 cache. When there is a cache hit in the message buffer, data is supplied directly to the L2 cache. This cache has a triple effect. First, it reduces overhead at the memory interface by saving the data two trips across the memory bus. Second, it keeps the data near the CPU, where it can be provided quickly on demand, thus reducing the overall end-to-end latency. Third, having a separate message cache avoids polluting the cache hierarchy because the processor's working set is not evicted by incoming messages.

A user-level accessible NI is used to reduce send and receive overhead. Having the network interface on the same die, possible in the System on a Chip (SoC) era, opens up possibilities to more tightly integrate it with the processor core, further reducing overhead and latency. Having the NI on die gives the processor access to it on a per cycle basis. This close coupling further reduces the overhead in getting information to and from the NI. Message sends and receives do not have to go out over slow and inefficient I/O buses. A zero-copy protocol[5,7] is used to eliminate copying overhead for received messages. The combination of user-level access to a closely coupled NI and the zero-copy protocol allow for efficient sends and receives.

3.2 User-level Notifications

Part of the inefficiency of interrupt processing is due to the legacy view that interrupts are expected to be infrequent. In a fine-grained message-passing environment that uses interrupts for notifications, this is not the case. One of the contributions of this work is to provide a mechanism whereby the network interface can directly deliver notifications to a user-level process without the aid of the operating system. This is accomplished by allowing the NI to share some control over thread execution.

Having the NI tightly coupled to the CPU makes it possible for the NI to share some control over process execution in much the same way as load-store units can control the pipeline as cache misses are detected. This can be done by thinking of interrupts and notifications differently. Notifications are a way of telling the user process that it needs to either wake up (in the case of polling) or stop what it is doing (in the case of an interrupt) and deal with a message arrival. One analogy to this would be cache misses in a modern superscalar processor. When a cache miss occurs, the memory reference instruction stalls. Other independent instructions may continue, but the instruction that caused the miss waits for the cache to fetch the relevant data. Once the data has been fetched, the cache returns it to the processor core, and the processor again places priority on the memory reference and dependent instructions.

This basic idea can be extended to message arrival notifications. When a process reaches a point where it needs a message arrival notification, it can tell the hardware what specific process state to change (i.e., program counter or runability) when an arrival occurs. It can then continue processing until either the notification occurs, or it runs out of things to process. When the notification finally takes place, the hardware can redirect the user-level software to work on processing the message arrival.

To give the NI shared control over user processes, three mechanisms are available in our architecture.

- If a thread wishes to be notified when a message arrives, it can set a lock in a hardware synchronization lock table[24]. The NI can clear the lock bit upon message arrival, which releases the process to run again. If the associated thread is not currently in the CPU, the OS

receives the notification, and sets the appropriate hardware state to notify the thread the next time it is scheduled.

- Just as an interrupt causes a current processor to asynchronously branch into a kernel level interrupt handler, the NI can cause a running user-level process to asynchronously branch to a notification handler.
- Upon message arrival, the NI can schedule a new thread on the SMT with a previously setup context. This new thread starts in either an unused context on the SMT or it can evict a running thread, according to OS policy. If no contexts are available, the NI would notify the OS, so that it could create the context and schedule it to run at a later time.

3.3 The Journey of a Message

Figure 3 shows how a message may look in our architecture.

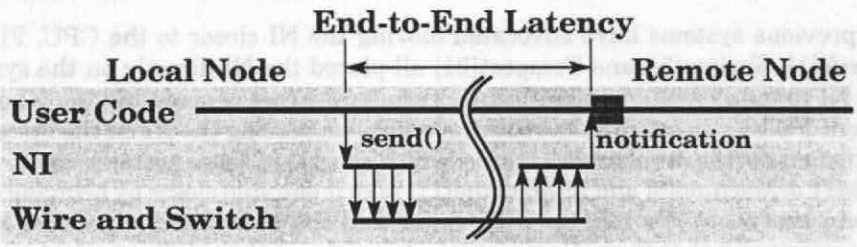


Figure 3: Anatomy of a message in our architecture

A user-level program that wants to send a message to a process on a remote processor first composes the message to be sent. Message control information is written into the message buffer. It includes information on how the message is to be handled on the remote end (where it will be placed and whether to notify the receiving process), a small amount of user-defined meta-data, and an optional local pointer to message data. The user process then directs the NI to send the message. The NI begins to prefetch message data from local memory into the message buffer if required, fills in information such as routing information, and begins streaming the message onto the wire. On the receive side, the message is placed directly in the address space of the receiving process by the NI. While the message is arriving, the NI places the message into the message buffer. As it fills cache lines in the message buffer, it acquires ownership of those cache lines though the L2-cache interface and sets up appropriate cache tags in the message buffer.

When an entire message arrives, the NI fills in a notification structure in the user process's memory space. It then determines which method should be used to notify the corresponding process of the message arrival. In the case of a blocked process waiting on a message arrival, the NI clears a lock bit in a synchronization table, which makes the corresponding thread runnable. The user-level process begins executing and handles the incoming message. In the case of an asynchronous branch or user-level interrupt, the SMT switches to an alternate program counter and stack. User-level code is then responsible for saving any of its own state as necessary before handling the notification. Finally, in the case of a created thread, the NI gives the SMT core minimal context, including a program counter, stack pointer, and a pointer to the notification structure. The SMT core begins executing at the given program counter.

In summary, we have presented an architecture that significantly reduces send, receive, and notification overhead. We have presented three separate user-level notification mechanisms, and have walked through the path a message takes in our architecture. The key features of the architecture are the following: SMT processors hide and tolerate message overhead and latency, send and receive overhead is reduced by a user-level network interface combined with efficient protocols, and notifications can be delivered directly to user-level without the overhead of an operating system.

4 Related work

Simultaneous Multithreading (SMT) architectures[8] have begun to see commercial attention.The SMT processor is targeted to tolerate latency and hide overhead by allowing one thread to process overhead or wait for a long latency operation while the execution of independent threads continues. SMT architectures promise the ability to simultaneously take advantage of both ILP and thread-level parallelism within a single processor core. This architecture helps pave the way to more efficient communication and synchronization of threads.

Dean Tullsen et al.[24] shows how extra lock and release hardware can be introduced to provide fine-grained synchronization for threads within the SMT processor. This efficient locking mechanism allows one thread to block on a hardware semaphore and be released by another co-operating thread quite efficiently. One of the suggested notification primitives in this paper extends some of the control over this hardware locking table to external events, such as message arrival notifications from the NI.

Several previous systems have advocated moving the NI closer to the CPU. Flash[14], Avalanche[23], Alewife[1], Shrimp[3], and Tempest[21] all placed the NI directly on the system memory bus. Moving the NI to the system bus significantly reduces the cost of accessing the NI over accessing it on a less efficient I/O bus. In addition to reducing overhead, placing the NI on the system bus allows these systems efficient access to coherency traffic, which several of these systems use to an additional advantage. The MIT J-Machine[6] and M-Machine[11] take it one step closer by bringing the NI directly onto the custom processor. Alewife, the J-Machine, and the M-Machine also have an interesting characteristic in common in that they all use a thread model or a thread-like model to deal with communication. Alewife uses a modified SPARC processor in an unconventional way to implement these threads.The J-Machine and M-Machine both build a custom processor to get the desired thread behavior. SMT processors now seem to be a natural way to achieve effective functionality of these machine with only minor modifications to CPU structure.

In many ways the architecture in this paper is similar to the M-Machine. Both take advantage of thread capable processors to hide message overhead, and both have forms of automatically dispatching threads when a message arrives. Our architecture differs from the M-machine in the following ways. Messages are received directly into a users address space via hardware, eliminating the need for trusted message handlers. Incoming messages are placed into a message buffer, or message cache, to avoid pollution of the processor's cache hierarchy. As a part of our work, we are evaluating the usefulness of this message-passing architecture both in the context of parallel processing and in the context of network-based IO. Finally this architecture is built upon modifications of upcoming SMT architectures.

Avalanche[23] placed the network interface on the system bus, keeping it close to the processor. This allowed it to participate in coherency traffic, and thus maintain a local network cache. The local cache enables the Avalanche network interface to supply network data to the processor more quickly than main memory. In addition, it avoids the overhead of wasting system bus bandwidth to transfer message data across the system bus twice; once on the way to main memory on message arrival, and once on the way back to the processor when the message is consumed.

For Hamlyn[5] Wilkes proposed sender-based protocols to reduce overhead. Having the sender manage its destination buffers implies that data can be easily and effectively received directly into the receiver's process space. Avalanche also used a sender-based messaging protocol (DDP)[7] to reduce overhead. Sender-based protocols allow simple and efficient hardware to place incoming messages directly into the receive process's address space. This avoids kernel involvement on receives. Unlike Hamlyn and DDP, the sender-based portion of the protocol in this work uses virtual addresses in combination with an NI TLB to remove restrictions on receive buffers.

Active Messages[10] embed a message handler in the header of a message. When a message arrives, the message handler is executed to handle the payload of the message. Though it is not specifically a goal of this work, the architecture described here would support Active Messages rather well.

A thread waiting for a message could immediately jump to the handler code in the header without the penalty of an interrupt and without interfering with the currently running thread. If messages are received directly into a message cache, then this handler code could potentially execute directly out of the message cache, also saving the cache overhead of bringing in conventional handler code. Illinois Fast Messages[20] is effectively a platform independent implementation of Active Messages.

U-Net[9] reduces communication overhead and latency by virtualizing the network interface. The local process communicates with the network interface by placing and picking up message packets from per-process send and receive queues. U-Net suggests placing a TLB in the network interface to avoid the added restriction of fixed pinned pages. The architecture in this paper also gives the NI access to a TLB to allow sends and receives to be handled in user-space.

5 Conclusion

SMT allows important computation to continue while interprocessor communication and I/O processing and communication overhead is handled in the background. Since message latency is similar to memory latency, one way of viewing this work is using an architectural technique for hiding memory latency to hide message latency. Conversely, we can view our work as generating more parallelism for SMT processors from I/O and parallel workloads.

To evaluate this architecture, we are extending LRSIM[22] to accurately model an SMT processor and adding a model of our network interface. LRSIM is based on RSIM[19], and has already been extended to include accurate I-cache, memory and I/O architecture models. The simulator includes a fairly complete NetBSD based kernel that will be extended to handle the SMT processor (all kernel operations are fully simulated). The simulator runs unmodified Solaris binaries. For our evaluations, we will model a 2-5 GHz 4-8 thread SMT that can issue 8-16 instructions per cycle. L1 instruction and data caches will be 32KB to 128 KB each, and the L2 Cache will be 4-16MB. The system network bandwidth modeled will range from 4Gb/s to 32Gb/s. Disk controllers, LAN interfaces (i.e., Ethernet), and other I/O devices will hang off the system network. There are a few open issues in our design:

- It remains unknown how much send side buffering will be optimal. Too little buffering could lead to too many bubbles in the network fabric. Too much could lead to increased message latency. For the purposes of this design, the amount of buffering will be user configurable.
- The point at which DMA becomes more efficient than PIO needs to be characterized for this architecture. Though it has been characterized for previous systems, the break-even point on our architecture may be different as PIO can overlap computation on the SMT processor.
- The relative performance of each of the notification mechanisms needs to be characterized.

The results from the simulations will be compared with existing and proposed systems to assess the benefits of our architecture.

Acknowledgements

We thank Lambert Schaelicke for his help in understanding and extending his interrupt measurement techniques. We also thank Lambert, John Carter, Katie Parker, and Greg Parker for help in reviewing this paper. This work was sponsored in part by DARPA and AFRL under agreement number F30602-98-1-0101 and DARPA Order Numbers F393/00-01 and F376/00. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official polices or endorsements, either express or implied, of DARPA, AFRL, or the U.S. Government.

References

[1] Anant Agarwal, et al. The MIT Alewife Machine: Architecture and Performance. In Proceedings of the 22nd Annual ISCA, 1995, pp. 2-13.

[2] Mark Birnbaum and Howard Sachs. How VSIA Answers the SoC Dilemma. IEEE Computer, 32(6):42-50.

[3] Matthias A. Blumrich, et al. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In Proceedings of the 21st Annual ISCA, April 1994, pp. 142-153.
[4] Doug Burger. Billion-Transistor Architectures. IEEE Computer, 30(9):46-48, September 1997.
[5] Greg Buzzard, et al. Hamlyn: a high-performance network interface with sender-based memory management. HP Laboratories Technical Report HPL-95-86, August 1995.
[6] William J. Dally, et al. Retrospective: The J-Machine. In 25 Years of ISCA - Selected Papers, 1998, pp. 54-58.
[7] Al Davis, Mark Swanson, and Mike Parker. Efficient Communication Mechanisms for Cluster Based Parallel Computing. Communication, Architecture, and Applications for Network-Based Parallel Computing, 1997, pp. 1-15
[8] Susan J. Eggers, et al. Simultaneous Multithreading: A Platform for Next-Generation Processors. IEEE Micro, 17(5):12-19, October 1997.
[9] Thorsten von Eicken, et al. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In Proceedings of the 15th ACM SOSP, December 1995, pp. 40-53.
[10] Thorsten von Eicken, et al. Active Messages: A Mechanism for Integrated Communication and Computation. In Proceedings of the 19th Annual ISCA, 1992, pp. 256-266
[11] Marco Fillo, et al. The M-Machine Multicomputer. In Proceedings of the 28th Annual International Symposium on Microarchitecture, 1995, pp. 146-156.
[12] Andrew Gallatin, Jeff Chase, and Ken Yocum. Trapeze/IP: TCP/IP at Near-Gigabit Speeds. In Proceedings of 1999 USENIX Annual Technical Conference, FREENIX Track, June 1999.
[13] Dana S. Henry and Christopher F. Joerg. A Tightly-Coupled Processor-Network Interface. In Proceedings of the 5th International ASPLOS, October 1992, pp. 111-122.
[14] Jeffrey Kuskin, et al. The Stanford FLASH Multiprocessor. In Proceedings of the 21st Annual ISCA, April 1994, pp. 302-313.
[15] Richard P. Martin, et al. Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture. In Proceedings of the 24th Annual ISCA, June 1997, pp. 85-97.
[16] Doug Matzke. Will Physical Scalability Sabotage Performance Gains? IEEE Computer, 30(9):37-39, September 1997.
[17] Larry McVoy and Carl Staelin. lmbench: Portable Tools for Performance Analysis. In Proceedings of the USENIX 1996 Technical Conference, January 1996, pp. 279-294.
[18] Shubhendu S. Mukherjee and Mark D. Hill. Making Network Interfaces Less Peripheral. IEEE Computer, 31(10), October 1998, pp 70-76.
[19] V. S. Pai et al. RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors. In Proceedings of the 3rd Workshop on Computer Architecture Education, 1997.
[20] Scott Pakin, Vijay Karamcheti, and Andrew A. Chien. Fast Messages (FM): Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors. IEEE Concurrency, vol 5, no. 2, April-June 1997, pp. 60-73.
[21] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In Proceedings of the 21st Annual ISCA, 1994, pp. 325-336.
[22] Lambert Schaelicke. Architectural Support for User-Level I/O. Ph.D. Dissertation, University of Utah, 2001.
[23] Mark Swanson, et al. Message Passing Support in the Avalanche Widget. Technical Report UUCS-96-002, University of Utah, March 1996.
[24] Dean M. Tullsen, et al. Supporting Fine-Grained Synchronization on a Simultaneous Multithreading Processor. In Proceedings of the 5th HPCA, January 1999.
[25] InfiniBand Trade Association. http://www.infinibandta.org.
[26] International Technology Roadmap for Semiconductors. Semiconductor Industry Assoc., 1998.
[27] Motorola Semiconductor Product Sector. RapidIO: An Embedded System Component Network Architecture. February 22, 2000.

A Binary Translation System for Multithreaded Processors and its Preliminary Evaluation

Kanemitsu Ootsu, Takashi Yokota, Takafumi Ono, and Takanobu Baba

Department of Information Science, Faculty of Engineering, Utsunomiya University
7–1–2 Yoto, Utsunomiya-shi, Tochigi, 321–8585 Japan.
phone&fax: +81–28–689–{6284, 6290} e-mail: {kim, yokota}@is.utsunomiya-u.ac.jp

Abstract

Thread level parallelism (TLP) is a key technology to coming generation of high performance processors. Although it provides higher processing capability, the loss of compatibility with existing processors is a crucial issue. This research is motivated by the following two points: (1) TLP requires multithread programming which is rather difficult for ordinary programmers, or complexed compilation technologies that can exploit multithread parallelism, and (2) existing binary codes should be executed efficiently on multithreaded processors. In this paper, we first propose a binary translation system, that translates existing binary codes to multithreaded ones and optimizes them dynamically during execution. The system inputs the original binary codes and translates them to internal RTL representation. It analyzes the structure of the program and applies multithreading to loop bodies in a thread pipelining manner. A pilot binary translator, that is a part of the proposed system, was built for the sake of preliminary evaluation. Evaluation results illustrate effectiveness of the system.

Keywords: *binary translation, thread level parallelism, multithreading, thread pipelining, run-time optimization.*

1 Introduction

Thread level parallelism (TLP) is one of the most promising key issue to high-performance processor architecture in the next generation. Present state-of-the-art technologies, such as superscalar, out-of-order, speculative execution, and value prediction, are successful in keeping continuous compatibility with conventional processor's instruction set architecture (ISA). And even in different architectures, i.e., VLIW (very long instruction word), a sort of binary translation technology is adapted so that the processor looks like conventional ISA from users view.

On the other hand, TLP essentially requires multithreaded machine codes to exploit full ability of the architecture. Because of the discontinuity of binary code compatibility, we can find the following two problems.

First, who (what) can produce multithreaded codes? Most programmers are not so skilled to make their application fully multithreaded. To this problem, further compiler technologies are required for automatic multithreading of an original application program in the near future. Second one is rather practical, i.e., we should abandon plenty of existing (single-thread) binary codes if their source codes are not available. The single-thread binary codes could run on multithreaded processors, although, they can receive no performance gain from TLP. Thus these two problems prevent TLP from being widely accepted. As a realistic solution to the problems, we focus our approach on the efficient reuse of existing binary codes on a multithreaded architecture that exploits rich TLP.

In this paper, we propose a binary translation and run-time optimization system.[14, 15] We first introduce binary translation technology that translates existing single-thread code to multithreaded ones. Source binary codes are analyzed, decomposed into threads, and then mapped onto the target architecture.

We then introduce dynamic (run-time) optimization of the translated codes. Because of lack of source code information, static analysis has some limitations: e.g., distinction of instruction words and data is not clear, and the target addresses of indirect jumps remain unknown.

The rest of this paper is organized as follows. We first discuss design principles to realize our ideas in Section 2, where we make some basic assumptions and discuss system requirements. Then we propose a binary translation and optimization system in Section 3, where basic components and their functions are discussed. Section 4 describes static optimizer in detail and Section 5 shows the preliminary evaluation. Section 6 presents related works which aim at binary translation or optimization. This section clarifies the standpoint of the proposed system and thus its unique features. Finally, we conclude this paper in Section 7.

2 Design Principles

2.1 Multithreading by the Thread Pipelining Model

In order to run a single-thread binary code efficiently on a multithreaded processor, logical structure embedded within the source binary code is extracted and the program is restructured to a set of threads. For the following discussion, we make an assumption on the target multithreaded architecture.

Needless to say, single-thread code follows a sequential programming manner. Although the ideal objective is to exploit all possible parallelism inherent in the program, it is not realistic for binary code inputs. We have started discussion with a simple idea: we payed our attention to *loop* structures.

The idea is very natural. A programmer tends to follow a sequential program (thread) depicted in a one-dimensional space. In such situations, a parallel structure is expressed as a loop. In other words, a loop structure contains inherent parallelism. Thus, it is appropriate that each iteration in the loop is converted to a thread. In many cases, a loop structure contains many iterations, and thus enables us to exploit the maximum parallelism.

Thread pipelining model[1] best fits to our purpose described above. Figure 1 shows partial structure of the multithreaded processor based on the model. Each thread generated by binary translation is mapped to a thread execution unit in order. Communication unit and Memory Buffer handle inter-thread control and dependencies, respectively.

Basically, each iteration corresponds to a thread, and threads are executed in a pipeline manner. Figure 2 illustrates the thread pipelining[1]. Each thread consists of four stages: Continuation, Target Store Address Generation (TSAG), Computation, and Writeback.

The Continuation stage introduces loop variables and necessary data so as to be used in the thread. After the Continuation stage completes, the succeeding thread is invoked. The TSAG stage checks dependencies of shared data between threads. Addresses of shared data are notified to Memory Buffer, which detects access dependencies between threads by monitoring addresses. The Computation stage does the peculiar calculation assigned to the thread. After completing the Computation stage, a thread terminates its life in the Writeback stage. The Writeback stage cannot be started until the preceding threads' Writeback stages are completed.

2.2 Single- to Multi-thread Binary Translation

As described above, we have introduced a thread pipelining concept to our system. This pipelining is a fundamental requirement in the proposed system. It is totally different from existing translation systems in that it converts single-thread code to multithreaded one whereas others translate to single-thread. In other words, ordinal translators don't change program structure, although, our system should arrange the analyzed program structure to achieve the best fit to the thread pipelining model.

Source binary code is first analyzed with its logical structure, and the structure is re-organized following the thread pipeline manner. Then, target machine code is generated. During analysis of program structure (i.e., control- and data-flow) and re-configuration phase, abstract representation is required. We have introduced RTL (register transfer level) representation, which is used internally in the translator. Actual translation procedure is displayed in Section 3.

2.3 Necessity of Run-time Optimization

In principle, binary translator converts a loop structure to a set of threads. This requires accurate analysis of program structure, however, the translator could not find full information because of lack of source code information. For example, indirect branch hides the target address of iteration. So, some loop structures may

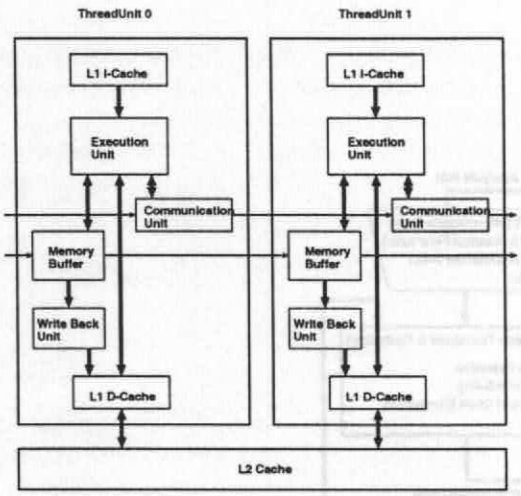


Figure 1: Thread Execution Units

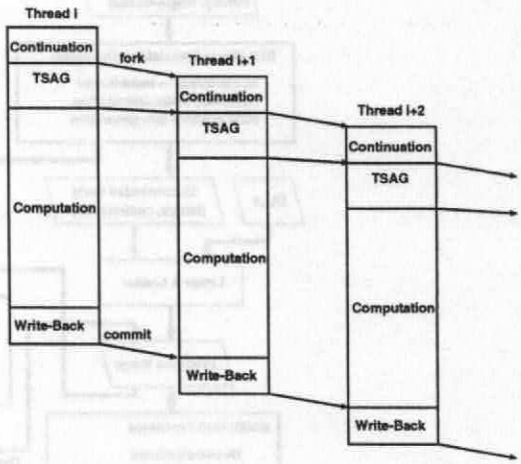


Figure 2: Thread Pipelining Model

remain unfound even when the analysis phase is completed. These loop structures cannot be converted to multithreaded code, and they do not appear until the translated codes run.

Thus, to exploit full parallelism of the program, it is necessary that behavior of the program is monitored and that '*hot*' portion is translated to multithreaded code. This methodology is a kind of run-time optimization. Like the binary translation introduced in the previous section, run-time optimization requires re-structuring of the program (i.e., converting single-thread code to multithreaded one) where other run-time optimization techniques do not essentially affect program structure.

3 The Binary Translation and Optimization System

3.1 Systems Logical Structure

As discussed above, in order to execute existing binary codes on next-generation multi-threaded processor, the system requires following two phases: (i) binary translation and static optimization and (ii) run-time optimization. Figure 3 illustrates the configuration of the proposed system.

In Figure 3, STO (Static Translation and Optimizer) executes the phase (i) and DTO (Dynamic Translation and Optimizer) performs (ii). The figure includes Multithreaded Processor, whose basic architecture is described in Section 2.1 and Figure 1. The processor's basic ISA is not limited to some specific architecture since the original binary codes may be translated according to the target architecture by STO and DTO.

STO inputs the sequence of the source binary codes and translates them to the target binary code. If the program requires dynamic linked libraries (DLLs), STO prepares the necessary libraries and links. Resulting executable binary image is put into the main memory and the processor executes the executable.

During the processor executes the translated binary code, the behavior of program is monitored. We introduce profiling techniques for monitoring. We assume that the processor has additional mechanisms that reduce profiling overheads.

DTO uses the profiling information and observes actual behavior of the application program. When detecting a buried 'hot' loop, it begins binary translation and optimization. It substitutes the single-thread 'hot' part by the translated multithreaded code, and thus accelerates total execution.

3.2 Static Translation and Optimization (STO)

As shown in Figure 3, STO inputs the source binary code and translates it to multithreaded code. Once STO reads the source binary code, the code is translated into an internal representation. The representation is, in principle, abstracted in a machine independent RTL (register transfer level) form. The internal representation

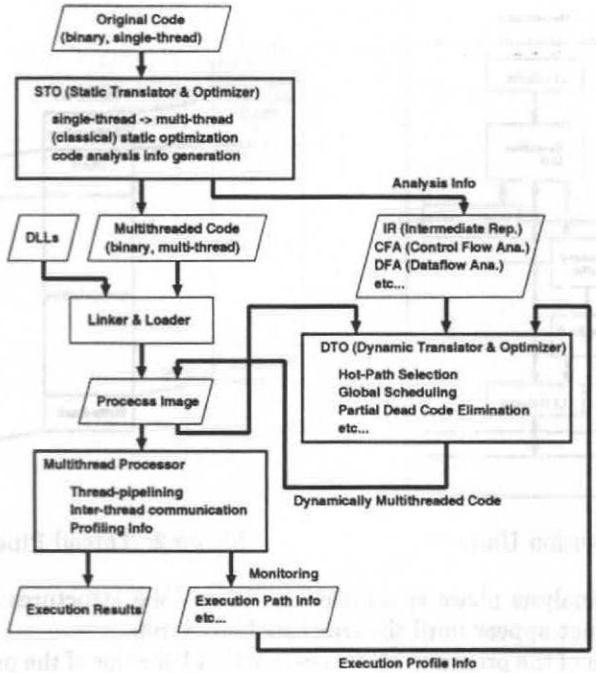


Figure 3: Proposed System Diagram

enables STO to do powerful optimization as taken in ordinal compilers/optimizers. In the proposed system, STO generates threads in the internal representation level.

Basically, STO works before the program runs. Its mission is to prepare translated binary code for the multithreaded processor before the program is started, however, it could not complete the translation. The reason is that no clear distinction is made between instruction code and data and that run-time information is buried. For example, an indirect jump operation hides its branch target address and thus prevents further analysis. Another example is the self-modifying code that determines its own execution code at run-time, so STO cannot know exactly what is to be done in the program. The remaining translation should be done at run-time and DTO handles it.

During analysis of input binary code, STO acquires useful information: code analysis information, control- and data-flow information. DTO does full use of these information. This reduces overheads in run-time optimization. STO inserts profiling codes so that DTO can collect proper information at low cost.

3.3 Dynamic Translation and Optimization (DTO)

DTO's major objective is run-time optimization (Figure 3). Unlike STO, DTO runs concurrently with the execution of application. It is invoked at proper intervals during application execution. DTO collects profiling information and monitors the program behavior. After detecting a hot-path, DTO arranges global scheduling, eliminates redundant codes, and applies possible optimization methods[2]. Then, DTO substitutes the original code to the optimized one.

Profiling codes are not removed by the DTO optimization. This means that profiling continues until the application is terminated. So DTO can apply further optimization incrementally and it can follow the change of program behavior. The DTO approach is similar to profile-guided compilation[3]. Since DTO can collect more detailed information, it should achieve deeper optimizations.

Source binary codes may contain self-modifying codes. STO cannot handle such codes since actual codes are determined at run-time. Thus, DTO should provide similar functions that STO does: i.e., the series of binary translation and optimization processes. Actually, input of source binary code, translation to internal representation, and control- and data-flow analysis should be processed by DTO.

4 Binary Translation Method

We have built an experimental binary translation software in order to estimate the effectiveness of the proposed system. The pilot translation system is to be a part of STO in the proposed system. This section introduces binary translation methods employed in the pilot system.

4.1 Basic Algorithm

As introduced in the previous section, the binary translator inputs binary codes and outputs multithreaded one. The translator performs the following steps:

(1) inputs source binary code and translates to internal representation,
(2) determines basic blocks, analyzes control-flow, and detects loop structure,
(3) analyzes data-flow, detects loop variables and inter-loop dependencies,
(4) converts loop structure to multithread codes in a thread pipelining fashion, and
(5) generates target machine code from internal representation.

The translator reads the source binary code. It begins code analysis from the starting address specified in the binary code. Input code is translated into the internal representation in order.

The internal representation categorizes instructions into six groups: alu operation, inter-register transfer, jump, branch, load/store, and other operations. Each instruction category has its unique operand expression. Figure 4 shows a part of instruction stream converted into the internal representation. The internal representation forms a list structure. Once the input binary codes are read and translated into the representation, succeeding processes, (2) to (5), are performed on the representation.

In the step (2), the translator determines basic blocks and analyzes control flow. It seeks back-edges, i.e., backward jumps/branches, in the internal representation. A back-edge is an important hint to mine a loop structure. The translator trys to find a path from the target address of a back-edge to the back-edge itself. If the path exists, it constitutes a loop structure.

In the step (3), the translator analyzes data-flow in the loop structure. It presumes loop variables used in the loop. The present pilot system finds the loop variables by increment of integer variables. The translator can analyze multiplexed loops.

The translator modifies the internal representation according to the result of multithreading operation (in the step (4)). Step (5) generates the target machine codes from the internal representation.

Next, we will explain step (4), the key part of the translator, in more detail.

In order to exploit sufficient parallelism by multithreading, we have found the following two requirements: (i) an interval of thread invocation should be shortened, and (ii) the synchronization time in resolving dependency should be reduced.

To solve (i), the Continuation stage prepares loop variables used in the succeeding thread. The values of loop variables are computed in the preceding thread. A newly created thread can start the execution of its loop body.

To reduce synchronization overheads due to inter-thread dependency (ii), the translator tries to move 'load' and 'store' instructions of shared data backward and forward in the Computation stage, respectively.

After the Continuation stage, addresses of inter-thread dependent data are registered in Memory Buffer in the TSAG stage. Execution of the consecutive TSAG stages cannot be overlapped since the stage determines shared data. Thus, at the entrance of the stage, the processor waits for completion signal from its preceding thread, and at the exit of the TSAG stage it sends completion signal to its succeeding thread. These inter-thread communications are handled by Communication Unit, shown in Figure 1.

Most calculations in the original loop body are executed in the Computation stage. Inter-thread dependencies are registered at the TSAG stage and Memory Buffer monitors all memory accesses. It handles inter-thread synchronization in a producer-consumer manner.

The Writeback stage writes calculation results onto memory. Since the resulting data should be stored in the semantic order, the stage cannot be overlapped between threads. Thus at the start-point of the stage, the processor should wait for a termination signal from its predecessor. After completion of this stage, the thread terminates.

Following the thread pipelining model, as shown in Figure 2, each thread consists of four stages. Three out of four stages cause overheads and only Computation stage performs the peculiar calculation in the iteration. Thus, to exploit sufficient performance on this model, the Computation stage should be long enough to hide overheads caused in other stages. We have introduced a loop unrolling technique as a solution. The succeeding section discusses the effectiveness.

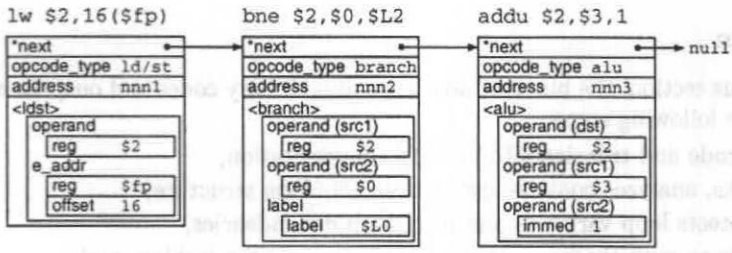


Figure 4: Internal Representation Example

4.2 Translation Example

Figure 5 illustrates a simple example of translation. The instruction stream, listed in the left side, is a part of the source binary code (disassembled for display use). In our pilot translator, only a hot-path (i.e., loop) is translated to multithreaded code as described in Section 4. The right side list shows the translated output of the hot-path.

To avoid complexity in evaluation processes, the original ISA is extended by adding several thread control instructions so that the codes can be run on the target multithreaded architecture.

A thread pipeline begins with **bstr** instruction. In the Continuation stage, the thread calculates the loop variable used in its succeeding thread from its own variable (address `($fp+16)` in Figure 5), stores the result by **sttsw** instruction, and then generates the succeeding thread by **lfrk**. Note that a loop variable is accessed via `($sp-8)` in this example.

In the TSAG stage, dependent address of `($fp+48)` is registered to Memory Buffer by **altsd** instruction. For proper synchronization among neighboring threads, **wtsagd** and **tsagd** instructions are used.

We can find a load instruction that fetches data from `($fp+48)`. When the instruction is executed, Memory Buffer detects the memory access and execution stalls until the preceding thread updates the data.

In the Writeback stage, **estr** instruction writes calculated data into memory and the thread terminates.

5 Preliminary Evaluation

5.1 Experimental Translation System

To evaluate the basic idea of binary translation to multithreaded codes and their optimization, we have built a pilot translation system. Figure 6 shows the block diagram. The objective of the system is to evaluate the STO functions which was described in Section 3.2. MultiThread Code Generator in Figure 6 follows all the translation steps (1) to (5) described in Section 4.1.

In Figure 6, MultiThread Code Generator translates source binary code to multithreaded one. Hot-path is manually guided to the translator for evaluation purpose. Resulting partial binary code is merged into the original (single-thread) binary code by **Binary Patcher**. Binary Patcher removes hot-spot codes from the original binary code, and inserts translated multithreaded code. Thus the hot-path is executed in the thread-pipelining manner and the rest of the program is executed in a single thread. At this timing, necessary run-time libraries are linked. Then, the prepared multithreaded binary code is executed in a simulator.

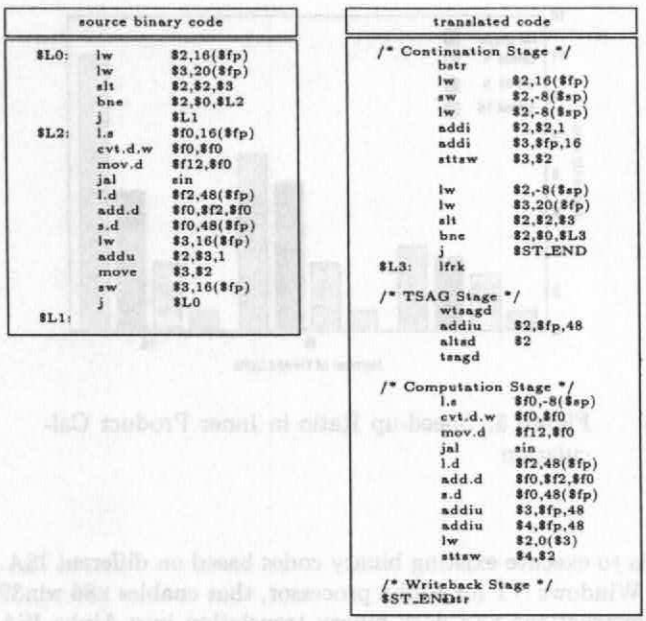


Figure 5: A Simple Example of Binary Translation

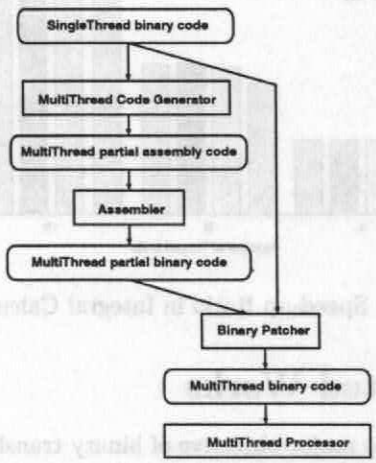


Figure 6: Block Diagram of the Pilot System

5.2 Evaluation Environment

We have assumed that the target multithreaded processor follows the architecture of SIMCA[4]. SIMCA is a simulator based on thread pipelining model and matches to our evaluation purpose.

Original binary codes are compiled by *gcc* cross compiler for SIMCA. The compiler's version is 2.7.2.3 and "`-O2`" option is applied. Application programs are (a) integral calculation in a '*sin*' trigonometric function using a trapezoidal equation and (b) inner product calculation.

Performance was measured as execution cycles of the hot-path by using the SIMCA simulator. Original binary code was executed on SIMCA and the number of execution cycles of the hot-path was measured. Similar evaluation was done for the translated code. By comparing the number of execution cycles, speed-up ratio was calculated.

In this preliminary evaluation, the number of thread units were assumed to be 4, 8, and 16. Furthermore, the loop-unrolling technique was applied to each application program; the measured unrolling factors were 4, 8, and 16 and no unrolling was measured for comparison purpose.

5.3 Evaluation Results

Figures 7 and 8 illustrate evaluation results for integral and inner product calculation applications, respectively.

In the integral calculation (Figure 7), the system gains performance linearly to the number of thread units. Unrolling factor does not affect the performance except 'no unroll' case.

In the inner product calculation (Figure 8), we can find that speed-up ratio is limited by unrolling factor. In 'no unroll' case, speed-up ratio is around 0.9 in spite of the number of thread units. We can find the similar phenomenon in the 'unroll 4' case. In the 'unroll 8' case, speed-up is achieved when 8 thread units are used. However, the performance saturates in the 16 thread units case. We can recognize linear speed-up in the 'unroll 16' case.

The integral calculation contains many operations enough to hide thread pipelining overheads. This leads near-linear speed-up according to the number of thread units.

On the other hand, the inner product calculation contains less operations than the integral calculation. Thus thread pipelining overheads could not be hidden unless sufficient loop-unrolling is applied.

These results reveal that efficiency in thread pipelining heavily depends on the grain size of calculation.

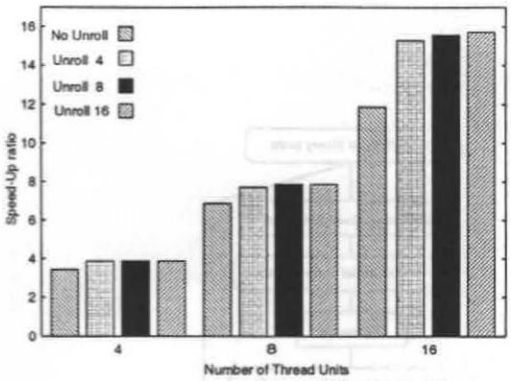


Figure 7: Speed-up Ratio in Integral Calculation

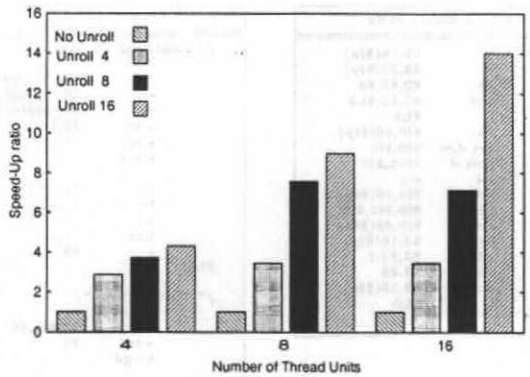


Figure 8: Speed-up Ratio in Inner Product Calculation

6 Related Works

In general, the major objective of binary translation is to execute existing binary codes based on different ISA.

FX!32[5] of Compaq is a translation subsystem in Windows NT for Alpha processor, that enables x86 win32 codes to run on Alpha platforms. It emulates x86 instructions and does binary translation into Alpha ISA codes. During idle time, The binary translation is executed with profiling results from the preceding emulation. Once the code is translated, the resulting native code is executed and earns high performance.

DAISY[6] of IBM translates well-used ISA codes, such as PowerPC and x86, so that programs run on the original VLIW processor. The system exploits instruction level parallelism (ILP). It does no emulation.

Transmeta's Crusoe[7] has similar mechanism to DAISY. Crusoe is based on VLIW and it has unique ISA. The processor runs CMS (Code Morphing Software) and the software dynamically translates x86 instructions to its internal ones. Different from DAISY, CMS translates only hot-spot codes and takes incremental optimization concurrently with program execution.

These systems listed above are for translation purpose into different ISA. Following systems aims at optimization.

Dynamo[8] of Hewlett-Packard translates PA-RISC binaries to PA-RISC codes for optimization purpose. Dynamo translates concurrently with emulation of PA-RISC instructions. From profiling results of emulation, it can find hot-spots and translates into optimized codes. The resulting codes are cached, thus, once the hot-spot is translated, the optimized codes are executed for acceleration.

Morph[9] of Harvard University does profiling under the cooperation with operating system, and it optimizes executed codes off-line using the results of profiling.

Deco[10] of Harvard University does run-time optimization and binary translation. Deco can re-translate optimized codes according to change of the program's behavior.

BOA[11] of IBM focuses EPIC-style approach, that aims at high clock frequency by simplified hardware, abandoned out-of-order superscalar mechanisms like PowerPC. BOA optimizes instruction scheduling for such architecture by using binary translation technology. Where DAISY translates only once, BOA continuously monitors the behavior of execution paths and does run-time optimization.

Java's HotSpotVM[12] collects profile information during interpretive execution. When it detects a hot-spot, the hot codes are translated into native codes. The VM uses the translated binaries so that it accelerates performance.

UQBT[13] is a framework of retargetable binary translation. The system's unique point is that, theoretically, it enables any ISA codes translated into any other ISA. Currently it supports SPARC, x86, and Java bytecode.

All systems shown above assume single-thread code and none aims at performance enhancement by multi-threading.

7 Concluding Remarks

In this paper, we proposed a binary translation and optimization system that enables existing binary codes to run on the future multithreaded processors. We first discussed about the basic assumption on the target architecture and the essential requirements for single-thread binary codes to be translated to multithreaded codes.

The proposed system roughly consists of static translator and optimizer (STO) and dynamic translator and optimizer (DTO). STO initially translates an input binary code to the multithreaded one. DTO handles the dynamic behavior of the translated program and optimizes according to profiling results at run-time.

A pilot binary translator was built for the sake of preliminary evaluation. Programs used for evaluation are integral calculation in a *sin* trigonometric function using a trapezoidal equation and inner product calculation. The results show overheads in thread pipelining and, if each thread has sufficient calculation, the overhead can be negligible and the speed-up, linear to the number of thread units, is achieved.

At the present time, DTO is not completed. We will continue to develop the proposed system and show effectiveness in practical programs such as SPEC benchmarks.

Acknowledgement This research was supported in part by the Grant-in-Aid for Scientific Research (C) of Japan Society for Promotion of Science (JSPS) No.12680328.

References

[1] J. Y. Tsai, J. Huang, and et al., "The Superthreaded Processor Architecture," *IEEE Transactions on Computers, Special Issue on Multithreaded Architectures*, Vol. 48, No. 9, 1999.

[2] D. F. Bacon, S. L. Graham and O. J. Sharp, "Compiler Transformations for High-Performance Computing," *ACM Computing Surveys*, Vol. 26, No. 4, pp. 345–420, 1994.

[3] M. D. Smith, "Overcoming the Challenges to Feedback-Directed Optimization," *Proceedings of ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo'00)*, 2000.

[4] J. Huang. "The SImulator for Multi-threaded Computer Architecture(SIMCA), Release 1.2.," http://www-mount.cs.umm.edu/Research/Ag-assiz/simca.html.

[5] R. J. Hookway and M. A. Herdeg, "DIGITAL FX!32: Combining Emulation and Binary Translation," *Digital Technical Journal*, Vol.9, No 1, pp. 3–12, 1997.

[6] K. Ebcioglu, E. R. Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility," *Proceedings of 24th Annual International Symposium on Computer Architecture*, pp. 26–37, 1997.

[7] A. Kaliber, "The Technology Behind Crusoe Processors," 2000, URL: http://www.transmeta.com/crusoe /download/pdf/crusoetechwp.pdf.

[8] V. Bala, E. Duesterwald, S. Banerji, "Dynamo: A Transparent Dynamic Optimization System," *Proceedings of Programming Language Design and Implementation*, 2000.

[9] X. Zhang, Z. Wang, and et al., "System Support for Automatic Profiling and Optimization," *Proceedings of 16th Symposium on Operating Systems Principles*, 2000.

[10] E. Feigin, "A Case for Automatic Run-Time Code Optimization," Senior thesis, Harvard College, Division of Engineering and Applied Sciences, 1999.

[11] S. Sathaye, P. Ledak, and et al., "BOA: Targeting Multi-Gigahertz with Binary Translation," *Workshop on Binary Translation (Binary99)*, 1999.

[12] Sun Microsystems, "Java HotSpotTM Technology," URL: http://java.sun.com/products/hotspot/

[13] C. Cifuentes and M. Van Emmerik, "UQBT: Adaptable Binary Translation at Low Cost," *Computer*, Vol. 33, No. 3, pp. 60–66, 2000.

[14] K. Ootsu, T. Ono, T. Baba, "A Methodology for Multithreading with Binary Translation," *IPSJ SIG Notes*, Vol.2001, No.10, pp.41–46, January 2001 (in Japanese).

[15] T. Ono, K. Ootsu, T. Yokota, T. Baba, "Preliminary Evaluation of Binary-Level Multithreading," *IPSJ SIG Notes*, Vol.2001, No.76, pp.183–188, August 2001 (in Japanese).

The Predictability of Computations that Produce Unpredictable Outcomes

Tor Aamodt Andreas Moshovos Paul Chow

Department of Electrical and Computer Engineering
University of Toronto
{aamodt,moshovos,pc}@eecg.toronto.edu

Abstract

We study the dynamic stream of slices that lead to branches that foil an existing branch predictor and to loads that miss and measure whether these slices exhibit locality (i.e. repetition). We argue that this regularity can be used to dynamically extract slices for an operation-based predictor that speculatively pre-computes a load address or branch target (i.e. an outcome) rather than directly predicting the outcome based upon the history of outcomes. We study programs from the SPEC2000 suite and find they exhibit good slice-locality for these problem loads and branches. Moreover, we study the performance of an idealized operation-based predictor (it can execute slices instantaneously). We find that it interacts favorably with an existing sophisticated outcome-based branch predictor, and that slice-locality provides good insight into the fraction of all branch mispredictions it can potentially eliminate. Similar observations hold for operation-based prefetching of loads that miss. On average slice locality for branches and loads was found to be above 64% and 76% respectively when recording the 4 most recent unique slices per branch or load over a window of 64 committed instructions, and close to 61% and 73% for branches and loads respectively when we look at slices over a window of up to 128 committed instructions. The idealized operation predictor was found to correct approximately 68% of branch mispredictions or prefetch about 67% of loads that miss respectively (slices detected over a window of 64 instructions). At the same time, on average, the branch operation predictor mispredicts less than 0.6% of all branches that are correctly predicted by an existing branch predictor.

1 Introduction

Recently, the prospect of generalized *operation-prediction* has been raised as a way of boosting accuracy over existing *outcome-based* predictors. In operation prediction we guess a sequence of operations, or a computation *slice* that can be used to pre-compute a performance critical outcome (e.g., load address or branch target). This is in contrast to outcome-based predictors that directly predict outcomes exploiting regularities in the outcome stream. Since operation prediction does not require any regularity in the outcome stream, it has the potential of predicting outcomes that foil existing outcome-based predictors (in section 2, we provide an example that illustrates the potential of operation prediction).

Several recent proposals have shown that *slice-based precomputation* (the mechanism operation-prediction uses for predicting outcomes) can be used to successfully prefetch memory data, and may potentially be used to pre-compute hard to predict branches [4,9,15,16,10,11,12,17]. In this work, we study program behavior to understand *why operation-prediction works or may work* for predicting otherwise hard to predict program events.

We build on the experience with outcome-history-based dynamic prediction and study whether typical programs exhibit the behavior necessary for *operation history-based prediction* to be successful. We explain that, in a way that parallels outcome-based prediction, operation predictors can be built to exploit regularities in the *operation* (i.e., computation) stream. For example, previous work has shown that sufficient *locality*, or repetition exists in the value stream of many programs. This program characteristic is what facilitates outcome-based value prediction. In this work we study a set of programs from the SPEC2000 suite to determine whether sufficient repetition exists in the *slices* used to calculate performance critical outcomes that otherwise foil existing outcome-based predictors. This program characteristic is necessary (but not sufficient as we explain in section 2) if history-based operation prediction is to be successful. We restrict our attention to mispredicted branches and to loads that miss and study how much repetition, or *locality* exists in the operation streams that lead to them. To the best of our knowledge, no previous work on the dynamic locality characteristics of such slices exist. With few exceptions and as we explain in section 4, related proposals approach slice pre-execution as an alternate execution model, where the compiler orchestrates slice generation and pre-execution. While compiler directed slice pre-execution is an interesting and viable option, dynamic slice detection and execution can have its own advantages (e.g., binary compatibility). Accordingly, we believe it is an important alternative that deserves attention.

Our study provides the foundation necessary for understanding whether programs exhibit some of the behavior necessary for operation prediction. Moreover, our results provide insight on what kind of operation predictors we should be considering if we are to achieve a desired accuracy and coverage. For example, our study shows how successful a *last-operation* predictor can potentially be or whether *pattern-based operation* predictors may be necessary. A last-operation prediction would simply record the slice used to calculate a branch or load and use it the next time around to pre-calculate the branch or the load address. Such a predictor can be successful only if slices tend to repeat multiple times. Alternatively, a pattern-based operation predictor can exploit patterns in slice occurrence, e.g., slice S1 appears always after slice S2, and so on. While more complex, a pattern-based operation predictor could offer better accuracy and coverage over a last-operation one. How-

ever, in this work we restrict our attention to analyzing the potential of operation prediction. Specifically, the predictors we studied pre-compute their slices instantaneously. An actual predictor would require some time to execute through the predicted slice, hence it may not be able to pre-execute the slice early enough for prediction purposes. Further work is necessary to determine whether this is possible. Yet, in previous work we have shown that a simple predictor for loads that miss can successfully pre-execute loads that miss often for a set of pointer-intensive applications [9].

Our results indicate that performance critical slices exhibit high locality, more so for loads that miss. In particular, we find that average slice locality for branches and loads is above 64% and 76% when we record up the 4 most recent slices per branch or load respectively over a window of 64 committed instructions and close to 61% and 73% for branches and loads respectively when we look at slices over a window of up to 128 committed instructions. Our idealized operation predictor can correctly predict about 68% of mispredicted branches and accurately prefetch 67% of loads that miss (slices detected over a window of 64 instructions). At the same time, on the average the branch operation predictor mispredicts less than 1% of all branches that are correctly predicted by an existing branch predictor. Overall, we find that coverage (e.g., the fraction of branches that get a correct prediction from the operation predictor but an incorrect prediction from the existing outcome-based predictor) is highly correlated to the locality exhibited by the corresponding slices.

The rest of this paper is organized as follows. Section 2 reviews operation prediction, how it relates to outcome-based prediction, and the various choices existing when dynamically extracting slices. Section 3 presents our locality and accuracy results. In Section 4, we discuss related work explaining how operation prediction relates to other recently proposed slice-based execution models. Finally, Section 5 summarizes our findings and offers concluding remarks.

2 Operation Prediction Basics

In this section we review operation prediction, explain how it relates to existing outcome-based predictors, and discuss what requirements exist for operation prediction to be successful. In section 2.1, we discuss some of the choices that exist in dynamically extracting slices and explain the choices made for the purposes of our study.

Consider the example code fragment of figure 1(a). It is an infinite while loop containing a switch statement. What particular target the switch statement will follow depends on the value read from the uni-dimensional buffer. First, consider how an outcome-based predictor will attempt to predict the branch that implements the switch statement. Such a predictor will observe the outcome stream of this branch (and possibly of other branches also). That is, it will observe the various targets taken by the switch statement during successive iterations of the while loop. It will try to associate each target occurrence with an appropriate *target history*, that is a sequence of past targets that preceded the one in question. The hope is that next time the same target history appears, the same target will follow. For example, such a predictor may observe that when the targets for "A" and "B" appear, then with high probability the target for "C" appears. This predictor may then guess "C" every time "A" and "B" appear in sequence. Essentially, the outcome-based predictor builds a tabular, approximate representation of the program's function by observing the values (outcomes) it generates. Outcome-based prediction is successful if the outcome-stream exhibits sufficient repetition, a property commonly referred to as *locality*. In our example code, repetition will exist only to the extent that the data stored in the buffer array follows some repeatable pattern. Operation prediction offers the potential of predicting outcomes that do not necessarily follow a repeatable pattern. Rather than trying to guess the program's function based on the values it produces, it directly observes the computation stream, attempting to exploit any regularities found there. Returning to our switch statement example, let us now take a closer look at what happens during execution time. Figure 1(b) shows how the switch statement is implemented in pseudo-MIPS machine code. When the code of part (a) executes, the computation stream will contain repeated appearances of the computation *slice* shown in part (b). While the target computed by each slice may be different, we can observe that the actual slice remains constant. Operation prediction builds on this observation and attempts to *dynamically* identify such slices and use them to pre-compute outcomes that otherwise foil outcome-based predictors. As we explain in section 4, operation prediction has existed in restricted form for years. For example, stride-based prefetchers or value predictors are examples of specialized operation prediction where the actual slice or class of slices is built in the predictor design.

In this work we are concerned with generalized operation prediction where the slices are dynamically extracted and predicted. Following a generalization of the model proposed by Moshovos *et al.*, [9], an operation predictor for our example would identify the "jr" (instruction 7) as a problematic control flow instruction, or as a *target* instruction. At commit time, it will extract the computation slice that lead to the particular instance of the target instruction as shown in part (b). This slice, will contain only the instructions that contributed to the calculation of the actual target. Note that these instructions are not necessarily adjacent in the dynamic instruction trace (a mechanism for extracting such slices has been proposed [9]). This slice will be stored in a *slice cache* where it will be identified by the *lead* instruction (i.e., the oldest one, instruction 1 in our example). Next time the lead instruction appears in the decode stage, the slice will be executed as a separate *scout thread*. Provided that the scout thread completes before the appropriate instance of the target instruction appears, the processor may use its result to predict the target. The aforementioned steps for operation prediction parallel those for outcome-based prediction. In operation prediction the unit of prediction is a slice while in outcome-based prediction it is an outcome. Accordingly, detecting a slice and storing it in the slice cache is equivalent to observing an outcome and recording it in a prediction table. Executing a scout thread is equivalent to probing the prediction table.

The operation predictor described uses history-based prediction concepts. Such a predictor observes the slices of otherwise unpredictable results. If these slices tend to follow a repeatable pattern then it may be possible to use the past history of appearances to accurately predict the slices of future instances and hence pre-compute otherwise unpredictable outcomes. The same principle underlies many existing outcome-based predictors instead of exploiting regularity in the slice stream we instead exploit regularity in the outcome stream (e.g., values, addresses and branch directions). For history-based operation prediction to be successful it is necessary to

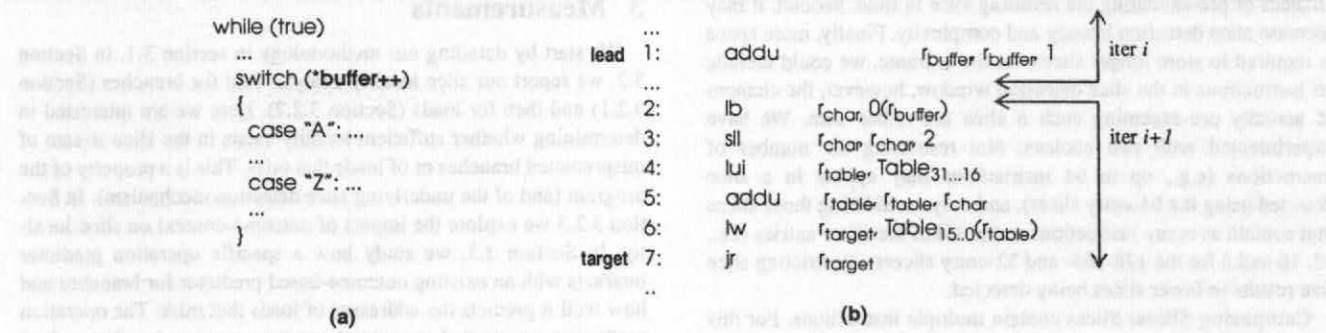


***Figure 1:** A switch statement whose target behavior depends on the data stored within the buffer array. (b) The computation slice that calculates the target during run-time.*

have sufficient regularity in the computation, or slice stream of the instruction we want to predict. Moreover, the slices so identified must be able to execute and complete before the main thread needs the prediction itself. In this work we focus mainly on the first requirement. In particular, we study the slice locality characteristics of some SPEC2000 programs focusing on branches that are mispredicted by an outcome-based branch predictor and on loads that miss.

Before we present our results it is necessary to re-iterate why scout threads may be able to run-ahead of the main thread and to comment on how operation-prediction relates to outcome-based prediction. Scout threads may be able to pre-calculate a result because: (1) The main thread includes all other intervening instructions which need to be fetched, decoded and executed. (2) The main thread also may be stalled due to intervening control-flow miss-predictions. Since scout threads do not include any control flow, they may proceed undisturbed. Finally, while operation prediction may be able to predict outcomes that do not exhibit regularity, it does need to calculate these outcomes. Outcome-based prediction forgoes this calculation replacing it with a straightforward table lookup. Hence, whenever outcome regularity exists outcome-based prediction may be preferable over operation prediction.

2.1 Slices and Slice Locality

Before defining and measuring slice locality we must be clear about how we define a *slice*. Conceptually, a slice may include instructions that appear long in advance (e.g., thousands of instructions) of the target instruction. Moreover, a slice could be defined to contain arbitrary control-flow and memory dependences (to adhere to the static definition of a computation slice). With this definition, the slice for each instance of the "jr" instruction in figure 1 would include all preceding instances of instruction 1 (updates of the buffer pointer), plus all instructions that wrote the corresponding data element of the buffer array (this may include instructions past a system call). Such a definition is impractical for our purposes. Accordingly, our slice definition stems from a practical implementation of a slice detector [9] and of the sketch of how an operation predictor could work discussed earlier. In the rest of this section we explain the choices we made in defining and extracting slices, and then we present our definition of slice locality.

Slice Detection Window: In searching for instructions to construct a slice, we consider only those instructions that appear within a fixed distance from the target instruction. In particular, we extract slices using a fixed length *slice detection window* or *slicer*. The instructions in the slicer form a continuous chunk of the dynamic instruction trace. Only committed instructions enter the slicer. When a target instruction is committed, its slice is extracted using a backwards data-flow walk which eliminates all operations that do not directly contribute to the target outcome. Slicer size affects slice length and therefore it impacts slice locality and the ability to pre-execute slices early enough. While a shorter slicer may result in fewer shorter slices per target instruction and hence in higher repetition in the dynamic slice stream, the distance between the target and lead instructions in these slices could be small. Consequently, it may be harder for those slices to run-ahead of the main thread. For this reason we experimented with various slicers of 32, 64 or 128 instructions. We could study locality with larger slicers. We have performed some experiments and found that locality drops rapidly beyond 128 for most programs.

Control-Flow: Besides how far back we look in the dynamic instruction trace, a second choice in detecting slices is whether we include intervening control-flow instructions. In this study we do not. Slice detection occurs over a chunk of the dynamic instruction trace. Since this is a trace, it only includes a specific control-flow path and does not contain the parts of the static slice that would appear on other control-flow paths. Accordingly, from a practical standpoint it is convenient to ignore any intervening control flow instructions. Later on we explain, that the *implied control flow path* (i.e., the directions of all intervening branches at detection time) can be used to select the appropriate slice for prediction.

Memory Dependences: Another choice regarding slices is whether we follow memory dependences including stores and their parents. Conceptually, the following tradeoffs exist: Including memory dependences may allow us to look further in the past, capturing a lead instruction that appears further away from the target. Moreover, including memory dependences may improve slice accuracy since, if a memory dependence exists, we will be waiting appropriately for the corresponding data. However, since memory dependences may be changing over time, including them could result in incorrect slices. We report results for slices that follow memory dependencies.

Slice Size: Slices with only one instruction (the target), are always discarded in this study, as the practical implementation discussed earlier cannot use them to any benefit. We could also choose to restrict our attention to those slices that contain at most a fraction of all instructions in the slicer. While including more instructions may allow us to capture an earlier lead instruction, at the same time it has several, potentially negative implications: First, it reduces the

chances of pre-executing the resulting slice in time. Second, it may increase slice detection latency and complexity. Finally, more space is required to store longer slices. At the extreme, we could include all instructions in the slice detection window, however, the chances of actually pre-executing such a slice are rather slim. We have experimented with two choices: Not restricting the number of instructions (e.g., up to 64 instructions may appear in a slice detected using the 64-entry slicer), and only considering those slices that contain as many instructions as the 1/4 of the slicer entries (i.e., 32, 16 and 8 for the 128-, 64- and 32-entry slicers). Restricting slice size results in fewer slices being detected.

Comparing Slices: Slices contain multiple instructions. For this reason and in contrast to outcomes, there are several ways in which two slices can be compared for the purposes of measuring locality. In this study, we consider two slices identical if they are *lexically identical*. That is, if they contain the same instruction sequence. With this definition two slices may be considered equivalent even if the PCs of individual instructions may differ. This definition is both practical and it accommodates identical slices that may appear on different control-flow paths. For the purposes of locality measurements we ignore the implied control flow in slices. So two slices that are lexically identical but appear on different control flow paths and have different implied control flow will be considered the same.

Slice Locality: For unrestricted slices (i.e. for any length, even slices containing just the target operation), we can now define *slice-locality(n)* of a target instruction as the relative frequency with which a detected slice was encountered within the last *n unique* slices detected by preceding executions of the same static instruction. *Slice-locality(1)* is the relative frequency that the same slice is encountered in two consecutive executions of a target instruction. A high value of *slice-locality(1)* suggests that a simple, "last slice encountered"-based predictor could be accurate. For values of *n* greater than 1, *slice-locality(n)* is a metric of the working set of slices per instruction. Formally, it is the relative frequency with which the same slice was detected within the last *n* unique slices detected for the specific instruction, assuming there is always a slice. When excluding slices due to the restrictions considered earlier, *slice-locality(n)* is the relative frequency that a given branch or loads's slice both meets the restriction criteria, and was seen in the last *n* unique slices that also matched the criteria. While a small working set does not imply regularity, we will later explain that it may be possible to execute all these slices in parallel and then select the appropriate one based on the implied control flow.

Outcome Context: In practice, having identified a problem instruction, one might detect a slice and record it independent of the whether the underlying outcome based predictor was correct, or choose to record a slice only when a misprediction or cache miss actually occurs. The difference is that an outcome may only be hard for the outcome-based predictor to anticipate when following the implied control-flow of a small subset of all slices seen. We have measured the impact on locality as viewed from mispredicted branches and cache misses under both circumstances and conclude that statistically there is a benefit to waiting for a mispredicted target branch, or load that misses, when detecting slices for a particular static branch or load. Except where stated otherwise (i.e., in Section 3.2.3) all measurements reported in this paper are based upon the latter approach.

3 Measurements

We start by detailing our methodology in section 3.1. In Section 3.2, we report our slice locality analysis first for branches (Section 3.2.1) and then for loads (Section 3.2.2). Here we are interested in determining whether sufficient locality exists in the slice stream of mispredicted branches or of loads that miss. This is a property of the program (and of the underlying slice detection mechanism). In Section 3.2.3 we explore the impact of outcome-context on slice-locality. In Section 3.3, we study how a specific operation predictor interacts with an existing outcome-based predictor for branches and how well it predicts the addresses of loads that miss. The operation predictors we studied execute slices *instantaneously* when a lead instruction is encountered. Our goal is to understand the potential of slice prediction. Further work is necessary to develop realistic predictors where slice execution takes some time. Our results provide the insight necessary to do so in a well educated manner.

3.1 Methodology

We have used the programs from the SPEC2000 suite shown in table 2. All programs were compiled with gcc (-O2 -funroll-loops -finline-functions) for the MIPS-like Simplescalar instruction set (PISA). We have used the test input data sets. To obtain reasonable simulation times, we skipped the initialization phase and warmed up the caches and the branch predictor for the next 25 million instructions. The actual number of instructions skipped (i.e., functionally simulated) is shown in table 2. Our measurements were made over the next 300 million instructions. In table 2, we also report the L1 data cache miss rates and the branch prediction accuracies (direction and target address). In the interest of space, we use the labels shown in table 2 in our graphs. To obtain our measurements we have modified the Simplescalar 3 simulator. Our base configuration is an 8-way dynamically-scheduled superscalar processor with the characteristics shown in table 1. Our base processor has a 12 cycle minimum pipeline.

3.2 Slice Locality

In this section we study the locality of slices first for branches and then for loads. For branches, we focus on those dynamic instances that are mispredicted by the underlying outcome-based predictor and study whether locality exists in their slice stream. This is necessary if history-based operation-prediction is going to be successful. For loads, we focus on those dynamic instances that miss in the data cache. In both cases we examine only the slices that lead to mispredictions, or cache misses, respectively except in Section 3.2.3 were the impact of ignoring outcome-context is examined.

Measuring locality in the way we do here allows us to avoid any artifacts that a specific implementation of operation prediction may introduce. Later in section 3.3, we study models of specific operation predictors.

3.2.1 Branch Slice Locality

Figure 2 reports the weighted average of slice-locality(n) for those branches that are mispredicted by the underlying outcome-based branch predictor. To calculate *slice-locality(n)*, the distributions for each static branch are weighted by the relative number of outcome-based mispredictions associated with that branch, and so the overall figure naturally emphasizes those static branches which are mispredicted most often. We report locality in the range of 1 (bottom bar) through 4 (top bar) and for a variety of slicer configu-

Base Processor Configuration			
Branch Predictor	64K GShare+64K bimodal with 64K selector	**Fetch Unit**	Up to 8 instr. per cycle. 64-entry Fetch Buffer Non-blocking Fetch
Instruction Window Size	128 entries	**FU Latencies**	same as MIPS R10000
Issue/Decode/Commit BW	8 instructions / cycle	**Main Memory**	Infinite, 100 cycles
L1 - Instruction cache	64K, 2-way SA, 32-byte blocks, 3 cycle hit latency	**L1 - Data cache**	64K, 4-way SA, 32-byte blocks, 3 cycle hit latency
Unified L2	256K, 4-way SA, 64-byte blocks, 16 cycles hit latency	**Load/Store Queue**	64 entries, 4 loads or stores per cycle Perfect disambiguation

Table 1: *Base configuration details. We model an aggressive 8-way, dynamically-scheduled superscalar processor having a 128-entry scheduler and a 64-entry load/store queue.*

Benchmark	*Label*	*Inst. Skipped*	*MR*	*BPA*	*Benchmark*	*Label*	*Inst. Skipped*	*MR*	*BPA*
164.gzip	*gzp*	101 M	3.1%	92.3%	**183.equake**	*eqk*	359 M	2.7%	90.4%
175.vpr	*vpr*	33 M	2.6%	91.0%	**188.ammp**	*amp*	100 M	28.3%	99.1%
176.gcc	*gcc*	200 M	0.9%	91.4%	**197.parser**	*prs*	144 M	2.3%	91.2%
177.mesa	*msa*	101 M	0.7%	99.9%	**255.vortex**	*vor*	102 M	0.7%	98.5%
179.art	*art*	1,686 M	43.9%	98.4%	**256.bzip2**	*bzp*	100 M	3.9%	97.6%
181.mcf	*mcf*	50 M	5.3%	90.9%	**300.twolf**	*twf*	188 M	6.2%	85.1%

Table 2: *Programs used in our experimental evaluation. MR is the L1 data miss rate. BPA is the branch prediction accuracy (direction+target). We simulated 300 million committed instructions after skipping the initialization phase.*

rations. To identify the slicers we use an *"NSM"* naming scheme. *"N"* is the size of the slicer, i.e., 256, 128, 64 or 32. *"S"* can be either *"U"* (unrestricted) or *"R"* (restricted) and specifies whether we restrict slice size to up 1/4 of total number of instructions in the slicer or not. Finally, "M" signifies that slices include memory dependencies. For example, *64UM* corresponds to a slicer with 64 entries that can produce slices of up to 64 instructions and that is capable of following memory dependences. *32RM* is a slicer that has 32 entries and that can detect slices that include only up to 8 instructions and that can follow memory dependences. We have experimented with various slicer configurations. In the interest of space we report the following seven from left to right: *256RM, 128RM, 64RM, 32RM, 64UM,* and *32UM.*

Before we present our results it is important to emphasize that while high locality is desirable, *any* locality may be useful for improving branch prediction accuracy. This is because we measure locality only for mispredicted branches. As we will show in section 3.3, even when little locality exists, it can positively impact overall branch prediction accuracy.

With unrestricted slices, in all cases but *gzip* and *mesa*, using a shorter slicer results in higher locality with the average locality going from 73% to 83% comparing 64UM to 32UM. With restricted slices and a short detection window (32RM) there is much lower locality compared to unrestricted slices (32UM), and furthermore, the locality increases going from a 32-entry slicer to a 64-entry slicer, on average from 59% to 66%. This result suggests that many slices have more than 8 instructions that are close to the target instruction. This result corroborates the observation by Zilles and Sohi that many operations that directly contribute to the computation of the target are clustered close to the target operation [15]. As we use a fixed ratio of 1/4 to restrict slices, a shorter slicer is penalized more heavily than a longer one. Indeed, for a 256-entry slicer (256RM) we see the dominant trend is again a decrease in locality for longer slices.

On the average, slice-locality(4) is about 49% with the 256RM slicer and rises to about 61%, and 65% for the 128RM and 64RM slicer, while falling back to 59% for the 32RM slicer. More importantly, most of the locality is captured even if we can record a single slice per instruction. In particular, slice-locality(1) is approximately 34%, 41%, 45% and 46% for the 256RM, 128RM, 64RM and 32RM slicers. This suggests that a last-slice-based predictor may be quite successful.

As we move towards larger slicers, locality usually drops. In the worst case of *vpr*, slice-locality(1) drops to about 10% with the 256RM slicer. For several programs the drop of locality with increased slicing windows is a lot less dramatic and slice-locality(1) remains well above 30% for 256RM for half the benchmarks. However, a larger slicer does not necessary result in lower locality. In particular, for *gzip* and *mesa* locality(1) *increases* as the slicer is increased from 32 to 64 entries, *even for unrestricted slices.* This anomaly has been studied carefully and appears to be the result of intervening control flow: A larger slicer allows us to look through more instructions when detecting a slice, and hence capture longer slices. Normally, this tends to strongly reduce slice locality because the number of implied control flow paths leading up to the target multiplies as additional basic blocks appear in the slicer. However, a longer slicer may also increase locality when a slice skips over a segment of instructions that fluctuates in length due to intervening control flow. With a short slicer, the earlier part of the slice may be evicted occasionally. With a longer slicer, the whole slice may still appear in the slicer.

In table 3 we report the average instruction distance between the lead and the target instructions and the average instruction count per slice. We define *instruction distance* as the number of intervening instructions (including the lead) in the original instruction trace. In the interest of space we restrict our attention to the 256RM, 128RM, 68RM and 32RM slicers. These two metrics provide an indication of whether the slices could potentially run-ahead of the main thread (of course, this can only be measured using an actual implementa-

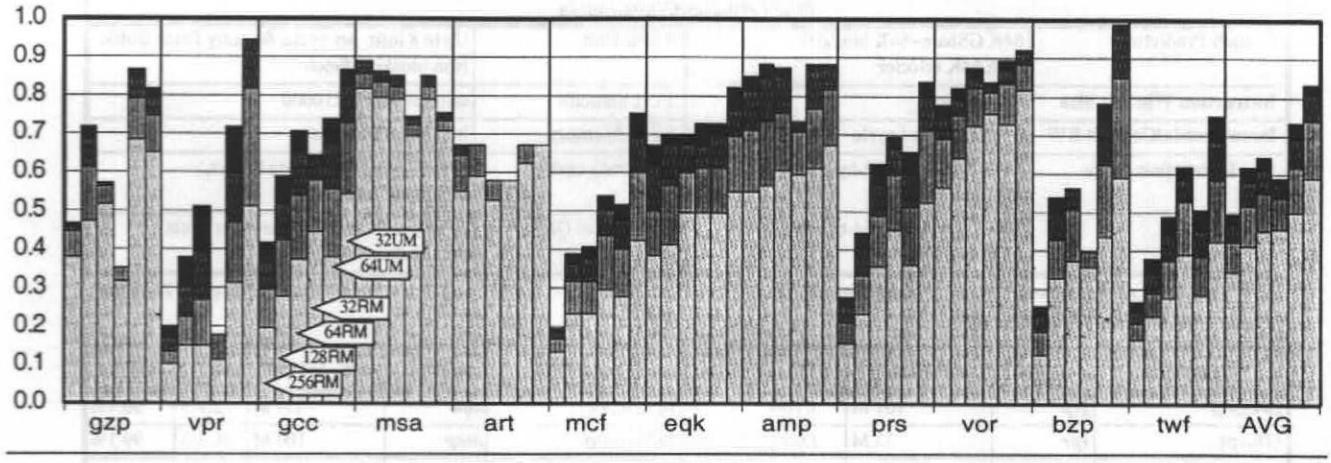


Figure 2: Weighted average slice-locality distribution for mispredicted branches (see text for description of weighting procedure). Range shown is 1 (bottom) to 4 (top). We use a NSM scheme where N is the size of the detection window (256, 128, 64, or 32), S is "U" if no restrictions on slice size are placed and "R" if we restrict slices to 1/4 of the slice detection window, and finally, "M" indicates that we follow memory dependences in constructing slices.

Program	256-R-M			128-R-M			64-R-M			32-R-M		
	Dist.	Cnt.	#St	Dist.	Cnt.	#St	Dist.	Cnt.	#St	Dist	Cnt.	#St
gzp	110.1	12.7	0.84	90.1	14.2	1.09	42.1	7.2	0.35	16.4	4.3	0.15
vpr	112.5	19.1	0.54	95.9	15.8	0.27	50.8	10.4	0.05	22.3	5.1	0.17
gcc	111.8	15.2	0.73	93.3	11.7	0.45	41.9	7.6	0.18	17.8	4.7	0.05
msa	104.3	8.0	0.00	77.2	5.7	0.00	43.0	4.6	0.00	16.2	3.6	0.00
art	126.4	13.3	0.00	109.6	9.1	0.00	46.0	5.4	0.00	15.3	4.1	0.00
mcf	104.0	18.5	0.24	100.0	16.8	0.12	43.3	8.5	0.04	17.2	5.2	0.00
eqk	90.6	6.8	0.02	64.0	6.0	0.00	35.3	4.5	0.00	14.7	3.2	0.00
amp	106.6	12.1	0.49	80.3	9.4	0.32	40.4	7.3	0.06	18.4	5.0	0.05
prs	111.7	16.4	0.61	101.2	12.2	0.48	47.7	7.4	0.18	19.8	4.6	0.07
vor	109.7	13.7	0.29	99.7	9.4	0.32	47.9	6.4	0.13	22.0	4.1	0.04
bzp	111.8	18.0	0.11	103.5	19.4	0.15	48.3	9.8	0.00	23.3	5.7	0.00
twf	95.6	15.2	0.10	74.3	13.2	0.10	44.5	9.0	0.06	20.3	4.8	0.03

Table 3: Branch slice statistics: Weighted average instruction distance ("Dist."), instruction count ("Cnt."), and number of stores ("#St") for various slice detection setups. Each slice weighted by the number of mispredictions potentially corrected.

tion of an operation predictor). Overall, slice instruction count is relatively small and remains small even when we move to longer slicers. Moreover, the lead to target instruction distances are on the average considerable, especially with the 256RM slicer.

Table 3 also reports the average number of stores per slice. The number of stores is a metric of the number of memory dependences in each slice. The number of memory dependencies detected tends to grow with slicer size (similar to observations by Zilles and Sohi [15]), however, for the slicer sizes studied here, the number of dependencies detected was small.

These results are encouraging as they suggest that relatively high locality exists in the computation slices that lead to unpredictable branches. Moreover, slices tend to be small in size (on the average), spread over several tens of instructions of the original program. Having shown that programs exhibit the locality necessary for operation prediction of otherwise mispredicted branches, in Section 3.3.1 we measure how an approximate operation predictor interacts with the underlying outcome-based branch predictor.

3.2.2 *Load Slice Locality*

Figure 3 reports weighted average slice-locality(n) for those loads that miss in the L1 data cache. The weighting of the distribution for each static load is based upon the frequency of misses for that load. We report results for the same slicer configurations we presented in section 3.2.1. We observe trends similar to those for mispredicted branches but with locality being stronger. On the average slice-locality(1) is 58%, 56%, 47%, and 38% for the 32RM, 64RM, 128RM, and 256RM slicers. Recording up to 4 slices per instruction results in a locality of 68%, 77%, 73% and 60% respectively. For most programs, load slice locality is stronger than branch slice locality was. In table 4 we also report the average lead to target instruction distance, instruction count, and number of included stores for load slices. Slice instruction distance increases with the slicer size and is relatively large. Moreover, slice instruction count remains relatively small even with the larger slicers. Load slices tend to contain more memory dependencies that branch slices.

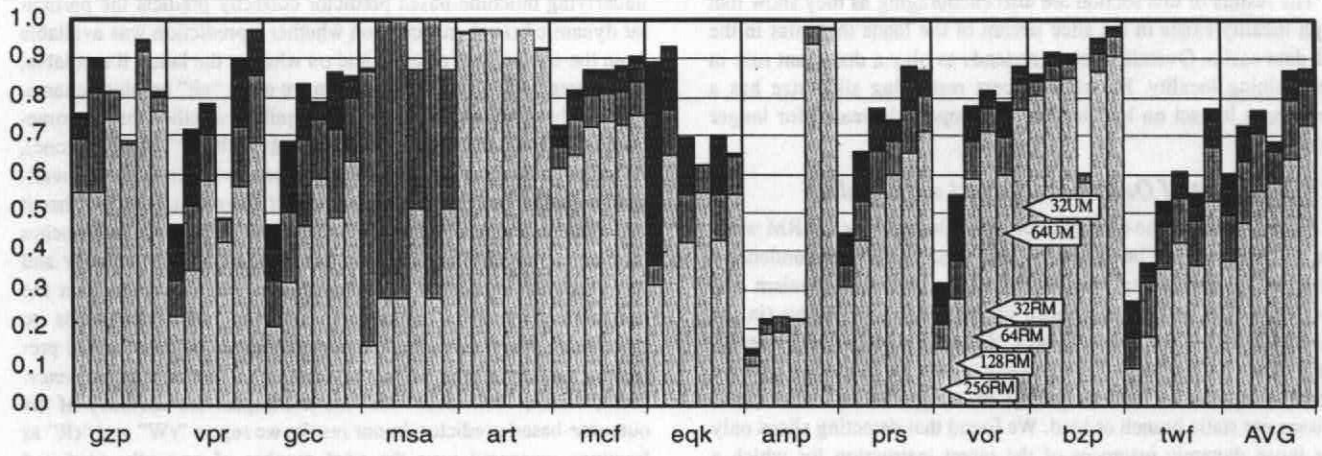


Figure 3: Slice locality distribution for loads that miss in the L1 data cache. Range shown is 1 (bottom) to 4 (top). We report results for the same slicer configurations as in figure 2.

Program	256-R-M			128-R-M			64-R-M			32-R-M		
	Dist.	Cnt.	#S	Dist.	Cnt.	#S	Dist	Cnt.	#S	Dist.	Cnt.	#S
gzp	126.3	13.3	3.07	87.4	10.4	1.75	29.7	5.1	0.93	18.7	3.8	0.84
vpr	112.8	18.3	1.77	97.4	13.9	1.22	45.1	9.4	0.76	15.8	3.9	0.50
gcc	118.5	14.5	1.62	99.6	10.1	1.28	46.3	6.1	0.95	22.7	3.8	0.78
msa	126.7	15.0	1.36	113.7	10.3	1.08	55.9	6.6	0.96	22.2	4.4	0.76
art	126.9	10.6	0.00	109.5	6.2	0.00	46.1	3.9	0.00	17.4	3.0	0.00
mcf	124.8	22.8	0.10	108.5	14.0	0.07	47.0	7.9	0.04	14.9	4.8	0.02
eqk	117.5	11.0	0.57	88.3	7.2	0.46	37.7	6.0	0.22	18.1	3.9	0.14
amp	124.9	18.2	3.10	113.2	16.2	1.36	50.9	8.6	0.80	23.9	5.0	0.66
prs	125.5	22.1	1.25	113.9	14.1	1.27	53.6	8.0	0.78	22.6	4.8	0.47
vor	124.9	16.9	2.05	113.3	11.1	1.61	53.6	6.1	0.99	24.2	3.5	0.63
bzp	117.3	14.0	0.08	100.0	10.8	0.13	42.8	7.9	0.05	16.6	4.5	0.07
twf	93.7	11.2	0.29	65.9	10.2	0.30	40.8	7.3	0.29	21.6	4.9	0.16

Table 4: Load slice statistics: Weighted average instruction distance ("Dist."), instruction count ("Cnt."), and number of stores ("#St") for various slice detection setups. Each slice weighted by the number of L1 cache misses potentially prefetched.

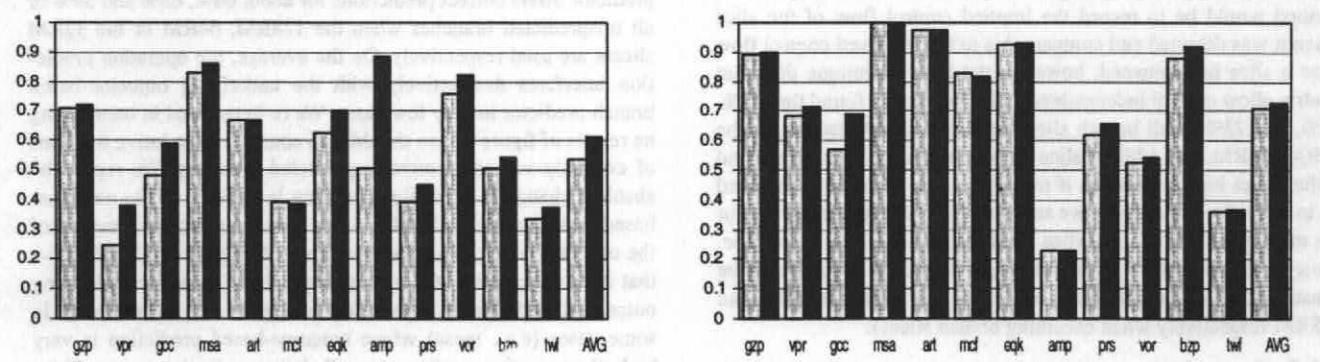


Figure 4: The effect of outcome-context on slice locality for branches (left) or loads (right): Detecting slices specifically when a branch was mispredicted or load missed improves locality. Bars represent locality(4) for a 128-R-M detection mechanism. Darker bars are for detection only on mispredict, or on cache miss, for branch slices and load slices respectively. Lighter bars represent the locality for slices of mispredicted branches and loads that miss when slices are extracted independent of whether there was a misprediction or cache miss.

The results of this section are also encouraging as they show that high locality exists in the slice stream of the loads that miss in the L1 data cache. Overall, slicer size tends to play a dominant role in determining locality. For short slicers restricting slice size has a very large impact on locality, but this impact decreases for longer slicers.

3.2.3 *Effect of Outcome Context on Locality*

Figure 4 reports the change in observed locality for 128RM when we allow slices to be added to the slice cache independent of whether the underlying outcome-based prediction mechanism was correct (in the case of branches), or there was a cache hit (in the case of loads). Note that both sets of measurements still represent only slices for branches that mispredict, or loads that miss, and are again weighted by the frequency of mispredictions and cache misses per static branch or load. We found that detecting slices only for those dynamic instances of the target instruction for which a misprediction, or cache miss event occurs improved locality by around 8% and 2% for branches and loads respectively. It is for this reason that all other results are reported based upon the detect-only-on-miss approach.

3.3 Accuracy and Interference with Outcome-Based Prediction

In this section, we model specific operation predictors and study their accuracy. We first explain how our branch operation predictor works. A slice is detected after each branch that was mispredicted. Detected slices are stored in an infinite slice cache where they are identified by the *lead* instruction. Only up to 4 slices per lead instruction can be present in the slice cache, however other than this there is no restriction on the total number of slices in the cache. Upon encountering a dynamic instance of the lead instruction we spawn *all* slices that are associated with it. Note that these slices may relate to the same, or different target operations. For the purposes of this study, we assume that the resulting scout threads complete *instantaneously*, however the outcomes of these threads are not used immediately. Also, all register and memory values from instructions before the lead are assumed to be available. The outcome from slice execution is saved while the slice is matched-up to the arriving flow of instructions. This matching is based upon matching instructions and register dependences. A more practical method would be to record the implied control flow of the slice when it was detected and compare this to the observed control flow after a slice has spawned, however, the latter technique does not readily allow control independence. On average we found that 47%, 58%, and 73% of all branch slice executions are discarded for the 32RM, 64RM, and 128RM slice detection mechanisms. When and if the target branch appears, if more than two slices have matched up to the instruction stream we select the first slice that spawned, or the most recently extracted slice if both spawned at the same time. Most of the time there is only a single prediction available to be consumed, if any (89%, 85%, and 80% for 32RM, 64RM, and 128RM respectively when executing branch slices).

3.3.1 *Branches*

To quantify the *potential* accuracy of our operation predictor for branches and how it interacts with the underlying outcome-based predictor we provide a breakdown of operation prediction for all dynamic branches. We break down branches based on whether the underlying outcome-based predictor correctly predicts the particular dynamic branch instance, on whether a prediction was available from the operation prediction and on whether the latter, if available, was correct. For ease of explanation we use a "vP" naming scheme. "v" can be *w*(rong) or *r*(ight) and signifies whether the outcome-based predictor correctly predicted the branch. "P" can be *N*(one), *W*(rong) or *R*(right) and signifies whether no prediction was available from the operation predictor, and if there was one, whether it was correct or not. For example, *rN* and *rR* correspond to branches that were correctly predicted by the outcome-based predictor and for which no prediction or a correct one was available from the operation predictor respectively. Category *rW* corresponds to *destructive interference* between operation and outcome-based prediction, while category *wR* corresponds to *constructive interference*. *"rN", "wN", "rR"* and *"wW"* do not impact the accuracy of the outcome-based predictor. In our results we report "rW" and "rR" as fractions measured over the total number of correctly predicted branches by the outcome predictor. We also report "wW" and "wR" as fractions measured over the total number of incorrectly predicted branches by the outcome predictor. Ideally, "rW", "wN" and "wW" would all be 0%, in which case "wR" would be 100% (all previously mispredicted branches are now correctly predicted by the operation-based predictor)

Figure 5 reports accuracy results for operation predictors that utilize, from left to right, a 128RM, 64RM or a 32RM slicer. Part (a) reports accuracy for correctly predicted branches (categories "rR" and "rW") while part (b) reports accuracy for mispredicted branches (categories "wR" and "wW"). Categories "rN" and "wN" are implied (missing upper part of the bars). In comparing the results of two graphs we must also take into account the relative fraction of correctly and incorrectly predicted branches (i.e., the accuracy of the underlying outcome-based predictor). We do so later in this section. In most cases, the operation predictors interact favorably with the underlying outcome predictor since "rW" is in most cases very small. In all programs, the operation predictor correctly predicts a large fraction of those branches that are mispredicted by the underlying outcome-based predictor as shown in part. (b) (category "wR") while it incorrectly predicts very few (category "wW").

On the average, ignoring timing considerations, the operation predictor offers correct predictions for about 66%, 68% and 59% of all mispredicted branches when the 128RM, 64RM or the 32RM slicers are used respectively. On the average, the operation prediction interferes destructively with the underlying outcome-based branch predictor in very few cases. We re-iterate that in interpreting he results of figure 5, one should also consider the relative fractions of correctly versus incorrectly predicted branches. We report the absolute change in prediction accuracy in addition to the outcome-based branch predictor in table 5 (the branch prediction accuracy of the outcome based predictor was reported in table 2). We observe that in most programs the operation predictor helps the underlying outcome-based predictor resulting in higher overall accuracy. In some cases (e.g., mesa) where outcome-based prediction is very high, the operation predictor actually harms overall accuracy. Since in most cases, this destructive interference occurs for programs with high branch accuracy, it may be possible to use a confidence mechanism (e.g., a counter with every slice) to filter out those slices that lead to incorrect predictions very often or to simply disable operation prediction when outcome prediction is above a threshold. Such

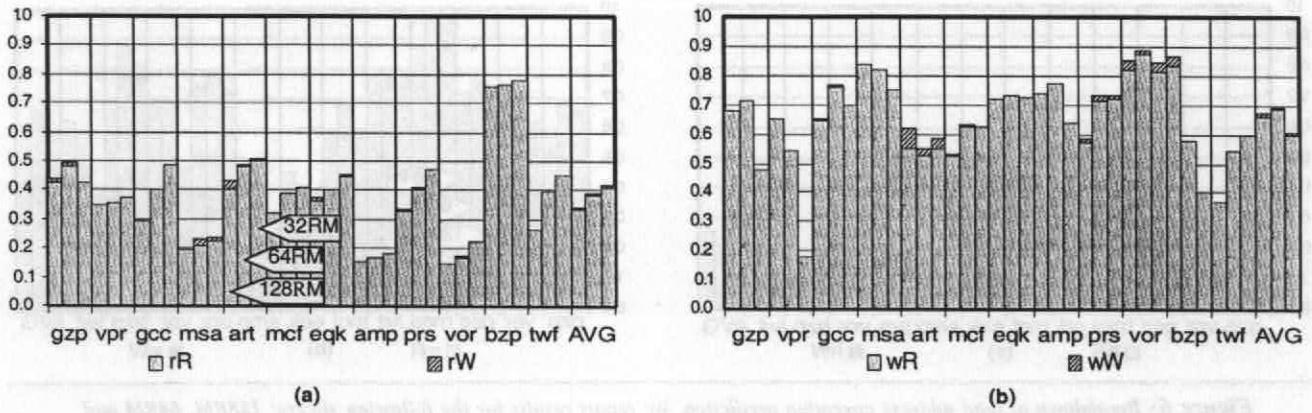


Figure 5: Interaction of operation branch prediction and outcome-based branch prediction. We report results for the following slice detection mechanisms: 128-R-M, 64-R-M and 32-R-M. (a) Correctly predicted branches and (b) Mispredicted branches.

Program	*128-R-M*	*64-R-M*	*32-R-M*	*Program*	*128-R-M*	*64-R-M*	*32-R-M*
gzp	+3.3%	+4.5%	+3.6%	*eqk*	+5.5%	+6.5%	+6.5%
vpr	+5.9%	+4.9%	+1.6%	*amp*	+0.6%	+0.6%	+0.5%
gcc	+5.4%	+6.4%	+5.9%	*prs*	+4.8%	+5.8%	+5.8%
msa	+0.1%	-1.7%	-1.7%	*vor*	+1.1%	+1.2%	+1.1%
art	-1.7%	+0.1%	+0.2%	*bzp*	+1.9%	+1.4%	+1.0%
mcf	+4.7%	+5.7%	+5.7%	*twf*	+5.4%	+8.1%	+8.9%

Table 5: Change in branch prediction accuracy with operation prediction over the base outcome-based predictor.

an investigation is beyond the scope of this paper. Overall the fraction of mispredicted branches that get a correct prediction from the operation predictor is greater than the locality we observed in section 3.2.1. The main reason is that we restrict the number of slices to 4 per lead PC as opposed to 4 per target PC which allows for greater coverage if a lead operation is associated with a single target.

3.3.2 *Loads*

Finally, we report accuracy for an operation predictor for load addresses. The results are shown in figure 6 for predictors based on the 128RM, 64RM and 32RM slicers. In part (a) we report results for those loads that *hit* in the data cache, while in part (b) we report results for the loads that *miss* in the data cache. The results of part (a) are provided for completeness. These loads hit in the data cache, so correctly predicting their addresses is not as important. We show two categories: *hR* are the loads whose addresses are correctly predicted while *hW* are the loads whose addresses are incorrectly predicted. In an actual implementation *hW* may translate into cache pollution. Overall, *hW* is negligible. In part (b) we report a breakdown of predictions for loads that miss in the data cache. Two categories are shown; *mR* includes the loads for whom the addresses are correctly predicted while *mW* includes those that are not. Ideally, mW would be 0% and mR would be 100%. In all cases, mW is barely noticeable. Moreover, mR tends to be higher for shorter slicers. We can observe that the accuracy of the operation predictor is extremely high (mR vs. mW). Moreover, the operation predictor offers correct predictions for many of the loads that miss in the data cache. Overall coverage is less than the locality we observed in section 3.2.2. The main reason is that now we restrict the number of slices to just 4 per lead PC as opposed to 4 per target PC (this restriction was placed since we need to associate slices with the lead PC in this operation predictor). In many programs, the same lead PC appears in the slices for more than one target load. Accordingly, thrashing occurs and coverage suffers. For example, consider a linked list where each element is a structure with multiple fields. All loads that access each field may be missing at the same time. All these loads will be getting the base address of the element in question from the same load. Consequently, their slices will probably have the same lead instruction and hence they will cause thrashing in the lead PC's slice set. A potential solution to this problem could be to allow more slices per lead PC. Alternatively, we may opt for carefully selecting the loads for which we detect slices and apply operation prediction (e.g., first loads that misses per block as opposed to *all* loads that miss per block).

The tradeoffs in load address prediction are quite different than those for branch prediction. In load address prediction, an incorrect prediction does not necessarily impact performance negatively. It can only do so indirectly by increasing resource contention or by polluting the data cache. Also, while we may predict the exact address incorrectly, we may still predict the correct cache block address correctly. Moreover, while it is desirable to have a high *coverage* (that is to provide correct predictions for as many of the loads that miss as possible), higher coverage may not translate in higher performance for reasons that include the following: Two loads that miss may be accessing the same block, accordingly, we may actually prefetch the block even if we do not correctly predict both of them. Also, in some cases, performance may be limited by other

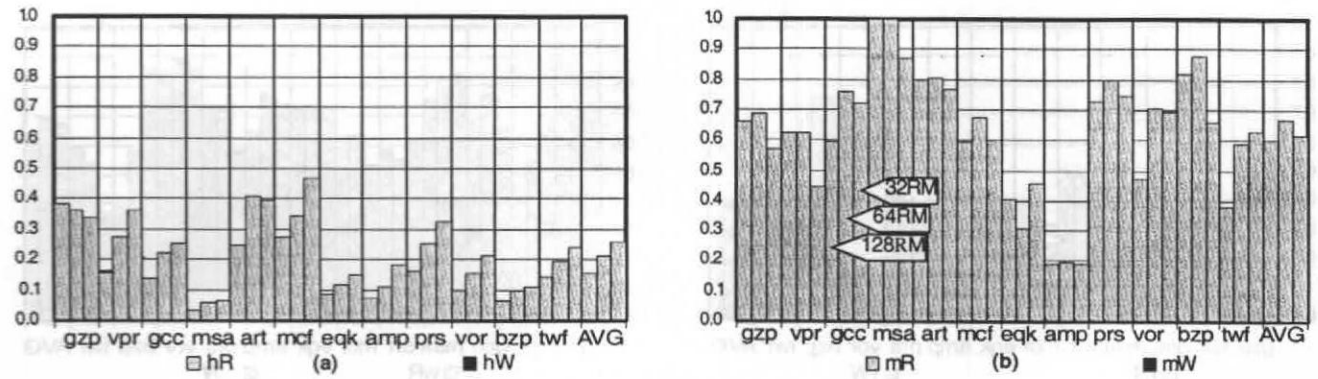


Figure 6: *Breakdown of load address operation prediction. We report results for the following slicers: 128RM, 64RM and 32RM. (a) Loads that hit in the data cache. (b) Loads that miss in the data cache.*

loads, hence correctly predicting a load address may have a negligible impact on performance.

4 Related Work

Operation prediction shares similarities with a number of recently proposed multi-threaded models where a number of potentially speculative, *helper threads* are used to enhance an otherwise sequential, main thread. *Simultaneous subordinate micro-threading* and *assisted execution* are two such proposals [2,13]. In the example application of SSMT given in [2] the helper threads are implemented in microcode and are used to enhance branch prediction.

Zilles and Sohi suggested extracting slices at compile time and using them to pre-execute performance critical instructions [15,16]. Assuming compile-time extraction, they have demonstrated that such slices can greatly improve performance, especially if they are optimized. Farcy *et al.*, proposed an operation predictor for branches for a restricted class of branches [5]. Moshovos also suggested the possibility of generalized operation prediction [8]. Moshovos *et al.*, proposed slice processors, the first dynamic, autonomous and generalized operation predictor-based prefetcher and demonstrated that it can improve performance even when compared to an outcome-based predictor [9]. Collins *et al.*, demonstrated a software-driven slice-based prefetcher for an EPIC-like architecture [4]. In parallel with this work, Collins *et al.*, also proposed a dynamic slice pre-execution prefetcher where slices are optimized and can be chained [17]. Annavaram *et al.*, proposed a non-speculative slice-based prefetching scheme where slices are detected and pre-executed from the fetch buffer and demonstrated that it can effectively prefetch data for a 4-way OOO core with a 64-entry scheduler [1]. Luk described a software-controlled prefetching method based on slice pre-execution [6]. In the *Speculative Data-Driven Multithreading (SDDM)* execution model, proposed by Roth and Sohi, performance critical slices leading to branches or frequently missing loads are pre-executed [12]. A *register integration* mechanism is used to merge slice produced results into the main thread and to filter out any incorrectly calculated values. As proposed, SDDM relies on a profiling phase and the compiler to build slices and to orchestrate their execution.

Some operation outcome predictors exist. Stride predictors are an early example where the actual computation is built in the design. Roth *at al.*, proposed an operation predictor for recursive data structures [10], while Mehrotra *et al.*, proposed operation predictors for linked lists and arrays [7]. Roth *at al.*, proposed an operation predictor for indirect jumps [11]. In all aforementioned proposals, the class of predictable operations is fixed in the design. *Slipstream Processors* also use a helper thread to run-ahead of the main sequential thread in effect pre-executing instructions [14]. The helper thread is formed by removing predictable computations from the main sequential thread. They study the dynamic creation of *chaining slices* in which a slice can, in essence, re-spawn itself. A similar chaining mechanism was proposed by Zilles and Sohi in [16] based on hand-optimized slices. Finally, an *Instruction Path Coprocessor* could potentially be used to support dynamic extraction and execution of slices [3].

This study also appears in our recent technical report [18]. To the best of our knowledge, no other work on the locality characteristics of the slice stream of mispredicted branches or loads that miss exists. Moreover, in their majority, most aforementioned slice-based execution models rely on compile-time slice creation or manual selection.

5 Conclusion

In this study we were motivated by the recently proposed operation-based prediction. Existing outcome-based predictors rely on regularities in the outcome stream so that they can accurately predict a large fraction of the program's outcomes. However, some outcomes do not exhibit sufficient regularity. Operation prediction has the potential of successfully predicting some of these outcomes. Operation prediction looks for regularity in the computation stream that produces outcomes that do not exhibit sufficient regularity. It works by dynamically extracting the computation slices that lead to such outcomes and by attempting to pre-execute them next time around. For operation prediction to be successful, it is necessary that the computation stream of such outcomes exhibits sufficient regularity.

Several works have demonstrated that operation prediction method work or may work for branches or loads, In this work we study program behavior and explain *why operation prediction may work*. In particular, we studied the locality of the computations that lead to otherwise unpredictable outcomes. We focused on loads and branches and studied locality under various realistic assumptions about slice detection. Moreover, we have studied models of opera-

tion predictors and how they interacted with an underlying outcome-based predictor. Our results demonstrate that high locality exists in the computation stream of unpredictable branches and of loads that miss in the data cache. Moreover, we have shown that the potential exists for operation prediction to boost accuracy over an existing outcome-based branch predictor and of accurately predicting the addresses of load references that would miss in the data cache. To the best of our knowledge no previous work on the locality of slices for mispredicted branches and loads exist.

While our results are promising we have not studied actual operation predictors taking timing into account. Nevertheless, we have seen that slices tend to spread over large region of the original instruction stream while they contain on the average few instructions. Moreover, our results remain valid and important as they demonstrate that programs do exhibit the behavior necessary for operation prediction to be successful. Further investigation is required in tuning operation predictors so that the make use of available resources effectively while being able to execute scout threads early enough for providing predictions for modern high-performance processors.

6 Acknowledgments

This research was supported by a Natural Sciences and Engineering Research Council of Canada (NSERC) research grant and by research funds from the University of Toronto. Tor Aamodt was supported by an NSERC PGS 'B' scholarship. We acknowledge the comments and suggestions of the anonymous referees that helped improve this paper.

REFERENCES

[1] M. M. Annavaram, J. M. Patel, and E. S. Davidson. Data Prefetching by Dependence Graph Pre-computation. In *Proc. 28th International Symposium on Computer Architecture*, July 2001.

[2] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt. Simultaneous subordinate microthreading (SSMT). In *Proc. 26th Intl. Symposium on Computer Architecture*, pages 186-195, May 1999.

[3] Y. Chou and J. Shen. Instruction path coprocessors. In *Proc. 27th Intl. Symposium on Computer Architecture*, pages 270-281, June 2000.

[4] J. D. Collins, H. Wang, D. M. Tullsen, C. J. Hughes, Y. Fong Lee, D. Lavery, and J. P. Shen. Speculative Precomputation: Long-range Prefetching of Delinquent Loads. In *Proc. 28th International Symposium on Computer Architecture*, July 2001.

[5] A. Farcy, O. Temam, and R. Espasa. Dataflow Analysis of Branch Mispredictions and Its Application to Early Resolution of Branch Outcomes. In *Proc. 31st Annual International Symposium on Microarchitecture*, Dec. 1998.

[6] C.-K. Luk. Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors. In *Proc. 28th International Symposium on Computer Architecture*, July 2001.

[7] S. Mehrotra and L. Harrison. Examination of a memory access clasification scheme for pointer-intensive and numeric programs. In *Proc. 10th Intl. Conference on Supercomputing*, Sept. 1997.

[8] A. Moshovos. *Memory Dependence Prediction.* Ph.D. thesis, University of Wisconsin-Madison, Madison, WI, Dec. 1998.

[9] A. Moshovos, D. N. Pnevmatikatos, and A. Baniasadi. Slice Processors: An Implementation of Operation-Based Prediction. In *Proc. International Conference on Supercomputing*, June 2001.

[10] A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. In *Proc. 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115-126, Oct. 1998.

[11] A. Roth, A. Moshovos, and G. S. Sohi. Improving virtual function call target prediction via dependence-based pre-computation. In *Proc. Intl. Conference on Supercomputing*, pages 356-364, June 1999.

[12] A. Roth and G. S. Sohi. Speculative Data-Driven Multithreading. In *Proc. 7th International Symposium on High Performance Computer Architecture*, Jan 2001.

[13] Y. Song and M. Dubois. Assisted execution. Technical report, Technical Report CENG-98-25, Department of EE-Systems, University of Southern California, Oct. 1998.

[14] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: Improving both performance and fault tolerance. In *Proc. 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

[15] C. Zilles and G. Sohi. Understanding the Backward Slices of Performance Degrading Instructions. In *Proc. 27th International Symposium on Computer Architecture*, pages 172-181, June 2000.

[16] C. Zilles and G. Sohi. Execution-Based Prediction Using Speculative Slices. In *Proc. 28th International Symposium on Computer Architecture*, June-July 2001.

[17] J. D. Collins, D.M. Tullsen, H. Wang, and J.P. Shen, Dynamic Speculative Precomputation. To Appear *In Proc. 34th International Symposium on Microarchitecture*, Dec. 2001.

[18] Tor Aamodt, Andreas Moshovos, and Paul Chow, The Predictability of Computations that Produce Unpredictable Outcomes, Technical Report #TR-01-08-01, EECG, University of Toronto, August 2001.

Hierarchical Multi-Threading For Exploiting Parallelism at Multiple Granularities

Mohamed M. Zahran
ECE Department
University of Maryland
College Park, MD 20742
mzahran@eng.umd.edu

Manoj Franklin
ECE Department and UMIACS
University of Maryland
College Park, MD 20742
manoj@eng.umd.edu

Abstract

As we approach billion-transistor processor chips, the need for a new architecture to make efficient use of the increased transistor budget arises. Many studies have shown that significant amounts of parallelism exist at different granularities that is yet to be exploited. Architectures such as superscalar and VLIW use centralized resources, which prohibit scalability and hence the ability to make use of the advances in semiconductor technology. Decentralized architectures make a step towards scalability, but are not geared to exploit parallelism at different granularities. In this paper we present a new execution model and microarchitecture for exploiting both adjacent and distant parallelism in the dynamic instruction stream. Our model, called hierarchical multi-threading, uses a two-level hierarchical arrangement of processing elements. The lower level of the hierarchy exploits instruction-level parallelism and thread-level parallelism, whereas the upper level exploits more distant parallelism. Detailed simulation studies with a cycle-accurate simulator indicate that the hierarchical multithreading model provides scalable performance for most of the non-numeric benchmarks considered.

Keywords: Speculative multithreading, Control independence, Microarchitecture, Thread-level Parallelism, parallelism granularity

1 Introduction

A defining challenge for research in computer science and engineering has been the ongoing quest for faster execution of programs. The commodity microprocessor industry has been traditionally looking to fine-grain or instruction level parallelism (ILP) for improving performance, by using sophisticated microarchitectural techniques and compiler optimizations. These techniques have been quite successful in exploiting ILP.

Many proposals such as the multiscalar [3][13], trace processing [8], superthreading [14], and clustered multithreading [2][6] have been proposed to reduce the cycle time and to exploit thread level parallelism. All of these proposals are geared to exploiting thread-level parallelism at one granularity. In this paper we investigate a hierarchical multithreading model to exploit thread-level parallelism at two granularities. It makes use of decentralization and multithreading to extract both fine- and medium-grain parallelism (also known as ILP and TLP). This execution model has the potential for better scalability of performance than non-hierarchical multithreading execution models.

The rest of the paper is organized as follows. Section 2 presents the background and related work. Section 3 describes the HMT (Hierarchical Multi-Threading) thread model. Section 4 presents a detailed description of the HMT microarchitecture. Section 5 presents experimental results obtained from detailed simulations of a cycle-accurate HMT simulator. Finally we conclude in section 6, followed by a list of references.

2 Background and Related Work

Limited size instruction window is one of the major hurdles in exploiting parallelism. Programs are hundreds of millions of instructions. Window size is only two or three dozens of instructions. In order to extract lots of parallelism, we need to have a large instruction window, which increases the visibility of the program structure at run time. However, having a very large instruction window is difficult, for the following reasons: (i) Implementation constraints limit the window size. (ii) Branch misprediction reduces the number of useful instructions in the instruction window. (iii) Having a large instruction window increases the complexity of the instruction scheduler, thus increasing the cycle time.

The central idea behind multithreading is to have

multiple flows of control within a process, allowing parts of the process to be executed in parallel. In the *parallel threads* model, threads that execute in parallel are control-independent, and the decision to execute a thread does not depend on the other active threads. Under this model, compilers and programmers have had little success in parallelizing highly irregular numeric applications and most of the non-numeric applications. For such applications, researchers have proposed a different thread control flow model called **sequential threads** model, which envisions a strict sequential ordering among the threads. That is, threads are extracted from sequential code and run in parallel, without violating the sequential program semantics. Inter-thread communication between two threads (if any) will be strictly in one direction, as dictated by the sequential thread ordering. No explicit synchronization operations are necessary. This relaxation makes it possible to "parallelize" non-numeric applications into threads without explicit synchronization, even if there is a potential inter-thread data dependence. The purpose of identifying threads in such a model is to indicate that those threads are good candidates for parallel execution in a multithreaded processor.

Examples of prior proposals using sequential threads are the multiscalar model [3][13], the superthreading model [14], the trace processing model [8], and the dynamic multithreading model [1]. In the sequential threads model, threads can be *nonspeculative* or *speculative* from the control point of view. If a model supports speculative threads, then it is called **speculative multithreading (SpMT)**. This model is particularly useful to deal with the complex control flow present in typical non-numeric programs. In fact, many of the prior proposals using sequential threads implement SpMT [3][5][6][8][13][14].

The speculative multithreading architectures discussed so far use a single level of multi-threading. The program is partitioned into a set of threads, and multiple threads are run in parallel using multiple PEs. The PEs are usually organized as a circular queue in order to maintain sequential thread ordering. A major drawback associated with single-level multithreading is that it is limited to exploiting TLP at one granularity only, namely the size of each thread. Thus, if it exploits fine-grain TLP, then it does not exploit more distant parallelism, and vice versa. In order to obtain high performance, we need to extract parallelism at different granularities.

A second drawback of single-level multithreading is that it is difficult to exploit control independence between multiple threads. If there is a thread-level misprediction, then all subsequent threads beginning from the mis-speculated thread are generally squashed, even though some threads may be control independent on the misspeculated one. It is possible to modify the hardware associated with the circular queue in order to take advantage of control independence [9], however the design becomes more complicated.

3 The HMT Thread Model

We investigate *hierarchical multithreading (HMT)* to overcome the limitations of single-level multithreading. This section introduces the software aspects of our HMT model; the next section details the microarchitecture aspects. An important attribute of any multithreading system is its thread model, which specifies the sequencing of threads, the name spaces (such as registers and memory addresses) threads can access, and the ordering semantics among these operations, particularly those done by distinct threads.

As our HMT work primarily targets non-numeric applications, we use SpMT as its thread sequencing model. However, threads are formed at two different granularities. The control flow graph is partitioned into *supertasks*, which are again partitioned into tasks. A task is a group of basic blocks and can have multiple targets. A supertask is a group of tasks at a macro level, which can be thought of as a bigger subgraph of the control flow graph. A supertask represents a substantial partition of program execution, the idea being that there is little if any control dependence between supertasks, and ideally only minimal data dependence. Generally, instructions in two adjacent supertasks are far away in the dynamic instruction stream and have a high probability of being mutually control independent. Thus, we have three hierarchical levels of nodes in a CFG: basic blocks, tasks, and supertasks.

The criterion used for this partitioning is important, because an improper partitioning could in fact result in high inter-thread communication and synchronization, thereby degrading performance! True multithreading should not only aim to distribute instructions evenly among the threads, but also aim to minimize inter-thread communication by localizing a major share of the inter-instruction communication occurring in the processor to within each PE. In order to achieve this, mutually data dependent instructions are most likely allocated to the same thread.

In this paper, tasks are formed as done for the multiscalar processor in [3]. Supertasks are dynamically generated as a collection of tasks. This is done as follows: the task predictor begins assigning tasks to PEs of a superPE. As soon as all the PEs of the superPE are running these tasks are assumed to be a supertask,

given a unique ID and stored in a specific table. For the time being, each supertask is composed of fixed number of tasks. Thus, is task prediction is used in order to generate the required number of tasks per supertask. This may not be the best strategy but is used for the time being in order to test our architecture.

4 The HMT Microarchitecture

In this section we describe one possible microarchitecture for our HMT thread model. In order to parallelly execute multiple tasks and supertasks in an efficient manner, we investigate a two-level hierarchical multithreaded microarchitecture. The higher level is composed of several *superPEs*, and is used for executing multiple supertasks in parallel. At the lower level of the hierarchy, each superPE consists of several PEs, each of which executes a task. Figure 1 presents a block diagram of the higher level of the hierarchy. The superPEs are organized as a circular queue, with head and tail pointers, such that at any point of time the active superPEs are between the head and the tail. A global sequencer assigns supertasks to the superPEs.

4.1 Program Execution in the HMT Processor

Initially, all the superPEs are idle, with the head and tail pointers pointing to the same superPE. The global sequencer assigns the first supertask to the head superPE, and advances the tail pointer to the next one in the circular queue. The *supertask successor predictor* then predicts the successor of the supertask just assigned. (Our experiments show that control flow between supertasks is highly repetitive.) In the next cycle, the predicted successor supertask is assigned to the next superPE. This process is repeated until all of the superPEs are busy. Currently we set the size of a supertask to be equal to X tasks where X is the number of PEs per superPE.

Each superPE executes the supertask assigned to it. Only the head superPE is allowed to physically commit, and update the architected state. All the others buffer their values, as will be shown in detail later. When the head superPE commits, the head pointer is advanced to the next superPE. At that time, a check is done to see if the supertask prediction done for the new head superPE is correct or not. If the prediction turns out to be wrong, then all the supertasks from the new head until the tail are squashed, and the correct supertask is assigned to the new head.

The above description is for the higher level of the hierarchy, the one involving supertasks and superPEs.

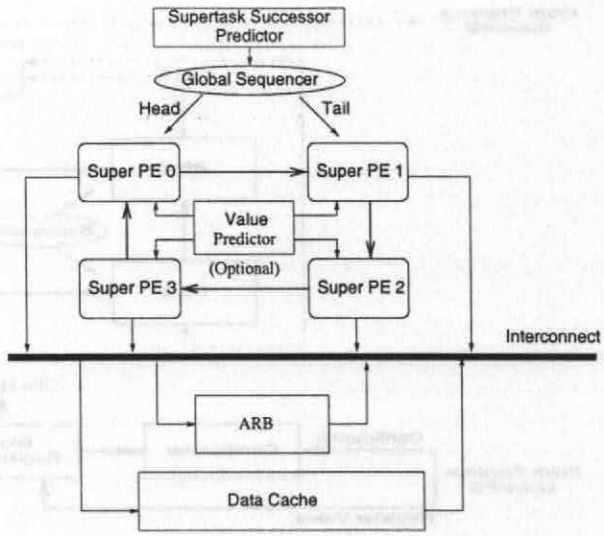


Figure 1: The HMT Processor

Next, we shall see how each supertask is executed within a superPE.

4.2 Lower Level of HMT Microarchitecture: SuperPE

The lower level of the HMT hierarchy considered in this paper is almost identical to a multiscalar processor, as described in [3]. The internals of this level are (briefly) described for the benefit of readers who are unfamiliar with the details of the multiscalar processor. Due to space limitations, this description is kept brief; interested readers are encouraged to consult [3] for the details. The internals of a superPE are shown in Figure 2. It consists of a group of PEs, each of which can be considered a small superscalar processor with a small instruction window, small instruction issue, etc. These PEs are connected as a circular queue with head and tail pointers, similar to the higher level. The circular queue imposes a sequential order among the PEs, with the head pointer indicating the oldest active PE.

A local sequencer with a local task successor predictor is responsible for assigning tasks to the PEs. When the tail PE is idle, a sequencer invokes the next task (as per sequential ordering) on the tail PE, and advances the tail pointer. When a task prediction is found to be incorrect, all subsequent tasks within the superPE are squashed; supertasks executing in subsequent superPEs are not squashed. Completed tasks are retired from the head of the PE queue, enforcing the required sequential ordering within the supertask. This retirement is *speculative*, if the superPE is not the current head of the higher level of the hierarchy.

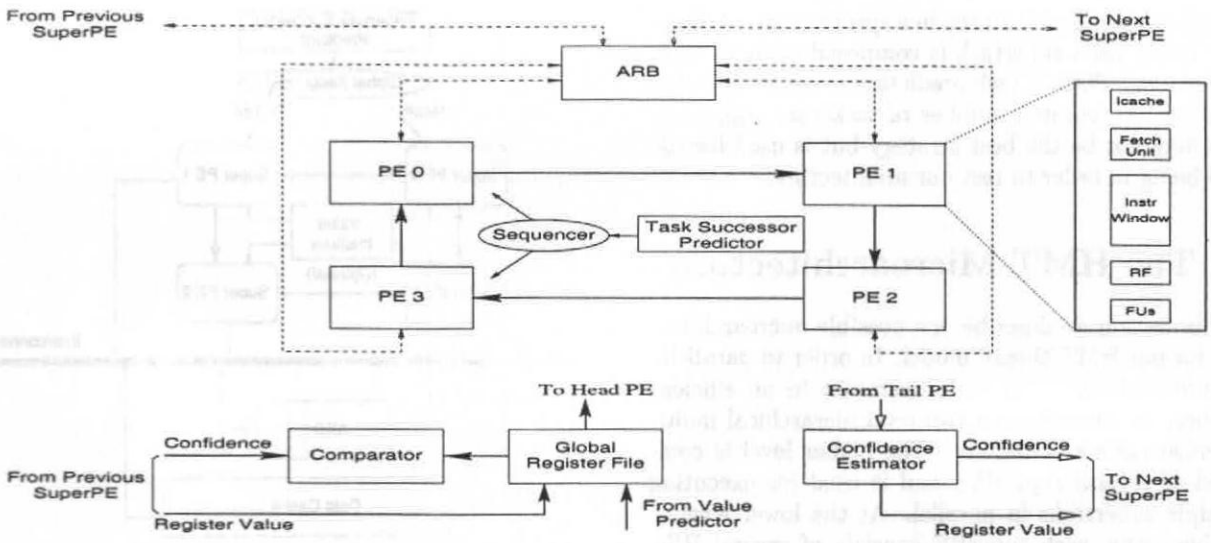


Figure 2: Block Diagram of a SuperPE

4.3 Inter-superPE Register Communication

Next, let us consider communication of register values from one supertask to another, at the higher level of the HMT hierarchy. As shown in Figure 2, each superPE maintains a *global register file* to store the supertask's incoming register values (from the previous superPE or the data value predictor). When a new register value arrives at a superPE, this value is compared against the existing value for that register in the global register file. If there is a change, then the new value is forwarded to its head PE, from where it gets forwarded to subsequent PEs, if required.

A superPE receives two sets of register values, one coming from the previous superPE (in case the current one is not the head superPE) and the other from the data value predictor. The choice between these values is done based on the confidence values. Each register value, whether coming from the value predictor or the predecessor superPE, has its own confidence value, and the superPE chooses the value with the highest confidence. Each superPE also has a confidence estimator to calculate the confidence values for the register values sent to its successor, as shown in Figure 2. Of course, if the predecessor superPE is the head and is committing, its values have the highest confidence.

The confidence estimations coming from the value predictor, are derived from the saturating counters of the predictor. The confidence of the values coming from the predecessor depends on the distance of the predecessor from the head superPE, the number of cycles since the assignment of the supertask to the predecessor as well as whether registers have been modified by the predecessor. In order to optimize inter-superPE register communication, we need to address two questions: (1) how often are register values sent to a superPE (whether from the data value predictor or from the predecessor superPE)? (2) which values should be sent?

For the first question, the following scheme is applied: the data value predictor predicts values for all registers only once for each supertask, and this is done at the time the supertask is assigned to a superPE. Each time a superPE wants to send values to its successor, it first calculates confidence for its values and then sends them. The receiving superPE will compare the confidence values of the new register values against the existing confidence values. If the new confidence value is higher for a particular register, then the new value is accepted. Register values are passed from a superPE to its successor when the successor superPE is about to speculatively commit its first task, which is not too early so the superPE does not use obselete values nor too late so the superPE would not have done a lot of useless work.

Next we address the second question of which value exactly to send. Sending all the register values has two drawbacks: (1) high bandwidth requirement, (2) low utility, as all registers may not have new values. Therefore, all registers are communicated only the first time. As time progresses since a supertask started execution, register values generated by that supertask tend to have higher confidence values and only modified registers are communicated to the subsequent superPEs. Also, all register values are communicated from the

head superPE when it is about to physically commit.

4.3.1 Data Value Prediction

The HMT microarchitecture can benefit from data value prediction, which involves predicting the incoming register values for a supertask. One option for the data value predictor is a hybrid predictor that uses 2-delta stride predictor [11] as well as context based predictor [12]. This hybrid predictor can be modified to predict all register values at once, in a way similar to [10].

Without the data value predictor, the only source of information available will be the premature register values coming from the predecessor superPE. Because these will most likely change during execution, the newly assigned supertask is likely to do a lot of useless work, if data value prediction is not used.

The job of the data value predictor is: given a supertask ID, predict *all* register values at the same time, together with confidence values for each predicted value. The predictor is updated each time a supertask is physically committed by the head superPE.

The data value predictor has two tables: SHT (Supertask History Table) and VPT (Value Prediction Table). Each SHT entry contains the following fields: (i)Frequency of accessing the entry, in order to use *least frequently used* replacement policy. (ii)Tag field, (iii)For each architected register it has: Last k values produced for this register, confidence estimator for the stride and predictor type (stride or context) that made the last prediction.

The last two values in the history of a register are used to make a prediction using a delta stride predictor. The confidence estimator of the delta stride predictor is simply a saturating counter that is incremented in case of correct prediction and decremented otherwise. The last k values are combined using a hash function and are used to index the VPT that contains k saturating counters for those values. The value with the highest counter is picked as the prediction of the context based predictor, with the corresponding counter as the confidence estimator. From the values predicted by the stride and context components, we pick the one with higher confidence. In the case of a misprediction, the counters corresponding to the predictor that made the last prediction are decremented. Similarly, in the case of a correct prediction, the corresponding counters are incremented. If the confidence estimators happen to be the same, one of the two values is selected at random.

4.4 Inter-SuperPE Memory Communication

At the higher level of the HMT hierarchy, inter-superPE memory communication is done by connecting the Address Resultion Buffers (ARBs) [4] of each superPE by a bidirectional ring. Thus, the memory data dependence speculation part is distributed at the higher level. When a load reference is issued in a superPE, and its ARB does not contain a prior store to the same location, the request is forwarded to the predecessor superPE's ARB, and so on. Similarly, when a store is issued in a superPE, it will be forwarded to the successor superPE's ARB, if no subsequent stores have already been issued to the same address from its superPE.

4.5 Advantages of the HMT Paradigm

Task mispredictions typically cause squashing only within its superPE. Data dependence violations only cause re-execution of the affected instructions.

The benefits of HMT stem from two important features: program characteristics and technological aspects. Program characteristics reveal the following: (i) Studies have shown that parallelism is there [7][15], but most of it cannot be exploited due to the large distance in the dynamic instruction stream. Our proposed architecture exploits some of the distant parallelism by executing supertasks in parallel. (ii) Multiscalar studies show IPC (Instructions Per Cycle) to be tapering off as more and more PEs are added [13]. This is because, in the case of a task misprediction, all the PEs starting from the PE with the misprediction will be squashed, thus decreasing the percentage of PEs doing useful work. We try to avoid this by letting each superPE have a small number of PEs and assign control-independent supertasks to multiple superPEs, as much as possible.

Also, if we squash a PE in a superPE, only the subsequent PEs in the same superPE are squashed; the remaining PEs in the other superPEs are not squashed (unless there is a change in successor supertask).

An important point to note is that the hierarchical arrangement of PEs as in the HMT microarchitecture does not require substantial additions or complexity to the hardware. The main newly introduced hardware is the confidence estimators for the register values and the comparators for comparing them. The data value predictor and the supertask predictor are two other newly introduced hardware structures. Apart from these, there is little new hardware. In fact, the two-level hierarchy can be dynamically reconfigured as a flat multithreaded processor, if required.

Default Values for Simulator Parameters			
PE		*Processor*	
Parameter	*Value*	*Parameter*	*Value*
Max task size	32 instructions		
PE issue width	2 instructions/cycle		
Task predictor	2-level predictor 1K entry, pattern size 6	*Supertask predictor*	2-level predictor 1K entry, pattern size 6
L1 - Icache	16KB, 4-way set assoc., 1 cycle access latency	*L1 - Dcache*	128KB, 4-way set assoc., 2 cycle access latency
Functional unit latencies	Int/Branch :- 1 cycle Mul/Div :- 10 cycles	*Data value predictor*	hybrid(stride, context), k=4 SHT 1K entries and VPT 64K entries

Table 1: Default Parameters for the Experimental Evaluation

5 Experimental Evaluation

The previous section presented a detailed description of an HMT microarchitecture. Next, we present a detailed quantitative evaluation of this processing model. Such an evaluation is important to study its performance characteristics, and to see how scalable this architecture is.

5.1 Experimental Methodology and Setup

Our experimental setup consists of a detailed cycle-accurate execution-driven simulator based on the MIPS-II ISA. The simulator accepts executable images of programs, and does cycle-by-cycle simulation; it is not trace driven. The simulator faithfully models the HMT architecture depicted in Figures 1 and 2; all important features of the HMT processor, including the superPEs, the PEs within the superPEs, execution along mispredicted paths, inter-PE & inter-superPE register communication, and inter-PE & inter-superPE memory communication have been included in the simulator. The simulator is parameterized; we can vary the number of superPEs, the number of PEs in a superPE, the PE issue width, the task size, and the cache configurations. Some of the hardware parameters are fixed at the default values given in Table 1. The parameters on the left hand side of the table are specific to a PE, and those on the right are for the entire processor. The successor predictor we use is similar to a two-level data value predictor [16].

For benchmarks, we use a collection of 7 programs, five of which are from the SPEC95 suite. The programs are compiled for a MIPS R3000-Ultrix platform with a MIPS C (Version 3.0) compiler using the optimization flags distributed in the SPEC benchmark makefiles. The benchmarks are simulated up to 100 million instructions each.

Our simulation experiments measure the execution time in terms of the number of cycles required to execute a fixed number of instructions. While reporting the results, the execution time is expressed in terms of instructions per cycle (IPC). The IPC values include only the committed instructions, and do not include the squashed instructions.

5.2 Experimental Results

Our first set of experiments are intended to show the benefits of the hierarchical arrangement. For this purpose, we simulate an HMT(3×4) processor (that is, an HMT processor with 3 superPEs, each of which has 4 PEs) as well as an HMT (1×12) processor, *both of which do not use data value prediction*, in order to show the potential of the HMT without the advantage of data value prediction. These results are presented in Figure 3. On the X-axis, we plot the benchmarks, and on the Y-axis, we plot the IPC values. Each benchmarks has two histogram bars, corresponding to HMT(3×4) and HMT (1×12), respectively.

The first thing to notice in Figure 3 is that the HMT(3×4) architecture is performing better than the non-hierarchical HMT(1×12) architecture for 5 of the 7 benchmarks. Looking at specific benchmarks, the HMT architecture performs relatively the best for `bzip2` and `ijpeg`. It performs relatively worse for `compress95` and `li`. Among these two, `compress95` does not have much parallelism. `li` has notable amounts of parallelism; on analyzing the results for `li`, we found that the prediction accuracy for procedure returns was low. We intend to investigate better predictors for predicting the return addresses in a hierarchical setting.

We next present some run-time statistics for the HMT(3×4) configuration; these statistics are somewhat different for the different configurations. Table

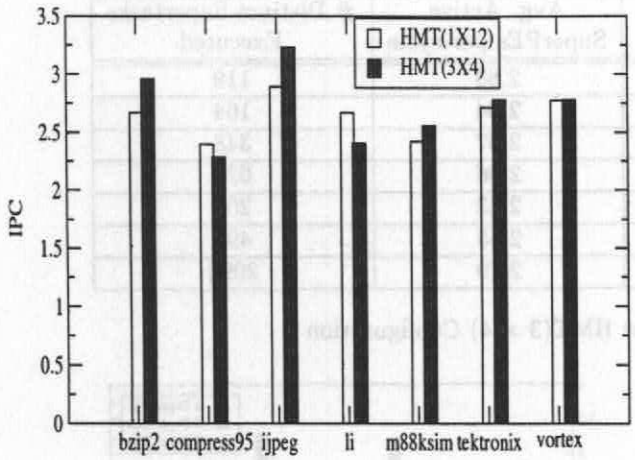


Figure 3: Performance of an HMT(3 × 4) Processor and an HMT(1 × 12) Processor, without Data Value Prediction

2 presents these statistics. In particular, it presents the data value prediction accuracy, supertask prediction accuracy, average active superPEs in a cycle, and the number of distinct supertasks executed.

Next, we perform sensitivity studies to study the performance of the HMT architecture under various conditions. In particular, we experiment with two different values for the number of PEs per superPE — 2 and 3. For each of these values, we vary the number of superPEs from 1 to 3. These studies use data value prediction at the higher level; performance does not scale very well when data value prediction is not used. Figure 4 presents the results of these sensitivity studies. The left graph is for 2 PEs/SuperPE and the right one is for 3 PEs/SuperPE.

From the results presented in Figure 4, we can see that except for `compress95` and `li`, all of the other benchmarks show good scalability in performance when the number of superPEs is increased from 1 to 3. That is, even when we use a total of 12 PEs, we still get reasonably good performance.

6 Summary and Conclusions

This paper presents a two-level hierarchical architecture that exploits parallelism at different granularities. The processing elements are organized in ring of rings, rather than a single ring. While each smaller ring (*superPE*) executes a sequence of tasks (one per PE) as in the Multiscalar, a high-level sequencer assigns *supertasks* to the superPEs. Such an architecture addresses several key issues, including maintaining a large instruction window, while avoiding centralized resources and minimizing wire delay.

Detailed simulation results show that for most of the non-numeric benchmarks used, the hierarchical approach provides better performance than the non-hierarchical approach. The results also show that a small percentage of the programs may not benefit from a hierarchical multithreading execution model.

The HMT microarchitecture presented in this paper is just one way to exploit parallelism at multiple granularities. Future work involves integrating compiler support. We intend to start with a post-compilation step to generate supertasks that are roughly of the same size, are somewhat control independent, and are somewhat data independent. We also intend to explore the memory hierarchy and memory disambiguation system to find the best model for the HMT model.

Acknowledgements

This work was supported by the U.S. National Science Foundation (NSF) through a CAREER grant (MIP 9702569) and a regular grant (CCR 0073582).

References

[1] H. Akkary and M. A. Driscoll, "A Dynamic Multithreading Processor," in *Proc. 31st Int'l Symposium on Microarchitecture*, 1998.

[2] P. Faraboschi, G. Desoli, and J. Fischer, "Clustered Instruction-level Parallel Processors," Tech. Rep., HP Labs, 1998.

[3] M. Franklin, *The Multiscalar Architecture*. PhD thesis, Technical Report 1196, Computer Science Department, University of Wisconsin-Madison, 1993.

[4] M. Franklin and G. S. Sohi, "ARB: A Hardware Mechanism for Dynamic Reordering of Memory References," *IEEE Transactions on Computers*, vol. 45, no. 5, pp. 552–571, 1996.

[5] V. Krishnan and J. Torrellas, "Executing sequential binaries on a clustered multithreaded architecture with speculation support," in *Int'l conf. on High Performance Computer Architecture (HPCA)*, 1998.

[6] P. Marcuello and A. Gonzalez, "Clustered Speculative Multithreaded Processors," in *Proc. Int'l conf. on Supercomputing*, pp. 20–25, 1999.

[7] I. Martel, D. Ortega, E. Ayguade, and M. Valero, "Increasing Effective IPC by exploiting Distant Parallelism," in *Proc. Int'l conf. on Supercomputing*, pp. 348–355, 1999.

[8] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. E. Smith, "Trace processors," in *Proc. 30th Annual Symposium on Microarchitecture (Micro-30)*, pp. 24–34, 1997.

Benchmark	Data Value Pred. Accuracy	Supertask Pred. Accuracy	Avg. Active SuperPEs per cycle	# Distinct Supertasks Executed
bzip2	83.4%	94.7%	2.88	119
compress95	46.3%	88.0%	2.64	164
ijpeg	69.9%	83.9%	2.67	348
li	65.0%	87.7%	2.06	628
m88ksim	91.6%	96.7%	2.83	209
tektronix	68.7%	89.3%	2.55	494
vortex	67.6%	74.7%	2.79	2994

Table 2: Run-Time Statistics for HMT(3 × 4) Configuration

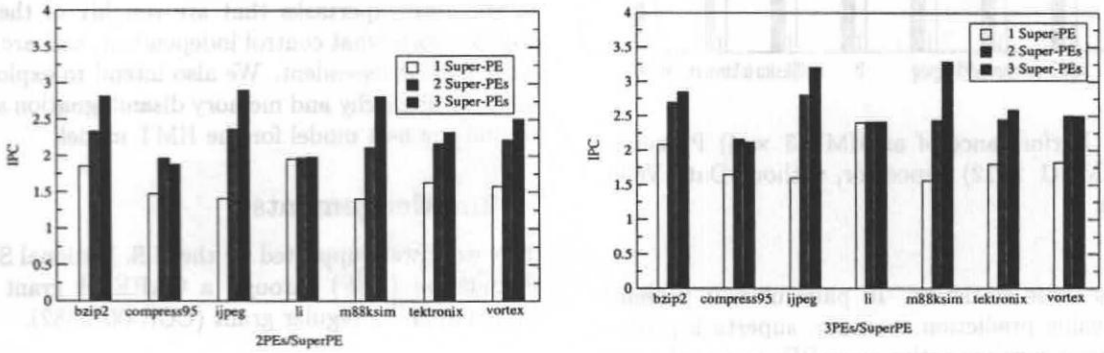


Figure 4: Performance of HMT Architecture for Different Configurations

[9] E. Rotenberg, and J. E. Smith, "Control Independence in Trace processors," in Proc. *32nd Int'l Symposium on Microarchitecture (Micro-32)*, 1999.

[10] R. Sathe, K. Wang, and M. Franklin, "Techniques for performing highly accurate data value prediction," *Microprocessors and Microsystems*, 1998.

[11] Y. Sazeides and J. E. Smith, "The Predictability of Data Values," in *Proc.30th Int'l Symposium on Microarchitecture*, pp. 248–258, 1997.

[12] Y. Sazeides and J. E. Smith, "Implementations of Context based Value Predictors," Tech. Rep., University of Wisconsin-Madison, 1997.

[13] G. S. Sohi, S. Breach, and T. N. Vijaykumar, "Multiscalar Processors," in *Proc. 22nd Int'l Symposium on Computer Architecture* , pp. 414–425, 1995.

[14] J.-Y. Tsai and et.al, "Integrating parallelizing compilation technology and processor architecture for cost-effective concurrent multithreading," *Journal of Information Science and Engineering*, vol. 14, March 1998.

[15] S. Vajapeyam, P. J. Joseph, and T. Mitra, "Dynamic Vectorization: A Mechanism for Exploiting Far-flung ILP in Ordinary Programs," in *Proc. 26th Int'l Symposium on Computer Architecture*, pp. 12–27, 1999.

[16] K. Wang and M. Franklin, "Highly Accurate Data Value Prediction using Hybrid Predictors," in *Proc.30th Int'l Symposium on Microarchitecture*, pp. 281–290, 1997.

Basic Mechanisms of Thread Control for On-Chip-Memory Multi-threading Processor

Takanori MATSUZAKI †, Hiroshi TOMIYASU ‡, Makoto AMAMIYA †

† Graduate School of Information Science and Electrical Engineering,
Kyushu University
6-1, Kasuga-Koen, Kasuga, Fukuoka, 816-8580, Japan
{takanori, amamiya}@al.is.kyushu-u.ac.jp

‡ Institute of Information Sciences and Electronics,
University of Tsukuba
1-1-1 Tennodai, Tsukuba, Ibaraki, 305, Japan
tomiyasu@is.tsukuba.ac.jp

Abstract

In this paper, we describe basic mechanisms of thread control for the FUCE processor. FUCE means FUsion of Communication and Execution. The goal of the FUCE processor project is to fuse the intra-processor execution and inter-processor communication. In order to achieve this goal, the FUCE processor integrates the processor units, memory unit and communication units into a single chip. The FUCE processor has mechanisms for pre-loading thread context and hiding memory access latency. With these mechanisms, no data cache memory is required, since memory access latency can be hidden due to a simultaneous multi-threading mechanism and the on-chip-memory system with broad-bandwidth low latency internal bus of the FUCE processor. This approach can reduce the performance gap between instruction execution, and memory and network accesses.

Keywords Multi-threading, pre-loading thread context, hiding of memory access latency, on-chip-memory processor, on-chip multi-processor.

1 Introduction

Currently, communication and VLSI device technologies are advancing towards higher and higher speeds and are becoming larger in scale. For example, optical-fiber transmission-line technology is now achieving Giga-bits/sec speeds and will achieve Tera-bits/sec speeds in the near future. New communication protocols, e.g., IP on WDM and IP on SONET, are now under development. In addition, hardware VLSI device technology is advancing to larger scale integration VLSI's with Giga-gate logic and Giga-bit memory on chip, and higher clock speeds of several Giga-Hz.

Software technologies including processor architectures, in contrast, are still developing within the conventional framework. The development of new architecture and software technologies is urgently required.

Against the background of these hardware and software technology trends, we are pursuing the FUCE project at Kyushu University. FUCE means FUsion of Communication and Execution. The main objective of this research is, as the name shows, to develop a new architecture that truly fuses communication and computation. The FUCE project aims to develop a new processor-architecture and kernel-software (operating system) for fusing computation and communications. We call the processor the FUCE processor, and the kernel-software CEFOS (Communication and Execution Fusion OS) [2]

The FUCE processor is an on-chip-memory processor developed based on a fine-grain multi-threading concept. In the FUCE processor, a thread is a tiny process executed without preemption. The fine-grain multi-threading technique promises high performance in fusing communication and internal execution. Both event handling, i.e., incoming/outgoing messages and I/O, and internal process execution are controlled by a uniform thread execution mechanism. The on-chip-memory processor technique also promises high performance in hiding memory access latency. No data cache memory is required in the FUCE processor, since the on-chip-memory system provides low latency memory accesses.

This paper introduces the FUCE processor and discusses the simultaneous multi-threaded execution mechanism and on-chip-memory system. Section 2 presents an overview of the FUCE processor. Section 3 discusses the FUCE process and FUCE threads. Section 4 covers the effect of hiding the memory access latency. Section 5 discusses the originality of the FUCE processor in comparison with related work.

2 Overview of the FUCE processor

The objective in designing the FUCE processor is to fuse the intra-processor execution and inter-processor communication so that the mechanism reduces the performance gap between intra-processor execution and inter-processor communication by integrating into one chip the execution units, communication units and memory unit.

An overview of FUCE Processor is shown in Figure 1.

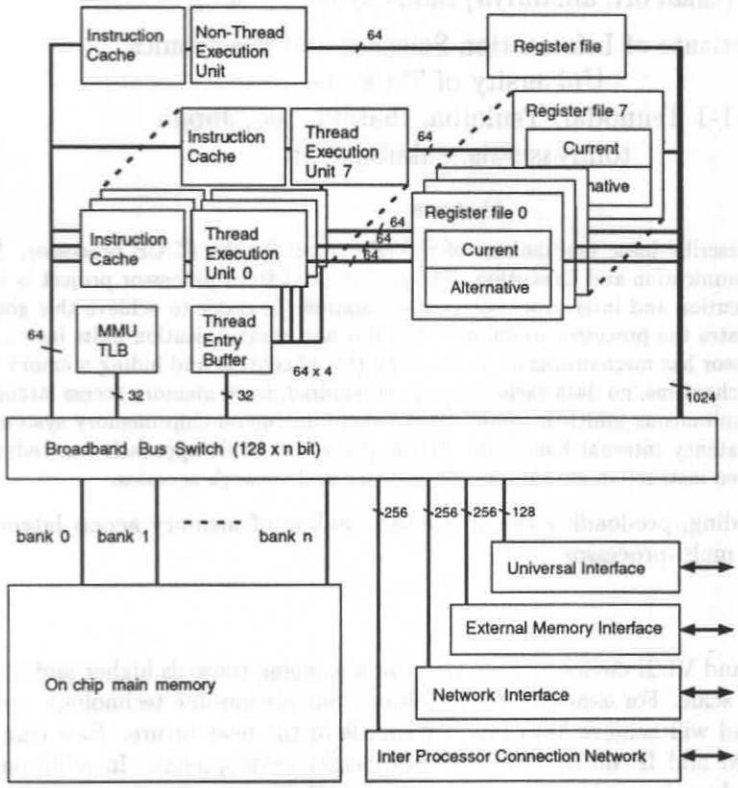


Figure 1: the FUCE Processor

The FUCE processor executes multiple threads in parallel and concurrently. It is designed as a multi-processor on a single chip to support high-speed multi-thread execution. In the on-chip multi-processor, multiple execution units, a ready thread queue control and event-handling units are incorporated into one chip. In addition to multiple execution units, a communication control unit and a main memory are also integrated into the chip. Another important design issue is the data path between execution units and memory units. The FUCE processor incorporates a broad-bandwidth low-latency internal bus. It also has external memory units (e.g. off-chip-memory or disks), which are connected to an external memory interface. It uses on-chip-memory as main memory and uses the external memory units as secondary memory.

Our design specifications for the FUCE processor are as follows:

1. Multiple units for thread execution.
2. Highly efficient thread execution.

3. Large size of register files.
4. High-speed broadband bus.
5. Large on-chip-memory.

The FUCE processor is being designed with future VLSI technology in mind. Table 1 shows the specifications of the FUCE processor. In the near future, VLSI technology will be able to achieve 800 Mega-transistors/chip with a chip size of $600mm^2$[11]. The FUCE processor uses half the area of the chip for on-chip memory. Also, we estimate that the number of transistors for a thread execution unit will be up to 5 Mega-transistors/unit, so 8-16 thread execution units and the non-thread execution unit will require 100 Mega-transistors. Therefore, the FUCE processor can integrate other units (e.g. communication control units and a broadband internal bus) into one chip.

Table 1: Specification of On-chip FUCE Processor

	2005	2010
Clock cycles	4 Giga-Hz	10 Giga-Hz
On-chip-memory Capacity	256 Mega-Bytes	1 Giga-Byte
On-chip-memory Speed	2 Giga-Hz	5 Giga-Hz
Internal Bus Speed	512 Giga-Bytes/sec	2.5 Tera-Bytes/sec

2.1 Execution Unit

The FUCE processor has two kinds of instruction execution units. One is the thread execution unit, and the other is the non-thread execution unit. The thread execution units execute threads without preemption. The non-thread execution unit executes non-thread code, which handles interrupts or exceptions for the OS kernel. Thread execution units have no preemption mechanism, while the non-thread execution unit can suspend and resume operations.

The FUCE processor has multiple thread execution units, each of which executes a thread independently. We believe that the memory system performance is more important than complicated high performance execution such as speculative execution and out-of-order execution. Therefore, the thread execution units of the FUCE processor are constructed with a simple pipeline structure. Here, the thread is defined as a sequence of instructions that is executed exclusively without any interruption except for some emergency cases such as infinite loops.

The features of the thread execution units are:

1. Thread execution units issue two instructions simultaneously and in order, and execute those two instructions in parallel.
2. Thread execution units transfer a set of registers in one instruction. We call this mechanism **block load/store**.
3. The execution pipeline of a thread execution unit will not stall while loading data from memory. We call this mechanism **non-blocking load**.
4. Thread execution units have two sets of register files: a current register file and alternative register file. The current register file is used for the thread execution in the foreground. The context of a ready thread is pre-loaded into the alternative register file in the background. The alternative register file takes the place of the current register file when the next ready thread runs.
5. Thread execution units have a thread entry buffer. This buffer is a hardware queue, which holds ready threads.

2.2 On-Chip-Memory

Current processors, which have off-chip memory, suffer from processor-pin bottleneck. It is therefore difficult to expand the memory buses of current processors. On the other hand, on-chip-memory processors scarcely suffer from processor-pin bottleneck, because on-chip-memory processors do not require

processor pins for the memory buses and it is easy to expand their memory buses. Furthermore, on-chip-memory processors make low-latency memory access possible. Low latency (e.g. around 4-8 cycles) and broadband (e.g. 512 Giga-Bytes/sec) data transfer are two features of the FUCE processor memory system.

The FUCE processor can also hide the memory access latency. In order to hide the memory access latency, it implements mechanisms that allow the thread execution units to access the on-chip-memory with low latency. These mechanisms are the block load/store and the non-blocking load. It can also reduce the overhead of memory access involved in switching the thread context thanks to the pre-loading of thread context.

The FUCE processor assumes that thread instructions are re-ordered by the compiler to pre-fetch the data. Re-ordering of instructions is such that the thread execution unit pre-fetches the data into a register during the thread execution. This pre-fetch instruction uses the non-blocking load and block load/store. This pre-fetch mechanism will avoid pipeline stall and reduce the idle time while the data are loaded into the register. In addition, the block load/store reduces the number of data transfer instructions.

With the memory units integrated into the chip, memory access becomes possible in four to eight cycles, and high-speed data transfer is performed between the execution unit and the memory unit. In addition, the FUCE processor has a mechanism for hiding the memory access latency, i.e. pre-loading of thread context. Note here that, even though the FUCE processor hides a low memory access latency (e.g. 4-8 cycles), it can not hide a large memory access latency of 100 cycles or more. Since the memory access to off-chip-memory takes too many cycles for the memory access latency to be hidden, the FUCE processor provides only on-chip-memory.

In fine-grain multi-threading, threads consist of small chunks of instructions and thread execution will terminate before data in cache memory are used more than once. In this situation, the gains afforded by cache memory are less than the overhead necessary for the cache memory control. But the FUCE processor can hide the memory access latency, and therefore no data cache memory is required for fine-grain multi-threading. On the other hand, the FUCE processor employs an instruction cache due to the principle of locality of instruction sequence in a thread.

In the FUCE processor memory system design, internal data transfer is more important than the data transfer between internal main memory and external memory.

3 FUCE process and FUCE threads

The basic structure of process and threads controlled in the FUCE processor is shown in Figure 2. The FUCE processor's process has more than one thread. The process is a unit of resource operation and the thread is a unit of processor assignment. Threads in the same process share elements of their processing environment such as a stack and a virtual memory space.

The basic features of the FUCE thread are as follows:

1. The FUCE thread is a fine-grain multi-thread. A large amount of the FUCE thread is assumed to run concurrently. Concurrent FUCE thread executions hide the communication latency.

2. The FUCE thread is a tiny process with no interruption. Therefore, the FUCE thread never suspends until it encounters its thread termination instruction. Furthermore, it has no limits in principle on its execution time.

3. The FUCE thread is a lightweight thread, and the lightweight thread can reduce the thread switching overhead.

4. The FUCE thread never accesses Off-Chip memory while it is running. Therefore, the FUCE thread is split when its execution has a large latency because it must access off-chip memory.

The FUCE processor can execute multiple threads belonging to different processes, enabling it to execute multi-threaded code on the multiple thread execution units. This approach can obtain sufficient ready threads to keep all of the thread execution units busy. If the FUCE thread has off-chip memory access, such as accessing external memory or accessing another node processor's memory, it will be separated into two threads, which are the caller thread and the recipient thread. In this way, the FUCE processor will be able to hide the latency of accessing off-chip memory.

3.1 Basic Mechanisms of Thread Control

The basic mechanisms of thread control are shown in Figure 3. To simplify the hardware mechanism of thread control, an operating system controls the thread execution in the FUCE processor. The thread execution control is performed through the software and hardware queue.

The operating system has one software queue, and controls the thread execution with the software queue. The software queue has threads, which are synchronized threads and can be executed immediately. Also, the software queue is able to have many threads which belong to another process. The operating system manages many processes, and each process has one thread scheduler.

The operating system controls the thread execution in the FUCE processor, synchronizing the threads. In order to synchronize the threads, the operating system has thread schedulers, which synchronize the threads in each process. The thread scheduler controls the thread execution order dynamically by synchronizing threads, and enqueuing synchronized threads in the software queue.

The hardware queue consists of the thread entry buffer. A Ready thread is registered in the thread entry buffer, and waits to be assigned to an idle thread execution unit. The operating system, when notified that the thread entry buffer is empty, moves a thread to the thread entry buffer. This notification is managed by hardware, but the transaction is controlled by software. The FUCE processor also has special instructions which control thread registration.

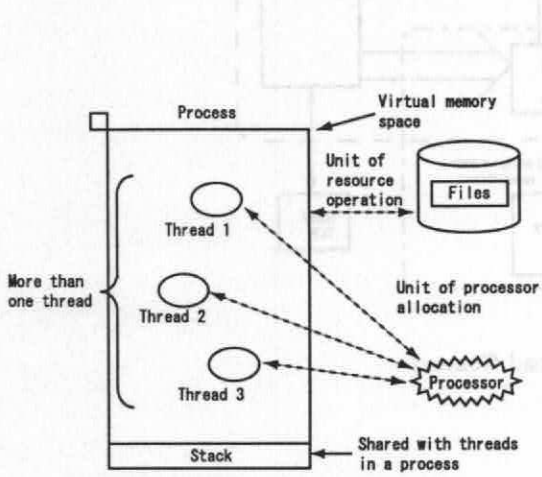


Figure 2: Basic Structure of FUCE process and FUCE threads

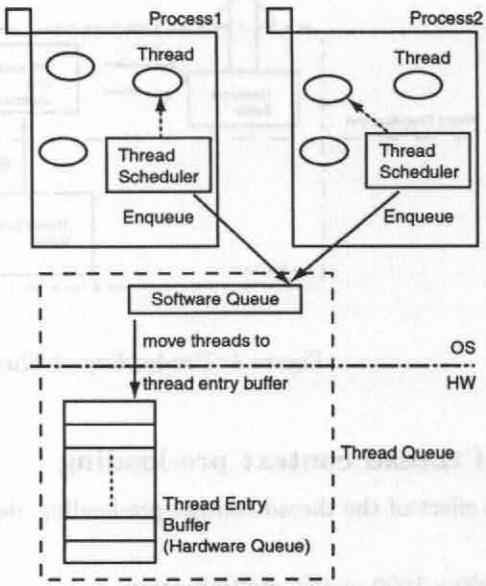


Figure 3: Basic Mechanisms of Thread Control

4 Hardware Support Mechanisms for Thread Execution

4.1 Thread Context Pre-loading

Context switches occur quite often in fine-grain multi-threading, and the overhead of a context switch is quite serious. In order to reduce this overhead, the FUCE processor pre-loads the thread context. In pre-loading, the processor uses two sets of register files and the thread entry buffer. The pre-loading mechanism allows the FUCE processor to achieve fast thread-context switches.

The FUCE Processor pre-loads the next thread context using the pre-loading unit, which is a special unit for the thread pre-loading. The FUCE processor has multiple thread pre-load units, each of which is connected to a thread execution unit. The pre-load units pre-load the next thread context with reference to the data included in the header of the next thread instructions. Figure 4 shows an overview of thread context pre-loading. The basic mechanism and function of thread context pre-loading are as follows:

1. A ready thread is assigned to an idle thread execution unit. The allocated thread uses the current register file when it begins to run.

2. When the thread execution unit begins to run the allocated thread, a new ready thread in the thread entry buffer is assigned to the thread execution unit.
3. While the thread execution unit is executing the allocated thread in foreground, it loads the thread context of a new ready thread, which is to be executed right after the current thread execution terminates, into its alternative register file in the background.
4. After the thread execution terminates, the thread execution unit exchanges the alternative register file and the current register file, and begins to execute the new thread.

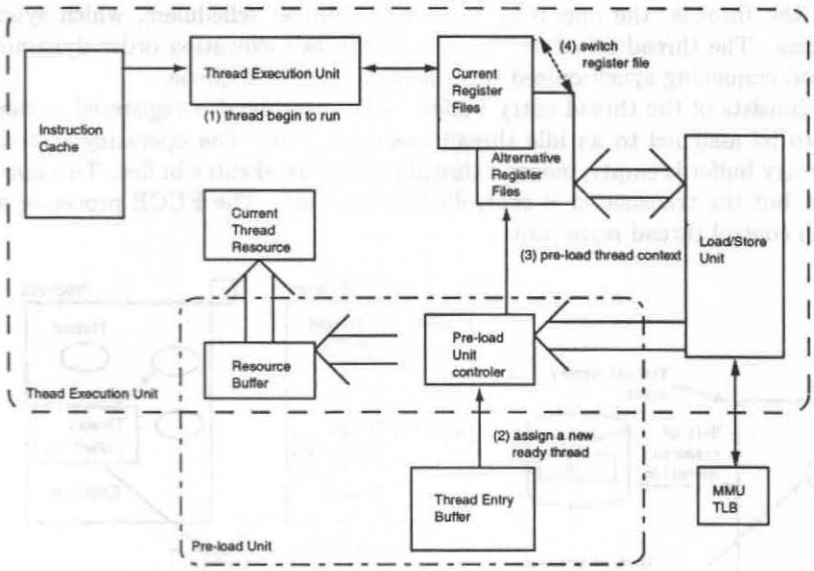


Figure 4: Pre-loading of Thread Context

4.2 Effect of thread context pre-loading

We evaluated the effect of the thread context pre-loading, described in section 4.1 using two benchmark programs:

- **Matrix:** 1000 × 1000 matrix multiplication.
- **8-Queens:** Finding all solutions of the 8-queens problem.

The specifications for the thread execution units used in the evaluation are shown in Table 2. In the evaluation, we used 8 thread execution units.

Table 2: Specifications of the thread execution units

Instruction Issuing Rate	2 instructions/clock-cycle
Number of Registers	64 × 2
Block-data per Transfer	4 blocks/instruction
Memory Access Latency	4, 6, 8, 10 cycles
Latency of Floating Point Execution	4 cycles

4.2.1 Matrix

The effect is examined for the matrix multiplication of 1000 × 1000 matrices. In this evaluation, we evaluated two cases of execution time and pipeline stall, and examined them for memory access latencies of 4, 6, 8 and 10 cycles: (a) With thread context pre-loading, (b) Without thread context pre-loading. Figure 5 shows the execution time and pipeline stall time.

In the Matrix problems, most of the instructions are load instructions, and the other instructions are multiplication and addition. The FUCE processor can load the data into the register without pipeline stall by using thread context pre-loading. So it can execute matrix problems with only a tiny pipeline stall (about 0.5%) (Figure 5: (a) With thread context pre-loading). On the other hand, It can not hide pipeline stall, when it does not use thread context pre-loading (Figure 5: (b) Without thread context pre-loading).

We can see from these results that thread context pre-loading makes a clear contribution to reducing pipeline stall and increasing the performance of thread execution.

4.2.2 8-Queens

We evaluated the execution time and the pipeline stall time in finding all solutions of the 8-queens problem. In this evaluation, on-chip memory access latency is 4 cycles. Figure 6 shows the execution time and the pipeline stall time.

In the 8-queens problem, the load instructions and the store instructions constitute about 35 percent of all the instructions. In addition, the load instruction and the store instructions are distributed. So, it is difficult for the FUCE processor to hide memory access latency (Figure 6: (a) Normal instructions). However, the FUCE processor can reduce on-chip memory access latency by separating instructions into threads and re-ordering thread instructions (Figure 6: (b) Re-ordering Threaded instructions). These approaches use thread context pre-loading.

We can see from these results that thread context pre-loading can reduce the execution time and pipeline stall time. This is because thread context pre-loading reduces the overhead of thread context switching.

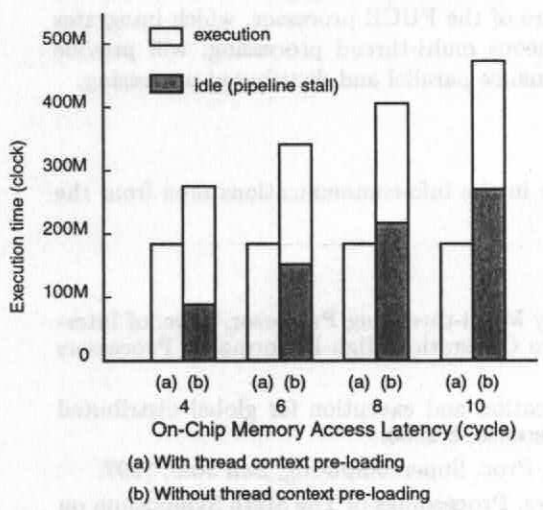


Figure 5: Matrix Multiplication

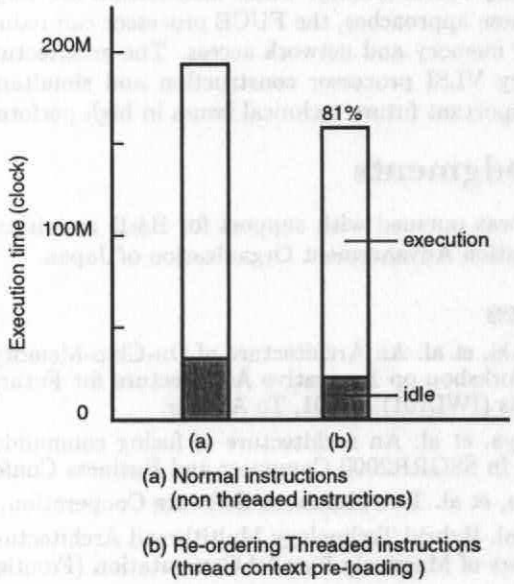


Figure 6: 8-Queens Problem

5 Related Work

The importance and feasibleness of the fine-grain multi-threading technique are generating interest in the parallel computing research field. For instance, the MTA machine has been commercialized[3] and the HTMT project is enthusiastically proceeding towards the goal of a Petaflops machine[4]. The concepts behind the FUCE machine have emerged from our research on multi-threading architecture and the parallel processing language V[5][6].

Because on-chip-memory processors achieve low latency and high-speed memory access, the on-chip-memory processor technique is being researched in the field of high performance computer systems. For

instance, PPRAM is an architectural framework for merged memory/logic ASSPs (Application-Specific Standard Products)[7]. Hydra[8] integrates multiple processors and cache memory on a single chip. The FUCE processor integrates not only multiple execution units and main memory but also communication control units into a single chip, in order to fuse communication and internal execution.

A great deal of research into on-chip multi-processors is also currently being pursued around the world. MP98[9], for example, supports efficient thread creation and execution through mechanisms involving inheritance of register values, resolution of data dependencies and speculative execution. However, these mechanisms make the hardware logic more complicated. The FUCE processor does not rely on such complicated mechanisms. SMT[10] is a simultaneous multi-threading processor, in which multiple threads are dynamically assigned to execution units at both the instruction level and the thread level. The FUCE processor is being developed based on the technique of fine-grain multi-threading and runs multiple threads concurrently in the multiple thread execution units. The FUCE processor is similar to SMT in that both rely on concurrent multi-thread execution. However, the multi-threading approach of the FUCE processor, in sharp contrast to that of SMT, is to assign each thread to a single execution unit and execute it exclusively on that execution unit in order to make the hardware logic simple and transparent.

6 Conclusion

This paper has discussed the basic mechanisms of thread control for on-chip-memory multi-threading processor architecture, and evaluated the effect of thread context pre-loading in the FUCE processor. In the FUCE processor architecture, we are making use of the technique of on-chip-memory processing. In addition, we are using thread execution support mechanisms, such as thread context pre-loading and the hiding of memory access latency. These mechanisms are very effective for thread execution.

Through these approaches, the FUCE processor can reduce the performance gap between instruction execution, and memory and network access. The architecture of the FUCE processor, which integrates on-chip-memory VLSI processor construction and simultaneous multi-thread processing, will provide solutions to important future technical issues in high performance parallel and distributed processing.

Acknowledgments

This research was pursued with support for R&D activities in the info-communications area from the Telecommunication Advancement Organization of Japan.

References

[1] T. Matsuzaki, et al. An Architecture of On-Chip-Memory Multi-threading Processor, Proc. of International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA01),1 2001, To Appear.

[2] M. Amamiya, et al. An architecture of fusing communication and execution for global distributed processing, In SSGRR2000 Computer and Business Conference, 8 2000.

[3] G. Alverson, et al. Tera Hardware-Software Cooperation, Proc. Supercomputing, San Jose, 1997.

[4] G. Gao, et al. Hybrid Technology Multithread Architecture, Proceedings of The Sixth Symposium on The Frontiers of Massively Parallel Computation (Frontiers '96), pp.98-105, October, 1996

[5] M. Amamiya, et al. Datarol:A Parallel Machine Architecture for Fine-Grain Multithreading, Proc. Third Working Conference on Massively Parallel Programming Models, London, 1997.

[6] M. Amamiya, et al. Co-Processor Design for Fine-grain Message Handling in KUMP/D, Proc. European Conference on Parallel Processing, Passau, pp.779-788, 1997.

[7] K. Murakami, et al. Parallel Processing RAM (PPRAM), Japan-Germany Forum on Information Technology, Nov. 1997.

[8] L. Hammond, et al. "The Stanford Hydra CMP", IEEE Micro, Vol. 20, No. 2, March/April 2000

[9] N. Nishi, et al. A 1GIPS 1W Single-Chip Tightly-Coupled Four-Way Multiprocessor with Architecture Support for Multiple Control Flow Execution, In Proc. ISSCC2000, WP25.5.

[10] Dean M. Tullsen ! $t al. "Simultaneous Multithreading: Maximizing On-Chip Parallelism," Proceedings of the 22nd Annual International Symposium on Computer Architecture, June, 1995.

[11] The International Technology Roadmap for Semiconductors (ITRS), http://public.itrs.net/

[12] Mosys, Inc. http://www.mosysinc.com/mhome/

Maximizing TLP with loop-parallelization on SMT

Diego Puppin
Massachusetts Institute of Technology
77 Massachusetts Avenue, NE43-618,
Cambridge, MA, 02139
diego@mit.edu
(617) 253-6284

Dean Tullsen
University of California, San Diego
9500 Gilman Drive
La Jolla, CA, 92093
tullsen@cs.ucsd.edu

Abstract

This paper describes research in exploiting loop-level parallelism on a simultaneous multithreading processor. We discuss some general and ad-hoc techniques for loop parallelization that proved to be effective with SMT, and how they were tuned for it. These techniques have been tested on the well-known Livermore loops, chosen for their variety of behaviors. The set of optimizations used produced significant improvement overall: we were able to improve average IPC from 2.72 to 3.97, and to gain an average speedup of 1.39 over optimized single-thread code, using up to eight threads.

We also describe a simple but effective method for determining the best number of threads to be used for parallel loops on a multithreaded processor. The model uses compile-time information to predict the most efficient point.

Keywords: simultaneous multithreading, loop parallelization, compiling

1 Introduction

The simultaneous multithreading (SMT) processor [12] is a computing paradigm that allows multiple threads to share the processing resources at the level of functional units each cycle. This allows thread-level parallelism to be exploited at a very fine granularity. The role of a parallel compiler for SMT is to extract parallelism, and to tune parallelization to take advantage of its peculiar resource sharing.

Parallelization on a SMT processor presents different challenges than a conventional parallel processor, due to the unique features of the processor. First, because threads share resources at such a fine level, increasing instruction-count to introduce parallelization can actually decrease performance if spare fetch and execution bandwidth is not available. Second, because all execution resources are available to even a single thread, increasing parallelism beyond the level that maximizes instruction-level parallelism, or saturates execution bandwith, is unnecessary and potentially harmful.

Thus, parallelization on SMT requires more careful tuning of parallelism and parallel optimizations, balancing the cost vs. benefit. Also, because thread-level parallelism is not constrained by memory layout (all threads share the same memory hierarchy), the compiler is free to optimize the number of threads used, and the methods of parallelization, independently for each loop in the program.

This paper is structured as follows. Section 2 describes some related work. Section 3 studies the effectiveness of techniques for loop-parallelization. Explored techniques include iteration interleaving, loop fusion, cyclic reduction, loop peeling, loop-invariant code motion, and local accumulation. Section 4 introduces a method for determining the best number of threads for a specific loop. The last section concludes and presents future work.

2 Related Work

In [13], effective techniques for fine-grained synchronization are discussed. SMT offers communication at the level of the L1 cache. The authors explain how to take advantage of this feature to parallelize tight loops that could not be parallelized on conventional parallel machines. Our work leverages some of their results.

In [6], Mitchell *et al.* were able to predict the performance of a few algorithms by measuring parameters such as data and register locality. However, prediction was strictly problem-dependent, and required a time-consuming data-fitting process. Because we limit our attention to determining the best number of threads, the method proposed here is simpler.

They also demonstrate the complex interactions between ILP-enhancing compiler optimizations and threading, also confirming that too much parallelization is not beneficial after the processor is saturated.

Other techniques that introduce novel approaches to creating thread-level parallelism on a multithreaded processor include slip-streaming [10], and speculative precomputation [2, 14]. However, this paper focuses on more traditional compiler-generated parallelism on a multithreaded processor.

The Cray MTA processor [1] features an advanced threading compiler which is capable of several of the transformations described here; however, the MTA is an LIW, cycle-interleaved multithreading processor; thus, it represents a significantly different execution model with less intimate sharing of and competition for resources between threads.

3 Effectiveness of loop-parallelization techniques

The first part of this research presents case studies of loop-parallelization with the SMT processor. The Livermore loops have been chosen for this purpose because of their wide availability, their well-known features, and their generality and variety. We apply some standard and some ad-hoc parallelization techniques to the various kernels in order to understand which are effective, and what is the performance gain. The goal is to show that a compiler can effectively target parallelization on the SMT processor. For the simulations, the parameters used are the same used in [11]. As explained there, these parameters describe a likely next-generation SMT processor, with out-of-order instruction execution, 8-wide fetch and execute, 2-level on-chip caches, and hardware support for 8 threads. In some instances, we also simulate 16 threads to verify that some benchmarks have optimal points beyond the limits of our machine.

Parallelization was performed at the high-level code (C source). All the kernels were rewritten manually, using general principles. We tried to emulate the work of an advanced compiler in all of our transformations.

Parallelization techniques used here include general loop restructuring, such as loop fusion, loop peeling, invariant code motion, and some more advanced techniques aimed at this architecture. These include interleaving, cyclic reduction, and local accumulation.

In the following analysis, two principal metrics will be discussed: processor utilization and completion time. Even if completion time is the most important

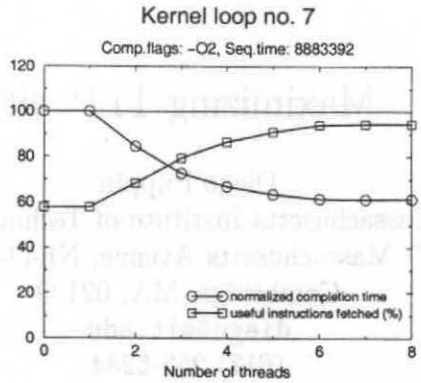


Figure 1: Execution statistics for kernel 7

value when discussing the effect of multithreading, processor utilization is a useful description of how well processor resources are exploited: we cannot expect very high improvement in terms of completion time if processor utilization is high for the sequential version. Processor utilization is measured as the average percentage of useful instructions fetched relative to the total bandwidth (8 instructions per cycle). Completion time is measured as a percentage of the time taken by the sequential version. For both metrics, the values corresponding to 0 threads in the plots refer to the sequential version, as opposed the the single-thread version of the parallel code, which is the 1-thread result.

All simulation was done using the SMTSIM [11] simulator, running Alpha executables and compiled with gcc at the highest level of optimization.

3.1 Independent iterations

The Livermore loop kernels can be classified into four groups: independent-iteration loops, loop-carried dependence loops, accumulation loops, and large loops. This and the following sections will present overall results for all the loops of each type, as well as specific discussion of loops that either had typical or noteworthy behavior.

Loops with independent iterations are basically vector computations that can be carried on independently for every element. They are easy to parallelize, using *iteration interleaving*. This consists in assigning iterations to threads not in blocks, but interleaved. As shown in [5], in these cases this is the most efficient way to express parallelism: better cache and TLB utilization is reached with this solution.

These kind of loops are easily run on the SMT processor, with good speedup. Figure 1 shows statitics

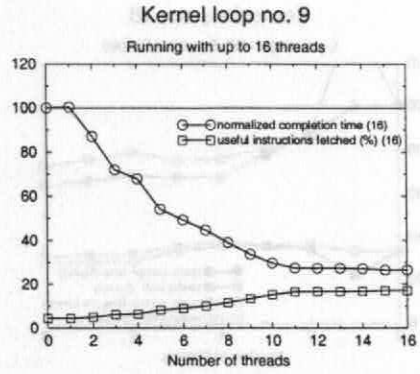


Figure 2: Execution statistics for kernel 9

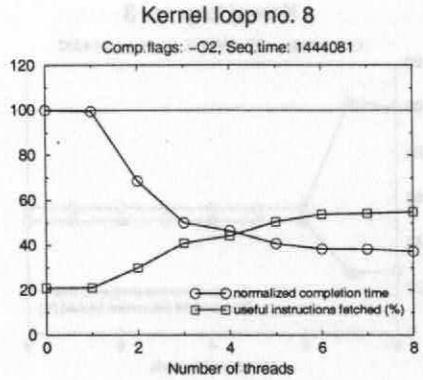


Figure 3: Execution statistics for kernel 8

for kernel 7 as an example. Completion time asymptotically decreases to 60% of sequential time, achieving much higher processor utilization. We cannot expect any more improvement, as almost all the processor bandwidth (92%) is taken by useful instructions. On average, these loops, when parallelized achieve about 1.5 speedup.

The highest available parallelism is found in kernel 9. This code suffers from very bad Dcache utilization (only 70% to 80% Dcache hit ratio) and very high average memory delay (116.5 cycles). When more threads are running, data are moved into the shared cache by the first thread: the other threads will find their data ready in the cache, due to the interleaving, with an effect similar to prefetching. The 8-thread version has an average memory delay of just 46.2 cycles, and runs 2.5 times faster. In figure 2, results are shown for up to 16 threads, which will be discussed further in section 4.

We should note that loop 8 is in this category (independent iterations) with some compiler assistance. In this kernel, some temporary values are stored tidily in a vector, but are never used outside the iteration that created them. We assume that a compiler can determine this fact using some simple techniques (comparing indices...), and then introduce suitable temporary variables, local to each iteration. This transformation makes iterations independent. This change proved to be effective with sequential code as well. The multithreaded code is more than twice as fast. Figure 3 shows this result, with the non-multithreaded code also taking advantage of this optimization.

3.2 Loop-carried dependence loops

Loops featuring loop-carried data dependences (Livermore loops 5, 11, 19, 20, 23) are more difficult to

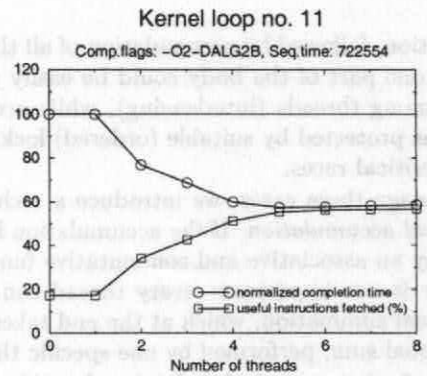


Figure 4: Execution statistics for kernel 11

parallelize: interleaving is not useful here, as iterations need to be executed strictly in order. Also, loop skewing failed due to high overhead.

Nonetheless, kernel 11 was successfully parallelized using *cyclic reduction* [4], a powerful algorithm for the running sum problem, which was tuned to SMT by reducing the level of recursion to 2.

The initial low processor utilization on this kernel offers large opportunity to introduce advanced techniques. Even if more instructions are executed, about 1.7 times as much in this case, the increase in available TLP allows much better instruction throughput, overcoming the large overhead. The multithreaded version is almost twice as fast (see figure 4). We expect this optimization to be very useful with other kinds of loops.

3.3 Accumulation loops

Livermore kernels no. 3, 4, 6, and 13 are particularly interesting, as they feature some independent

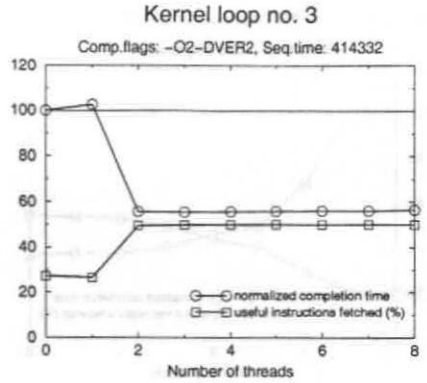


Figure 5: Execution statistics for kernel 3

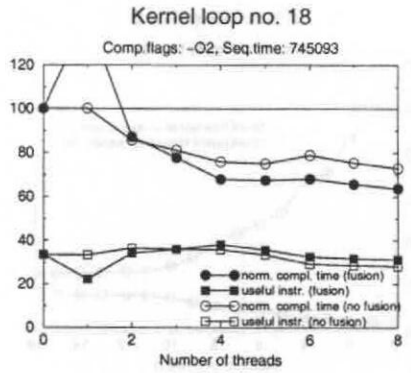


Figure 6: Execution statistics for kernel 18

computation, followed by accumulation of all the values. So, one part of the body could be easily parallelized among threads (interleaving), while accumulation was protected by suitable (ordered) locking to prevent critical races.

To manage these cases, we introduce a technique called *local accumulation*. If the accumulation is carried on by an associative and commutative function, the order is not important: every thread can compute a local summation, which at the end takes part in the global sum, performed by one specific thread.

Figure 5 shows typical behavior for this category (in this case, loop 3). The two-thread version presents a boost in performance with respect to sequential code (1.6 speedup), but after this, the global sum introduces sequentiality, which makes further threads useless. We verified that a tree reduction for summation is not effective, as its cost overwhelms the increased ILP.

It is interesting to note that the optimizations introduced are effective sometimes even if just one thread is running. The MT version of kernel 6 with one thread runs faster than the original sequential version (see table 1). In this case, we introduced a temporary variable to store the local summation, the value of which was then summed to the final results. This transformation allowed better register allocation and memory usage even when no actual TLP was exploited.

3.4 Larger loops

Five of the 24 Livermore loops featured larger, more complex loops, with complex branching and data-dependent memory-accesses. In these cases, the general technique was to interleave iterations, introducing ordered locking to protect possible dependences. We expect that a compiler will not be able to improve the performance significantly due to the difficulties of analyzing the complex code. In these cases, the programmer may be able to give more directions to the compiler, e.g. by augmenting the code with some compilation directives.

Nonetheless, some other standard techniques proved useful in some cases. The three loops composing kernel 18 are just a smart splitting of a larger loop, to increase ILP. When loops are fused, more opportunity for thread-level parallelism emerges, which is exploited when more threads are used. This solution is not good with few threads, because the increased ILP provided by loop distribution is greater than the TLP provided by loop fusion.

In figure 6, a comparison between the two versions (loop fusion and loop distribution) is given. To achieve the best performance at any number of threads, the compiler would have to produce multiple versions of the code, which could be selected at runtime based on the number of hardware contexts available. This idea is gaining importance for traditional processors also, under the broader denomination of *feedback-directed optimization* [7].

Another note is about kernel 24. This loop scans a vector looking for the first minimum. It was parallelized successfully (see figure 7) considering the minimum as an accumulation, using local variables to store the local minimum for each thread. This type of restructuring can be debatable, as this requires the compiler to recognize that the test `x[k] < x[m]` is a way to compute an accumulation function. but we believe that the usage of a library function `min` could make this transformation automatic (some libraries, such as *MPI*, feature specific parallel implementations for the minimum).

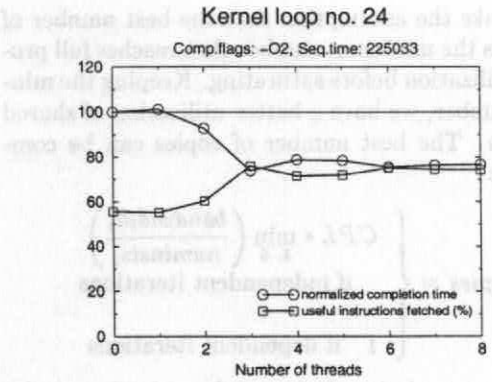


Figure 7: Execution statistics for kernel 24

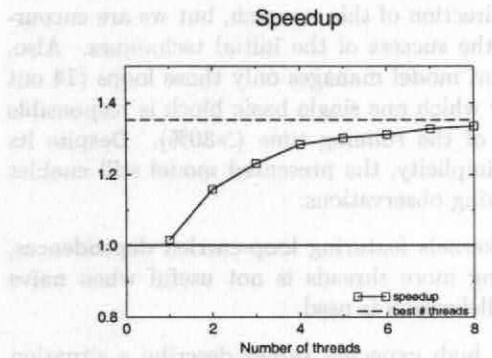


Figure 8: Average speedup for the Livermore loops

3.5 Overall speedup

In most cases, good speedups were achieved by applying varied optimization techniques appropriate to each loop. Overall results obtained with these techniques are given in figures 8, where the dashed line represents the speedup that can be reached choosing the best number of threads independently for every kernel. With the techniques discussed here, we were able to achieve significant speed-up: IPC is increased on average from 2.72 to 3.97, and completion time exhibits speedups averaging 1.39.

In this work, we assume a low-cost thread spawning mechanism, or that, in parallel code, threads available for parallel execution would be waiting on synchronization variables between parallel loops. This latter model can be implemented with low overhead, due to the fast on-chip communication offered by SMT.

Table 1 shows the results with more details. The *type* column represents the category in which the loop falls: independent iterations (IND), loop-carried

loop	type	1	2	4	8	best	IPC	MT IPC
1	IND	100	100	100	100	100	5.32	5.34
2	LRG	100	100	100	100	100	6.10	6.10
3	ACC	100	56	56	57	55	2.18	3.94
4	ACC	100	65	67	72	65	2.48	3.80
5	DEP	100	100	100	100	100	1.22	1.22
6	ACC	87	68	60	64	60	1.34	2.25
7	IND	100	84	67	61	61	4.62	7.53
8	IND	100	69	47	37	37	1.68	4.50
9	IND	100	98	67	37	37	0.35	0.93
10	IND	100	91	86	90	86	2.56	2.97
11	DEP	100	77	60	59	57	1.39	2.42
12	IND	100	76	76	75	75	4.92	6.56
13	ACC	100	68	41	39	39	1.53	3.95
14	LRG	100	100	84	83	82	3.17	3.86
15	IND	100	82	76	73	72	4.86	6.72
16	LRG	100	100	100	100	100	1.21	1.21
17	IND	100	51	27	25	24	1.68	6.90
18	LRG	100	87	68	64	64	2.68	4.21
19	DEP	100	100	100	100	100	1.01	1.01
20	DEP	100	100	100	100	100	0.70	0.70
21	IND	100	100	97	99	96	6.11	6.37
22	IND	100	84	71	46	46	2.54	5.57
23	DEP	100	100	100	100	100	1.26	1.26
24	LRG	100	93	79	77	74	4.46	6.03
average		99	85	76	73	72	2.72	3.97
speedup %		100.58	117.19	131.28	136.75	138.62		

Table 1: Experimental data for the Livermore loops

dependences (DEP), accumulation (ACC) and large loops (LRG). The table reports completion time and average speedup for the parallel code with 1, 2, 4 and 8 threads (results for the other numbers of threads are not shown for simplicity), normalized with respect to the sequential time, set equal to 100. We also set completion time equal to 100 if the MT code was actually slower, as we can always run sequential code if needed. The table then reports the completion time and average speedup reached with the best number of threads. It shows also IPC for the sequential code and useful IPC for the best number of threads, that is computed as:

$$useful\ IPC = sequential\ IPC * speedup$$

This way, we do not count any overhead introduced by multi-threading, we measure only how efficiently the useful instructions of the sequential code were run by the parallel code. In 7 cases we were able to have more than 6 useful IPC, out of a total bandwidth of 8. For instance, while kernel 11 reaches an actual processor utilization of about 60%, i.e. more than 4.5 IPC, useful IPC is only 2.42.

As the reader can observe, the best performance is not always reached with eight threads. For accumulation loops, for instance, a few threads are usually enough. Therefore, it is crucial to be able to determine the right number of threads to best exploit the available resources.

There are two reasons to limit thread use to the optimal point (or below). In some cases, performance decreases significantly when more threads are used. Second, from a system-level view, this presents the system with more options to use the idle threads for other purposes. Even if we determine that an application will saturate the processor (because it saturates

a particular resource, for example the floating point execution units), system-level performance could still be improved if other threads are introduced which do not bottleneck on the same resource. Techniques for identifying such threads are discussed in [9].

In the next section, we introduce a method that allows the compiler to determine the optimal number of threads to use for each loop.

4 Determining the best number of threads

Let's call *critical path length* (CPL) the length of the instruction critical path (longest chain of dependent instructions) within a basic block. To compute the CPL, we consider instruction latency. For memory operations, we use average memory access time as measured by the simulator on the sequential version of each kernel.

We wrote a small program that allowed us to collect CPL figures automatically, using Atom [3], a tool able to augment binary Alpha code with analysis routines. Atom is also used to determine which are the most stressed basic blocks, in the discussion that follows.

If the block represents a loop body, a single iteration will take at least CPL cycles. Not all the functional units will typically be used in this time: many of them may remain empty due to low ILP. We can have a performance gain if we can squeeze another copy of the loop body into the schedule, filling the empty slots.

A higher number of threads does not in general guarantee higher performance. Further performance improvement is limited if one of the functional units is *saturated*. In our terminology, this means that the number of instructions of a given type cannot be executed within CPL cycles by the available resources. In particular, we track the number of integer, load-store, synchronization and floating point functional units, as well as total issue bandwidth (five different counts). If a functional unit is not saturated, more copies of the body loop can run together as different SMT threads, as long as iterations can be executed partially or fully in parallel.

$$saturation_i \text{ if } numinsts_i > CPL * bandwidth_i$$

where $bandwidth_i$ is the bandwidth (available functional units) of instructions of type i, and $numinsts_i$ the count of instruction of type i. In this case, we break all instructions into the 5 categories just described.

We make the assumption that the best number of threads is the minimum number that reaches full processor utilization before saturating. Keeping the minimum number, we have a better utilization of shared resources. The best number of copies can be computed as:

$$best\ copies = \begin{cases} CPL * \min\limits_{1..5}\left(\frac{bandwidth_i}{numinsts_i}\right) \\ \quad \text{if independent iterations} \\ \\ 1 \quad \text{if dependent iterations} \end{cases}$$

This approach has been tried, with the results shown in table 2.

Refinement of this model to have a more precise approximation of the best number of threads is a continuing direction of this research, but we are encouraged by the success of the initial techniques. Also, the current model manages only those loops (14 out of 24) for which one single basic block is responsible for most of the running time (>80%). Despite its current simplicity, the presented model still enables the following observations:

- For kernels featuring loop-carried dependences, adding more threads is not useful when naïve parallelization is used.
- Very high expected values describe a situation in which increasing the number of threads is useful; particularly interesting is kernel no. 9, which scales up to 16 threads (see figure 2): the model says that it would scale well even to a really higher number of threads.
- Low expected values can be observed when the processor-utilization curve features a minimum; these situations require careful tuning of the number of threads.

The most unexpected result is given by kernel no. 6, which does not scale beyond 4 threads. Its low performance is mostly determined by a very high memory latency, the effects of which are hidden by fewer threads than expected by the model.

Giving the compiler control over the number of threads created, as well as the tools to choose the right number, can enable significantly higher performance than naïvely creating the maximum number of threads. This maximizes both per-application performance and system-level performance (not shown directly in this work) when the other thread contexts are made available for other purposes. This ability is maximized if the processor has the ability to dynamically allocate and deallocate threads during the execution of the program without high overhead.

Kernel	1	3	4	5	6	7	8
Actual	8	2	2	1	4	7	8
Expect.	8.8	5.1	5.1	1	17.4	6.3	5.5

Kernel	9	10	11	12	13	21	23
Actual	8(16)	7	1	2	5	3	1
Expect.	35.3	9.2	1	4	3.3	6.0	1

Table 2: Actual best number of threads compared with expected best number of copies

In this direction, we believe that this model can be used to predict symbiosis [8] effects. The model can tell which units are saturated and which are not, and how many threads are best if the available resources are reduced, in a certain sense predicting the interaction of two multi-threaded programs. If the two programs have different requests and they can run at their best with a limited number of threads, we can expect them to run together effectively. This will be an interesting topic of future research.

5 Conclusion and Future Work

We have shown that standard and ad-hoc parallelization techniques can improve significantly the performance of loops on an SMT processor: IPC increasead on average from 2.72 to 3.97, and completion time achieves an average speedup of 1.39 over the sequential version, using up to eight threads.

The demonstrated optimizations should be within the capabilities of a reasonable compiler. The Cray MTA compiler already does some of these optimizations for that architecture.

We have also shown a model to determine the best number of threads for a given loop. We were able to make predictions about the number that offered the best performance for specific loops, using compile time information. We believe that a compiler featuring such a model can boost the performance by running each loop with the most appropriate number of threads. We also believe that the model can improve system-level performance, predicting which programs can run together with good symbiotic effect.

Future work will be in the direction of developing and implementing an advanced parallelizing compiler for the SMT. This will require support of automatic rewriting for loops, using techniques such as those shown here, and a performance model that best exploits the TLP features of the architecture.

Acknowledgements

The authors would like to thank the reviewers for their useful comments. This work was funded in part by NSF Career award MIP-9701708 and equipment grants from Compaq Computer Corporation.

References

[1] R. Alverson, D. Callahan, D. Cummings, and B. Koblenz. The tera computer system. 1990 International Conference on Supercomputing, September 1990.

[2] J. D. Collins, H. Wang, D. M. Tullsen, C. J. Hughes, Y. Lee, D. Lavery, and J. P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. 28th International Symposium on Computer Architecture, 2001.

[3] Compaq. Alpha c compiler libraries: Atom. Online documentation.

[4] P. M. Kogge and H. S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers*, C-22(8):786–793, August 1973.

[5] J. L. Lo, S. J. Eggers, H. M. Levy, S. S. Parekh, and D. M. Tullsen. Tuning compiler optimizations for simultaneous multithreading. 30th Annual International Symposium on Microarchitecture (Micro-30), December 1997.

[6] N. Mitchell, L. Carter, J. Ferrante, and D. Tullsen. Ilp versus tlp on smt. Supercomputing '99, November 1999.

[7] M. D. Smith. Overcoming the challenges to feedback-directed optimization. ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo 00), Boston, MA, January 2000.

[8] A. Snavely, N. Mitchell, L. Carter, J. Ferrante, and D. Tullsen. Explorations in symbiosis on two multithreaded architectures. Workshop on Multithreaded Execution, Architecture, and Compilation, January 1999.

[9] A. Snavely and D. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. Architectural Support for Programming Languages and Operating Systems, pages 234–244, November 2000.

[10] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: Improving both performance and fault tolerance. Architectural Support for Programming Languages and Operating Systems, pages 257–268, November 2000.

[11] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. 23rd Annual International Symposium on Computer Architecture, Philadelphia, PA, May 1996. Reprinted in Readings in Computer Architecture.

[12] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. 22nd Annual International Symposium on Computer Architecture, Santa Margherita Ligure, Italy, June 1995. Reprinted in 25 Years of the International Symposia on Computer Architecture: Selected Papers, 1998.

[13] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. M. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. 5th International Symposium on High Performance Computer Architecture, January 1999.

[14] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. 28th International Symposium on Computer Architecture, 2001.

Speculative Multithreading: from Multiscalar to MSSP

Guri Sohi

My research group at Wisconsin has been working on speculative multithreading techniques for over a decade. This talk will overview some of what we have learned over the years. We will start with our early work on multiscalar, continue with data-driven multithreading and speculative slices, and then on to our most recent work on master-slave speculative parallelization.

Bio:

Guri Sohi teaches computer architecture at the University of Wisconsin-Madison. He joined the Wisconsin faculty after receiving his Ph.D from the University of Illinois in 1985, and is currently a Professor in both the Computer Sciences and Electrical and Computer Engineering departments. Sohi's research has been in the area of architectural and microarchitectural techniques for high-performance microprocessors, including instruction-level parallelism, out-of-order execution with precise exceptions, non-blocking caches, decentralized microarchitectures, speculative multithreading, and memory dependence speculation. He received the 1999 ACM SIGARCH Maurice Wilkes award for contributions in the areas of high issue rate processors and instruction level parallelism.

Branch-Prediction in a Speculative Dataflow Processor*

Bradley C. Kuszmaul† and Dana S. Henry‡

Abstract

A processor with an explicit dataflow instruction-set architecture may be able to achieve performance comparable to a superscalar RISC processor, even on serial code. To achieve this, the dataflow processor must support speculative operation, especially speculative branches, and a pipeline with bypassing for serial code. This paper outlines a set of mechanisms to implement speculative operation with a bypassing pipeline, in a paper design called the Speculative Dataflow Processor (SDP).

The SDP uses several novel ideas as compared to traditional dataflow processors. Branches are predicted and speculated using a new branch firing rule. Several branch statements are grouped together so that they use a single branch prediction. The scheduling and bypass logic is similar to, but simpler and faster than, the corresponding logic in a superscalar RISC processor.

Speculation introduces some new compiler issues. Additional care must be taken by the compiler to prevent speculative tokens from Iteration $i+1$ from overrunning the nonspeculative tokens from Iteration i of a loop.

1 Introduction

Processors with explicit dataflow instruction-set architectures (for example [PC90, GKW85]) have generally not been as fast as contemporary von Neumann processors. They have performed especially poorly on programs that have little parallelism. One approach to solve this problem is to design processors that are a hybrid of dataflow and traditional RISC processors to obtain the best of both worlds, executing both serial and parallel code efficiently. (See for example P-RISC [NA88] and simultaneous multithreading [TEL95] at the RISC end of the spectrum and EM-5 [SKY91] and the Tera MTA [ACC+95] at the dataflow end.) This paper argues that a "pure" dataflow processor can also compete effectively if two problems are solved: It must have speculative branch execution, and the pipeline must be very efficient for serial code. We present here a paper design of a processor, called the Speculative Dataflow Processor (SDP), that we believe will work reasonably well based on our analysis and also on the intution we have gained from several compiler and processor VLSI projects. We have not yet implemented a compiler, simulator, or a circuit design for SDP, although we are working on the compiler and simulator.

Our goal is to design a dataflow processor that competes effectively with a superscalar microprocessor. This means that we are not interested in high processor utilization, for example. Contrast this approach with, for example, the Tera MTA architecture [ACC+95]), which attempts to achieve high processor utilization and is willing to use a very expensive memory system to achieve it. A processor does not need to achieve high utilization of its ALU or VLSI, since VLSI is cheap. Since the memory system dominates the cost of a high-performance machine, it would suffice to achieve high memory-subsystem utilization.

In addition to branch prediction, a dataflow processor should speculate on load/store conflicts, but there is not space here to discuss that mechanism.

Figure 1 illustrates the mechanisms needed to implement branch speculation in our dataflow architecture. (Here we are describing the state of the machine with tokens drawn on arcs, but as we shall see later, we use an explicit-token store design in which the tokens correspond to entries in an activation frame.) Figure 1(a) shows a C code segment that we compiled to the dataflow graph in Figure 1(b). The graph contains arithmetic operators, such as "+", together with switch operators (which implement branches), identified by diamonds. Switch operators have two inputs: a predicate (shown entering from the left of the diamond) and a datum (shown entering from above the diamond.)

Several switches may share the same predicate. In Figure 1(b), two switches share the same predicate, "<". To help remind the reader that the switches are related we draw the switches with the same shared predicate on the same horizontal row. In our implementation, we will take advantage of their shared predicate to reduce their speculation costs. We distinguish between switches and branches as follows: A switch can route a single token according to a predicate. A branch is the collection of switches that implement a single branch in the original program. That is, a branch is the set of switches sharing the same predicate.

Except for switches, each operator in Figure 1(b) uses the traditional dataflow firing rule—the operator fires once a value, called *token*, arrives at each input. Switch operators may fire twice, however. A switch can fire whenever its data token (vertical input) arrives. If the switch's predicate token (left input) has not yet arrived, the switch may predict the predicate's value and passes the data token to the `T` output or the `F` output accordingly. The switch fires again once the predicate token arrives. If the predicate token's value does not agree with the prediction, the switch initiates branch recovery.

To illustrate how a switch recovers from misprediction, Figure 1(b) and Figure 1(c) show the runtime state immediately before and after a misprediction. We assume that, initially, one input token was inserted along each input arch `x`, `y`, `z`, and `w`. Solid tokens within each graph indicate data that has not yet been consumed. In Figure 1(b), the comparison operators, "<" and ">2", have not yet consumed their input tokens. At the same time, however, all switches have already predicted their outcomes. The predicted outcomes are shown in bold inside each diamond. Based on these predictions, tokens have already propagated all the way to the multiply operation.

To enable recovery from misprediction, we must remem-

*This research was partially supported by NSF CAREER Grant 9702980 (Kuszmaul) and NSF CAREER Grant 9702281 (Henry.)

†Akamai Technologies and Yale University. `http://eecs.yale.edu/~bradley`

‡Yale University. `http://eecs.yale.edu/~dana`

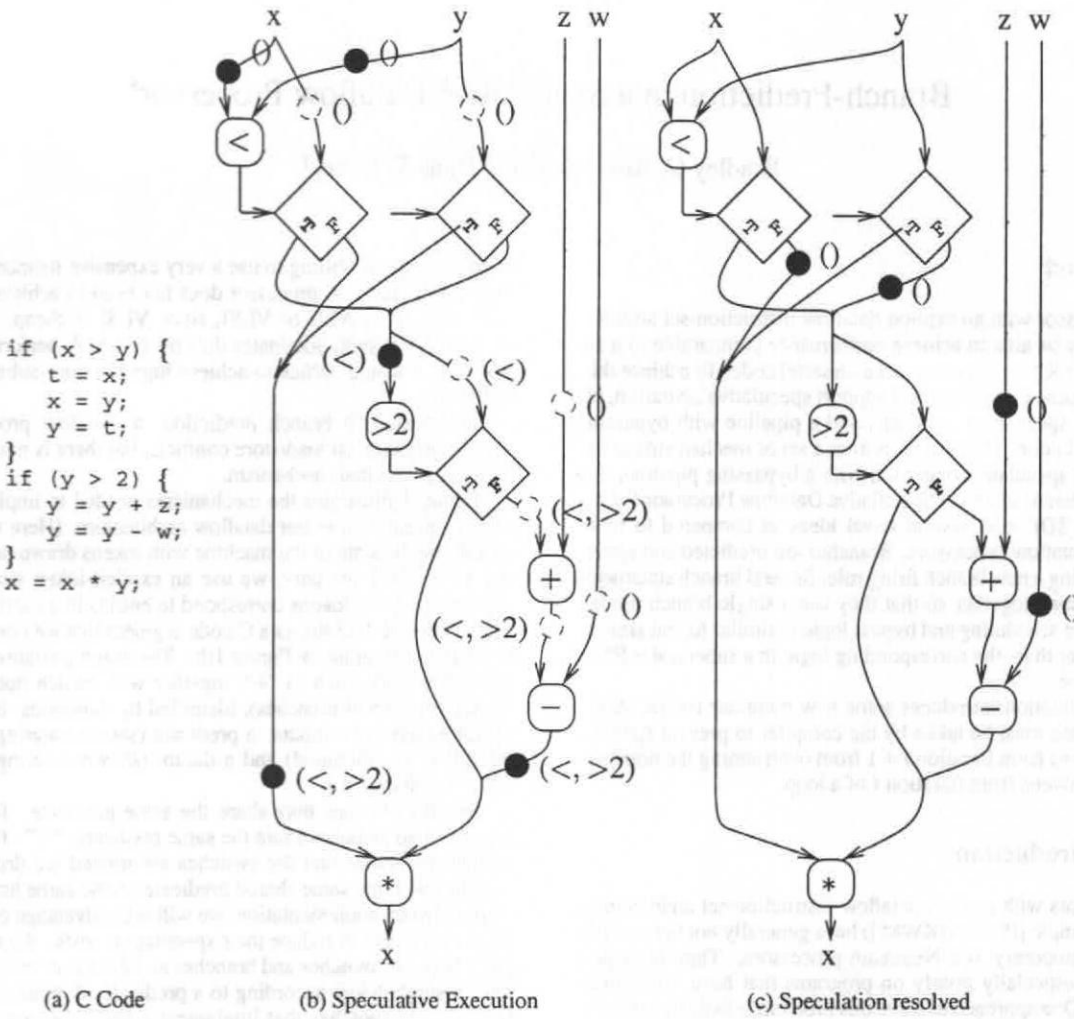


Figure 1: An example of how speculation works. (a) A fragment of C code. (b) The runtime state with the "<" test predicted true, and the ">2" test predicted false. (c) The restored graph after the "<" test resolves to false.

ber some tokens even after they have been consumed. These tokens are illustrated with dashed lines in Figure 1(b). The dashed lines show the input tokens of every operator that has fired speculatively. We keep track of which operators have fired speculatively by marking each token with the list of predicates on which it is speculating. For example, we remember the input tokens of the "+" operator because the operator's left input token is speculating on the outcome of both predicates.

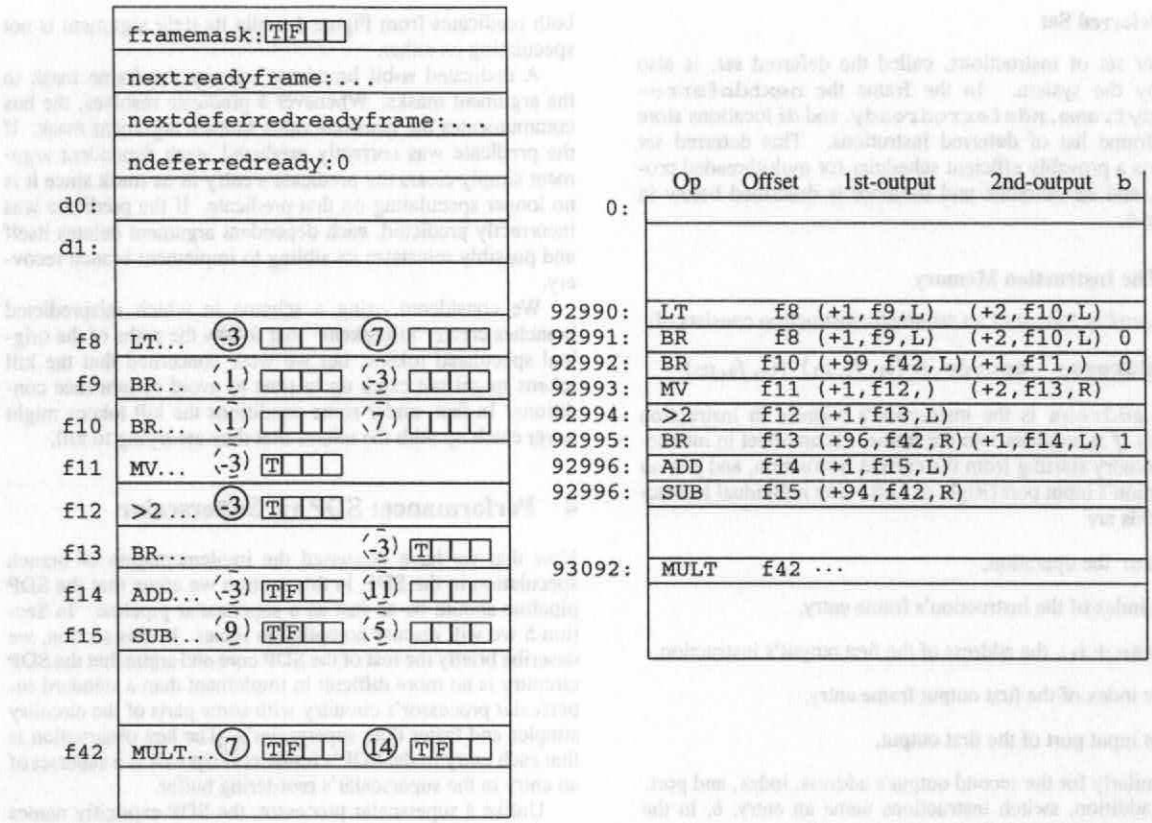
Figure 1(c) illustrates recovery from misprediction. In this example, the "<" operator has resolved to False; the affected switches have fired again and detected a misprediction. As a result, every operator that speculated on the predicate "<" undoes its computation, restoring any input token that it should not have consumed.

The rest of this paper is organized as follows. Section 2 describes the SDP instruction-set architecture. Section 3 sketches how to implement branch speculation in SDP. Section 4 argues that SDP should compare well to a superscalar processor. Section 5 discusses compilation issues raised by speculation. Section 6 shows how to support provably efficient multithreaded scheduling, and Section 7 concludes with a discussion of related and future work.

2 Instruction Architecture

Having outlined the idea behind the mechanism in Section 1, in this section we describe the instruction set architecture (ISA) for the SDP. The rest of the paper will then describe the implementation issues for this ISA.

Except for switches, the SDP processor's instruction set architecture is analogous to the explicit token store architecture pioneered by Monsoon [PC90]. Figure 2 illustrates the architected state using the code segment and execution graph from Figure 1. The state consists of set of frames, such as the one shown in Figure 2(a), and instruction memory The instruction memory holds the static information about the program (the "text" of the program), whereas the frames hold dynamic information for the procedure's outstanding instructions. Each frame corresponds to one procedure invocation or one thread,



(a) One frame (of many.)

(b) Instruction Memory.

Figure 2: Architected State.

typically.

2.1 The Frame

A frame is a contiguous region of memory which is used as the backing store for state that is normally kept in the processor core. When there are many active frames, the processor will need to move some of the frame state out of the core to the memory.

The frame includes

- a `framemask` which is used in branch speculation,
- a collection of frame entries (`f0`, `f1`, *ldots*),
- fields to implement ready-to-execute instructions, and
- fields to implement another set of deferred instructions.

Each of these are described below.

The Frame Mask

The framemask keeps track of all of the outstanding unresolved branches for a frame. Switches that use the same predicate share one branch-mask entry. In Figure 2, the first entry in the mask lists the prediction made by the "<" predictor in Figure 1; the second entry in the mask lists the prediction made by the "> 2" predictor in Figure 1.

The frame mask is part of the architected state because the compiler must manage the allocation of the frame mask entries.

The Frame Entries

For each instruction that has a token on one of its inputs, the frame keeps track of the instructions arguments and state. In Figure 2(a), arguments that have not yet been consumed appear inside a shaded token, and arguments that have been speculatively consumed appear inside a dashed token. There is an additional *argument mask* stored with each argument token which is part of our impleentation and will be described in the following section.

The Ready-to-Execute Set

Each frame keeps in its state the set of all instructions that are ready fire. More than one frame may have instructions which are ready to execute, however. The frame provides storage, called `nextreadyframe`, to build a linked list of all such frames.

The Deferred Set

Another set of instructions, called the deferred set, is also kept by the system. In the frame the `nextdeferredreadyframe`, `ndeferredready`, and `di` locations store a per-frame list of deferred instrutions. This deferred set supports a provably efficient scheduler for multithreaded programs, and its rationale and behavior is described below in Section 6.

2.2 The Instruction Memory

The assembly format of an arithmetic instruction consists of:

`address:` `opcode` f (i_1, f_1, p_1) (i_2, f_2, p_2)

where `address` is the instruction's address in instruction memory, f is an index into the frame, i is an offset in instruction memory starting from the current instruction, and p is an instruction's input port (Right or Left.) The individual instruction fields are

`opcode`: the operation,

f: the index of the instruction's frame entry,

`address` $+ i_1$: the address of the first output's instruction,

f_1: the index of the first output frame entry,

p_1: the input port of the first output,

and similarly for the second output's address, index, and port.

In addition, switch instructions name an entry, b, in the branch mask that holds their prediction while they speculate:

`address:` BR f (i_1, f_1, p_1) (i_2, f_2, p_2) b.

Branches that share the same predictor share the same mask entry. In addition, static switches that never dynamically coexist within the frame may also name the same mask entry.

3 Implementing Branch Speculation

Section 2 described the SDP instruction set architecture (ISA), which is the programmer-visible behavior of the machine. This section sketches an implementation, and Section 4 argues that the implementation should be at least as fast as a superscalar pipeline.

To implement branch speculation, we added a n-bit frame mask register to the frame. The register uses 2-bits to encode the state of each entry in the frame mask. There are three states, which we notate as

☐: the entry is not in use,
T: the entry's predicate is predicted taken, and
F: the entry's predicate is predicted not-taken.

Each entry's value is set the first time a switch fires speculating on the entry's predicate. Each entry's value is cleared whenever a switch fires for the second time, confirming or refuting that prediction.

We also maintain a n-bit *argument mask* with each argument field in the frame. Each argument mask lists a subset of the frame mask on which the corresponding argument is speculating. Figure 2 shows the setting of all the argument masks for the program state described in Figure 1(b). For example, the ADD instruction's left argument is speculating on both predicates from Figure 1 while its right argument is not speculating on either.

A dedicated n-bit broadcast bus ties the frame mask to the argument masks. Whenever a predicate resolves, the bus communicates the resolved value to each argument mask. If the predicate was correctly predicted, each dependent argument simply clears the predicate's entry in its mask since it is no longer speculating on that predicate. If the predicate was incorrectly predicted, each dependent argument deletes itself and possibly reinstates its sibling to implement branch recovery.

We considered using a scheme in which mispredicted branches create "kill tokens" that follow the paths of the original speculated tokens, but we were concerned that the kill tokens might not catch up in time to avoid certain race conditions. In fact, under some conditions the kill tokens might never catch up with the tokens that they are trying to kill.

4 Performance: SDP vs. Superscalar

Now that we have discussed the implementation of branch speculation in the SDP, in this section we argue that the SDP pipeline should be as fast as a superscalar pipeline. In Section 5 we will discuss compilation issues. In this section, we describe briefly the rest of the SDP core and argue that the SDP circuitry is no more difficult to implement than a standard superscalar processor's circuitry with some parts of the circuitry simpler and faster than superscalar's. The key observation is that each entry in the SDP's frame corresponds is a superset of an entry in the superscalar's reordering buffer.

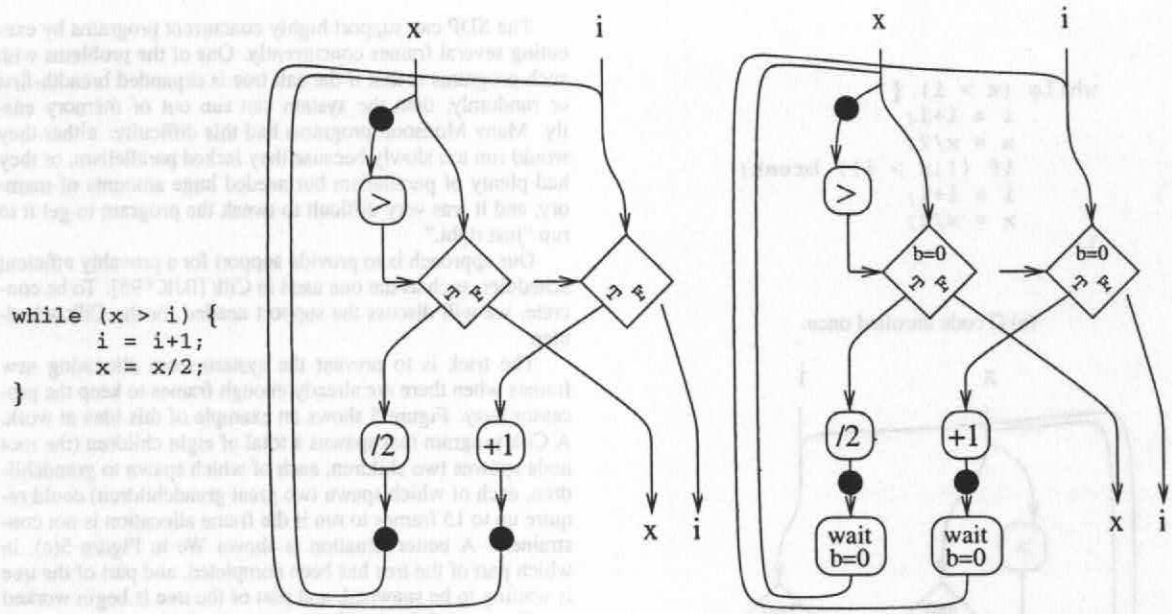
Unlike a superscalar processor, the SDP explicitly names the children of each instruction in the frame. As a result, the SDP does not have to broadcast each result to the entire frame. Instead, it can directly write each result into each child's frame entry. This optimization replaces area-intensive associative writes into the superscalar reordering buffer with faster and smaller direct writes into the SDP's frame.

However, explicitly naming each instruction's children also has its costs. If there are many destinations for an instruction, and the instruction has limited fan-out, then extra fanout instructions will be needed. In our architecture, we used instructions with fanout of two, but it may make sense to use instructions with a fanout of three or four to reduce the need for extra fanout instructions.

Also appearing in a frame entry but not in a reordering buffer entry is the argument mask described in the previous section. This mask supports selective recovery from misprediction in the SDP. Unlike a traditional superscalar processor, the SDP can back out of exactly those instructions that depend on a mispredicted branch. In contrast, a superscalar undoes *all* instructions following a mispredicted branch, whether they actually depend on the mispredicted branch or not.

The SDP does not need renaming logic since the compiler explicitly manages the reuse of frame entries. Explicitly managing storage reuse puts pressure on the size of the frame, however. It remains to be seen how large a frame is needed to achieve good performance.

The critical-path length of a program may be longer using SDP than using a serial instruction set, because in a superscalar, correctly predicted branches do not appear in the critical path of the program at all. In the SDP, even correctly predicted branches add the cost of the switch instruction to the critical path. The number of instructions can be greater in SDP than in superscalar processors as well, since a single branch in



(a) C code. (b) Compiled for traditional firing rule. (c) Compiled for speculation.

Figure 3: Loop barrier example. This code computes `i+=lg(x);x=1`.

a superscalar may correspond to many switch instructions in SDP.

In other aspects, the SDP is essentially identical to a superscalar reordering buffer entry, and executes in the same way. For example, the same bypassing techniques used by the superscalar processor can be used in the SDP.

5 Compiler Support

This section discusses compiler issues for SDP, which are important even for serial programs. The next section will discuss the hardware support needed for highly concurrent multithreaded programs. In addition to the hardware issues described in Sections 3 and 4, the compiler needs to take extra care when compiling for a speculative dataflow processor.

First, as we saw in Figure 1, the compiler must group together switches that use the same predicate. Such grouping reduces the number of outstanding predictions to the number of unresolved predicates rather than unresolved switches. In addition, the compiler must understand our new firing rule for switches. Without the compiler's cooperation, the speculative firing of a switch could yield multiple tokens along one arch in violation of our explicit-token-store dataflow architecture.

Figure 3 illustrates the effect of the new switch firing rule. It shows a simple serial C code loop compiled with the traditional single-firing rule (Figure 3(b)) and with our new speculative firing rule for switches (Figure 3(c)). In Figure 3(b) the compiler has used the traditional rule, assuming that each switch will fire only once, after both inputs have arrived. Under this assumption, all initial inputs to the loop will be consumed before the next iteration's inputs are generated. If the switches were to fire speculatively instead, without waiting for their predicate tokens, the program would fail. As Figure 3(b) illustrates, the next speculative value of `x` could reach the predicate operator "`>`" before the first value has been consumed.

To avoid multiple tokens along the input arch to the predicate operator, the compiler must introduce explicit *speculation barrier* instructions as in Figure 3(c). We have shown the branches with branch masks ("`b=0`"), and the speculatiion barrier is denoted by "`wait b=0`." A speculation barrier will not fire until the branch mask mentioned has resolved.

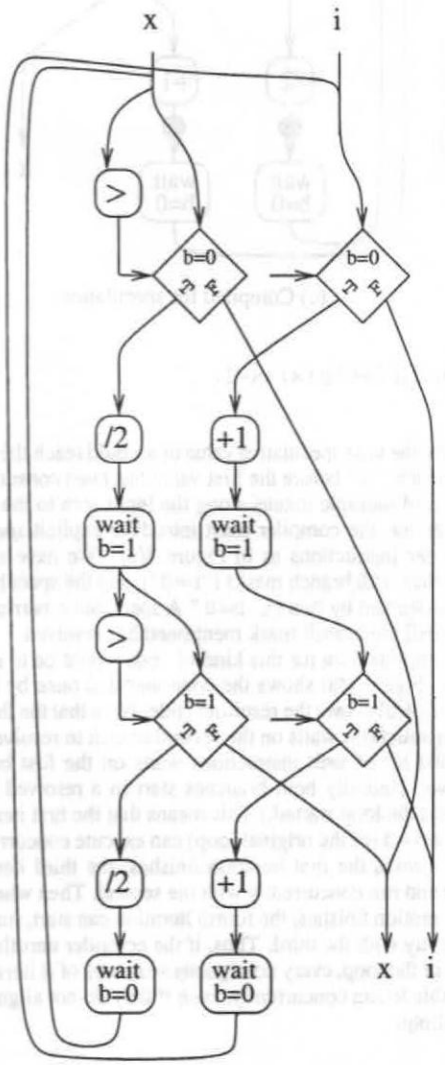
One optimization for this kind of code would be to unroll the loop. Figure 4(a) shows the code unrolled once by hand, and Figure 4(b) shows the resulting code. Note that the first set of wait instructions waits on the second branch to resolve, and the second set of wait instructions waits on the first branch to resolve. (Initially both branches start in a resolved state, which gets the loop started.) This means that the first iteration and the second (of the original loop) can execute concurrently. And then when the first iteration finishes, the third iteration can start and run concurrently with the second. Then when the second iteration finishes, the fourth iteration can start, running concurrently with the third. Thus, if the compiler unrolls k iterations of the loop, every contiguous sequence of k iterations will be able to run concurrently, even if they do not align with the unrolling.

6 Support for Parallel Programs

So far we have explained how to run serial programs on a speculative dataflow processor, taking advantage of the parallelism within one subroutine of an otherwise serial program. This section outlines how a dataflow processor can be designed to support provably efficient scheduling of highly concurrent multithreaded programs. Section 7 will then discuss the related and future work.

```
while (x > i) {
      i = i+1;
      x = x/2;
      if (!(x > i)) break;
      i = i+1;
      x = x/2;
}
```

(a) C code unrolled once.



(b) Compiled for speculation

Figure 4: An unrolled version of the code from Figure 3. When we unroll the loop we can use split-phase speculation barriers so that the two iterations of the loop can run concurrently..

The SDP can support highly concurrent programs by executing several frames concurrently. One of the problems with such programs is that if the call tree is expanded breadth-first or randomly, then the system can run out of memory easily. Many Monsoon programs had this difficulty: either they would run too slowly because they lacked parallelism, or they had plenty of parallelism but needed huge amounts of memory, and it was very difficult to tweak the program to get it to run "just right."

Our approach is to provide support for a provably efficient scheduler, such as the one used in Cilk [BJK+95]. To be concrete, we will discuss the support needed for the Cilk scheduler.

The trick is to prevent the system from allocating new frames when there are already enough frames to keep the processor busy. Figure 5 shows an example of this idea at work. A Cilk program that spawns a total of eight children (the root node spawns two children, each of which spawn to grandchildren, each of which spawn two great grandchildren) could require up to 15 frames to run if the frame allocation is not constrained. A better situation is shown We in Figure 5(c), in which part of the tree has been completed, and part of the tree is waiting to be spawned, and part of the tree is begin worked on. The part of the tree that is being worked on has at most 2 active leaves in this case, and the part of the tree that is waiting to be spawned is deferred.

To be more specific about the allocation rule, we provide here a brief review of the Cilk system, from the perspective of a multithreaded processor architect. In Cilk, the computation is structured into a call tree, in which a vertex corresponds to a subroutine instance, and in which certain subtrees can execute in parallel. To execute several subtrees in parallel, the programmer writes a collection of "spawn" procedure calls, and then a "sync" operation that waits for all the children to complete. An ordinary procedure call is simply a spawn of a single subtree, followed by a sync.

Cilk achieves optimal time and space bounds simultaneously. The time bounds are expressed using the time to execute on one processor, T_1, and the critical path length of the program, T_∞, which is the time it would take to run on an infinite number of processors. On P processors, Cilk can run a program in time that is $T_1/P + O(T_\infty)$. If the space bound on one processor is S_1, then the space bound on P processors is $P \cdot S_1$. These bounds are optimal under certain assumptions.

Cilk programs must be *strict* in order for the scheduler to achieve these bounds. Informally, a strict program is one in which, once a subtree starts, it is able to finish without waiting for other subtrees to finish.

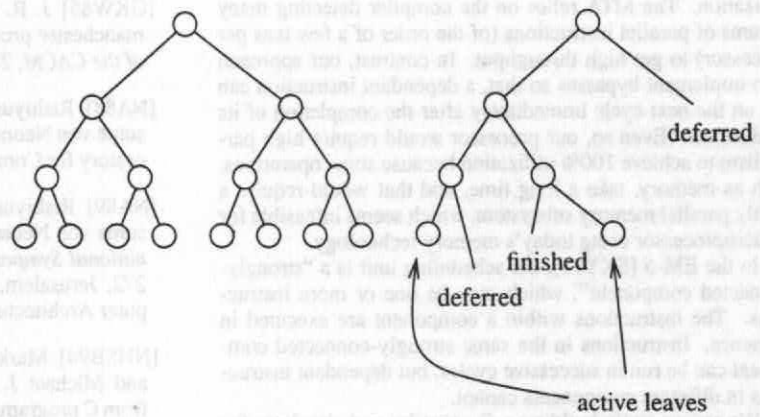
Cilk achieves these time and space bounds by guaranteeing that at most P "leaves" of the call tree exist at any given time. Another way to say this is that in the tree, at most P forks are expanded at any gien time.

When the system has fewer than P leaves running, every spawn actually starts up a new subtree in parallel. When the system has P leaves active, then no new subtrees are spawned. That is, the system runs in a serial, depth-first, order on each of the extant branches of the tree. That means that only one spawned child of a frame is actually started at a time. The others wait until the first one completes, and then another spawned child can run.

For the discussion here, we are interested in supporting Cilk on a single speculative dataflow processor that may have a limited number of frames. So P, instead of referring to the number of processors, refers to the number of leaf frames we can support in the processor core. The speedup bounds work

```
int recurse (int n) {
  if (n==0) serially_work();
  else {
    spawn recurse(n-1);
    spawn recurse(n-1);
    sync;
  }
}
```

(a) The program.



(b) The entire call tree for `recurse(3);`

(c) Only two leaves are allowed to exist at a time.

Figure 5: In Cilk, a limited number of forks in the dynamic call tree are allowed. Each node of the call tree shown in (b) represents one invocation of the procedure named `recurse`. The whole call tree includes eight leaf nodes, but if on a two-processor system we only want there to be two leaves enabled. Parts of the tree have already finished executing, and so their frame memory is deallocated, and part of the tree is waiting to execute, but we do not actually allocate memory until one of the leaves finishes.

out differently as well, since there are not actually P ALUs and other computational units, but the system still has a sound theoretical basis.

Thus, to make this dataflow-oriented Cilk work requires that the runtime system be able to distinguish between two cases when spawning a child. The "serial case" is when a child is being spawned and there are no other children currently in existance. The "parallel case" is when there is already a child running for a particular frame, and we must be careful not to start another child unless there are idle processing resources.

Most of the support for Cilk-scheduling within the SDP can be implemented in software, with a very small amount of hardware support. The system must maintain a separate "deferred" execution queue for the instructions that allocate new frames. Instructions are executed via the regular window-scheduling mechanism whenever possible. The rule for when a deferred frame allocation instruction can run is more complex, however.

The idea is that frame allocation instructions in the deferred queue should not be run if there are too many spawned children in the system. To make this work the processor keeps a global count of how many leaf children are running. The processor might be designed to allow, say, 16 concurrent leaves in the call tree to be executing. If the global count is less than 16, then the processor executes a frame allocation instruction out of the deferred queue (putting the resulting tokens into the regular execution pipeline) and increments the global counter. If the global count is greater than 16, then the processor does not execute instructions from the deferred queue.

Here is how the system can compute how many active leaves exist in the call tree. In software, a Cilk program sets up a counter in the activation frame to keep track of how many children are running (the "active-child count.") Initially the counter is set to zero. When spawning a child subroutine, if the active-child count is zero, the spawn is treated like a serial call (the frae allocation instruction is executed normally), and the local counter is incremented. If the active-child count is positive, then the token that starts the frame allocation is placed into the deferred queue, and the count is not incremented.

When a forked child completes, the system must decrement the parent frame's counter, and if that goes from two to one, it must decrement the global leaf counter, which will then allow some deferred instruction (if there is one) to run, allocating a new frame.

The effect of all this is to implement a provably efficient scheduler by providing the mechanisms needed to prove the Cilk results for SDP.

We could have taken the decision to perform Cilk-style scheduling in software, but we wanted to be able to write Cilk-style programs in which the spawn and procedure call instruction-sequence are the same. We wanted the "serial case" to run as fast as possible, and so we provide hardware support for the Cilk-style scheduling.

7 Related and Future Work

The biggest difference between our machine and previous pure dataflow machines, such as Monsoon [PC90] and the Manchester dataflow machine [GKW85] is that we make extensive use of speculation to achieve high performance on single-threaded code. In contrast Monsoon could only use one eighth of a single processor's cycles on single threaded code. The speculation we propose is possible because of advances in VLSI technology since the previous generation of pure dataflow machines.

Among the hybrids, the two machines that look the most like our proposed machine are the Tera (now known as Cray) MTA, and the EM-5. Beyond the fact that our machine is a pure dataflow machine, and makes extensive use of speculation their are some other interesting differences.

The Tera MTA [ACC+95] allows very restricted out-of-order execution within a single thread (called a stream). Each instruction specifies how many successive instructions can be issued before this instruction completes, and this is limited by 7. So in effect, a single stream has a window size of 8 or less. The pipeline depth for the MTA is about 70 clock ticks, and so at least 9 streams are required to achieve 100% processor

utilization. The MTA relies on the compiler detecting many streams of parallel instructions (of the order of a few tens per processor) to get high throughput. In contrast, our approach is to implement bypasses so that, a dependant instruction can run on the next cycle immediately after the completion of its predecessor. Even so, our processor would require high parallelism to achieve 100% utilization because some operations, such as memory, take a long time, and that would require a highly parallel memory subsystem, which seems infeasible for a microprocessor using today's memory technology.

In the EM-5 [SKY91], the scheduling unit is a "strongly-connected component", which may be one or more instructions. The instructions within a component are executed in sequence. Instructions in the same strongly-connected component can be run in successive cycles, but dependent instructions in different components cannot.

We are currently building a C compiler and simulator for the SDP. Several other researchers have shown that it is possible to systematically compile serial programs for dataflow machines [BP89, NHSB94, WA95]. Given that it is possible to compile serial programs, our compiler work is directed to supporting the thesis that a pure dataflow processor can compete with a von Neumann processor. We hope to soon have results about the effectiveness of our branch prediction scheme and of our fetch prediction scheme, and of the SDP in general.

As a possible improvement to the ideas of the SDP, we are considering a dataflow processor with a very different approach to managing data and tokens. Instead of using tokens that carry data, we are considering a dataflow processor that uses explicit registers, and in which the tokens carry only synchronization information. This would reduce the number of switches to be comparable to a superscalar processor. Instead of one switch for every data value, it would be more like one switch per branch. That would allow us to remove the branch masks from the ISA because the branch masks can be dynamically assigned to the instructions. This approach would also reduce the number of instructions in the code.

Acknowledgments

Yale graduate student Rahul Sami helped with the analysis of the related work, and has been examining the problem of compiling C for a dataflow machine.

References

[ACC+95] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera Computer System. `ftp://www.net-serve.com/tera/arch.ps.gz`, 1995.

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '95)*, pages 207–216, Santa Barbara, California, July 1995. (`http://theory.lcs.mit.edu/pub/cilk/PPoPP95.ps.z`).

[BP89] Micah Beck and Keshav Pingali. From control flow to dataflow. Technical Report TR 89-1050, Department of Computer Science, Cornell University, Ithaca, NY 14853-7501, October 1989.

[GKW85] J. R. Gurd, C. C. Kirkham, and I. Watson. The manchester prototype dataflow computer. *Communications of the CACM*, 28(1):34–52, January 1985.

[NA88] Rishiyur S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? CSG Memo 292, MIT Laboratory for Computer Science, November 1988. See [NA89].

[NA89] Rishiyur S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *The 16th Annual International Symposium on Computer Architecture*, pages 262–272, Jerusalem, Israel, May 1989. ACM SIGARCH Computer Architecture News, Volume 17, Number 3, June 1989.

[NHSB94] Mark H. Nodine, James E. Hicks, Cotton Seed, and Michael J. Beckerle. Generating parallelism profiles from C programs. Technical Report MCRC-TR-43, Motorola Cambridge Research Center, One Kendall Square, Building 200; Cambridge, MA 02139, September 1994. (Available as `http://csg-www.lcs.mit.edu:8001/mcrctr/tr43/ppg.html`).

[PC90] Gregory M. Papadopoulos and David E. Culler. Monsoon: An explicit token store architecture. In *Proc. 17th. Intl. Symp. on Computer Architecture, Seattle, WA*, May 1990.

[SKY91] Shuichi Sakai, Yuetsu Kodama, and Yoshinori Yamaguchi. Architectural design of a parallel supercomputer em-5. In *Proc. Japan Soc. Parallel Proc., Kobe Japan*, pages 149–156, May 14–16 1991.

[TEL95] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA '95)*, pages 392–403, Santa Margherita Ligure, Italy, 22–24 June 1995. *Computer Architecture News*, 23(2), May 1994.

[WA95] S. F. Wail and D. Abramson. Can dataflow machines be programmed with an imperative language. In G. Gao, L. Bic, and J.-L. Gaudiot, editors, *Advanced Topics in Dataflow Computing and Multithreading*, pages 229–265. IEEE Computer Society Press, 1995.

A Study of Compiler-Directed Multithreading for Embedded Applications

Anasua Bhowmik
Computer Sciences Department
University of Maryland
College Park, MD 20742
anasua@cs.umd.edu

Manoj Franklin
ECE Department and UMIACS
University of Maryland
College Park, MD 20742
manoj@eng.umd.edu

Quang Trinh
ECE Department
University of Maryland
College Park, MD 20742
trinh@eng.umd.edu

ABSTRACT

Growing demand for high performance in embedded systems is creating new opportunities to leverage techniques such as pipelining and instruction-level parallel processing, which were originally developed for general-purpose processors. In this paper, we investigate the applicability of compiler-directed multithreading in speeding up embedded applications. In particular, we take programs from the Powerstone benchmark suite—a collection of programs from the embedded applications area—and use our compiler-directed multithreading framework to partition them into multiple threads. While performing the partitioning, the compiler not only considers data dependence information, but also considers control independence information and profile-based information on the most likely control flow paths. Our compiler framework is implemented on the SUIF-MachSUIF platform. The average code expansion due to the introduction of thread information is only 0.82%, but, the performance potential is quite substantial. The effect of different criteria on our thread partitioning technique is evaluated using a trace-driven, multithreaded processor simulator. Our measurements indicate that future embedded processors can speed up the execution of sequential programs with low degrees of multithreading.

Keywords

Branch prediction, control dependence, Powerstone benchmarks, profiling, speculative execution, thread-level parallelism (TLP)

1. INTRODUCTION

Improving the execution speed of embedded applications is becoming an important problem. Any serious attempts at solving this problem should carefully consider the trends in technology. In spite of the severe power consumption requirements, the number of transistors in embedded processors has been rising, primarily due to advances in device technology. This ongoing explosion in device technology is complemented by a similar increase in clock speed. This situation is complicated by a constraint that is germane to embedded processors—low power consumption. Designers of embedded processors have been utilizing the increasing transistor budget to incorporate special features that speed up some aspects of embedded computing. But today embedded processors are being used for a wide variety of applications, and embedded processor designers have begun to include features that are traditionally found in general-purpose processors [16].

Recent studies on multithreading confirm that there is significant performance potential in executing a small number of threads in parallel. Furthermore, the use of multiple hardware sequencers or processing elements (to fetch and execute multiple threads)—besides making judicious use of the available transistor budget increase—fits very nicely with the goal of decentralization, which is very important to deal with on-chip wire delays. Using the increased device count to build additional processing elements (PEs) is indeed a very credible option [2] [5] [13] [14]. The primary means of increasing processor performance, besides increasing the clock speed and reducing the memory latency, has always been the exploitation of the inherent parallelism present in programs, with the use of a combination of software and hardware techniques. Although the majority of previous research in embedded processors focused on a single thread of execution, a more effective increase of parallelism can be achieved from the execution of multiple threads belonging to the same program[1].

This paper investigates the potential of software (compiler-based) techniques to partition sequential embedded programs into multiple threads that the hardware can execute in parallel. Because the compiler has an overall view of the program, it can find the control independent points in the program and partition the sequential program into multiple threads. It can also determine data dependences between distant code. We use both of these features, along with profile-based data on likely control flow paths, to partition sequential programs into multiple threads. Thus, our compiler based thread partitioning algorithm takes into account both control and data independence to do effective thread partitioning. From the hardware side, data value prediction is incorporated to reduce the effect of inter-thread data dependences.

Our studies with embedded applications have led to the following observations:

- The performance potential of single-threaded processors is fairly limited.

[1]The term "thread" has different meanings in different contexts; our notion of threads is finer than the coarse-grain OS-level threads, and comprise of tens to hundreds of instructions.

- Compiler-directed speculative multithreading, along with data value speculation, has good potential to speed up embedded applications
- Some embedded programs benefit from the use of non-loop threads

The rest of this paper is organized as follows. Section 2 provides background information on multi-threading for embedded systems applications. Section 3 presents an overview of our multi-threading compiler framework. Section 4 presents an experimental evaluation of the compiler-generated threads for the Powerstone benchmarks. Section 5 presents a summary and the major conclusions of this paper.

Benchmark	Description
auto	Automobile control application
bffo	
bilv	Shift, AND, OR operation
blit	Graphics application
compress	A Unix utility
des	Data Encryption standard
fir_int	
g3fax	Group three fax decode (Single level image decompression)
ucbqsort	U.C.B. Quicksort

Table 1: Powerstone benchmark suite

2. MULTITHREADING FOR EMBEDDED APPLICATIONS

2.1 Constraints for Embedded Processors

Embedded processors currently form an important sector of the processor market. They are particularly used in many applications in the communications and mobile computing area. Although the basic tenets of computing in the embedded systems world are the same as those in the general-purpose computing world, there are some additional constraints to be considered while designing embedded processors. These constraints concern primarily with power dissipation, code size, and die size. Many embedded processors are used in applications such as cellular phones where the power supply is derived from a battery. For such applications, it is very important that the power consumption of the processor is as low as possible. Many embedded systems are also constrained by memory size and die size limitations. Limited memory size implies that the code size should be as small as possible. In spite of these special constraints for embedded systems, the demands on the processing power for embedded applications has been steadily rising.

2.2 Parallelism in Powerstone Benchmark Programs

It is worthwhile to characterize embedded applications. In particular, we like to know how much parallelism exists, what kind of branch prediction accuracies we can obtain, etc. To that end, we measure the available parallelism (under different machine models) present in the Powerstone benchmarks, a collection of embedded application programs including automobile control, signal processing, graphics and fax applications. A description of the benchmarks is given in Table 1. These portable and embedded benchmarks are used to make design trade-offs in the architecture and the compiler of the Motorola low power M-CORE processor [16]. For this study, we use a software tool called TAPE (Tool for Available Parallelism Estimation) [3]. TAPE performs trace-based simulation, and performs a parallelism *limit study* by constructing a dynamic dependence graph (DDG) based on the different kinds of dependences present among the instructions of the trace. TAPE allows different models for handling control dependences: realistic branch prediction, realistic branch prediction augmented with exploitation of control independences, and perfect branch prediction.

Let us take a quick look at the amount of parallelism available in the Powerstone benchmarks under these different control flow models. Table 2 presents the available parallelism obtained for 3 abstract machine models (given in 3 columns): (i) an execution model in which control speculation is employed within a window of 32 instructions, but control independence is not utilized, (ii) an execution model in which control speculation is employed, and control independence is utilized whenever a branch is mispredicted within a window of 256 instructions, and (iii) an execution model that utilizes perfect branch prediction and a window size of 256 instructions. The first case indicates a limit of what can be achieved by ILP (instruction-level parallelism) techniques, and the second indicates the potential of pursuing multiple threads. These measurements were done with the Alpha instruction set architecture.

Table 2 also presents the branch prediction accuracies obtained for the benchmarks. Whereas many of the benchmarks obtain very high prediction accuracies, in the range 96%-99.9%, there are a few that obtain substantially low prediction accuracies—`compress` (91.0%), `ucbqsort` (81.06%), and `des` (75.42%). The parallelism obtained by branch prediction alone is naturally low for these three benchmarks (around 5). The column that is of particular interest to us is the penultimate one, because it shows the potential of multithreading to improve performance. On looking at this column, we can see that except for `auto`, `des`, and `g3fax`, the others can obtain reasonable performance enhancements by small-scale multithreading.

This characterization also indicates that for most of the programs in the Powerstone benchmark suite, instruction-level parallelism (ILP) techniques can capture only a limited amount of parallelism.

2.3 Speculative Multi-threading for Embedded Processors

Many of the embedded applications are non-numeric in nature. In particular, in such applications memory addresses are difficult (if not impossible) to statically predict—in part because they often depend on run-time inputs and behavior—that makes it extremely difficult for the compiler to statically prove whether or not potential threads are independent. To deal with these difficulties, the speculative multithreading (SpMT) model has been found to be more effective [8] [15]. This model is particularly important to deal with the complex control flow present in typical non-numeric programs. In this model, threads are extracted from sequential code and run in parallel, without violating the sequential program semantics. This means that inter-thread com-

Table 2: Available Parallelism with Different Control Flow Models

Benchmark	Branch prediction accuracy	Available parallelism with		
		No utilization of control independence	Utilization of control independence	Perfect branch prediction
auto	99.86%	6.67	6.77	6.77
bffo	99.23%	10.00	20.59	20.60
bilv	97.14%	12.16	15.89	15.89
blit	99.90%	9.99	10.22	10.22
compress	91.00%	5.44	10.25	14.93
des	75.42%	4.44	8.52	9.27
fir_int	97.09%	10.18	19.45	19.56
g3fax	96.09%	5.70	7.51	7.84
ucbqsort	81.06%	5.65	10.78	15.73

munication between any two threads (if any) is strictly in one direction, as dictated by the sequential thread ordering. Thus, no explicit synchronization operations are necessary, as the sequential semantics of the threads guarantee proper synchronization. Program correctness will not be violated if at run time there is a true data dependence between two threads. The purpose of identifying threads in such a model is to indicate that those threads are good candidates for parallel execution.

3. COMPILER BASED THREAD PARTITIONING

In this section we provide a brief description of our compiler framework for thread partitioning. A detailed description is beyond the scope of this paper; the objective of this paper is to study the effectiveness of multithreading for embedded applications.

3.1 Multi-threaded Architectural Model

The multi-threaded architectural model assumes that the program has been partitioned into a collection of threads. Each thread can spawn any arbitrary number of threads. A particular thread can also be spawned from different places. Threads can be spawned speculatively if required; i.e., a thread can be spawned before knowing for sure that control flow will reach that thread. If it is found that the control speculation was wrong, then the speculative thread is squashed from its PE. But other threads spawned by this speculative thread will be aborted, only if those threads are also control dependent on the same branch. If they are control independent of that branch they can continue execution.

In its general form, this multi-threaded processor hardware consists of a number of processing elements (PEs). Each PE has its own program counter, fetch unit, decode unit, and execution unit, so as to fetch and execute instructions from the thread currently assigned to it. The PEs are connected together by an interconnection network.

3.2 Compiler Framework

In this subsection we briefly describe our compiler framework for thread partitioning. The layout of our overall system is shown in Figure 1.

While partitioning the program into threads, the compiler has to consider three mutually independent factors—*data dependence*, *control dependence*, and *thread size*—together, to decide a good partitioning. Partitioning programs into threads for non strict languages (like C) such that total execution time is minimized, is an NP-Complete problem. So we formulate some metrics and use them to find a good solution of the partitioning problem.

In the following subsections we discuss how the compiler takes care of data dependence, control dependence, and the thread size. The compiler does the program analysis and partitioning on a high level intermediate representation. The high level representation retains all the source level pointer and type information, and hence it is possible to take into account the dependences due to pointer aliasing and array references. Hence the compiler is able to extract parallelism even from pointer intensive programs. We assume that the multi-threaded architecture can take care of the anti- and output- register dependences with dynamic register renaming. We have used the profiling information to find out the most likely path, that the control will take and this information is used by the compiler to specify threads that are to be spawned speculatively.

3.3 Program Profiling

We have used a separate compiler pass to instrument the source code and to gather the profiling information. In the profiling pass, we find out for every basic block, which basic block is most likely to be visited next. The compiler uses this to find out the most likely path and also to estimate the number of instructions that would be executed between two basic blocks. Furthermore, we find out the number of loop iterations using the profiling information. The estimate on the number of loop iterations helps us to decide whether to execute the loop iterations in parallel even in presence of available parallelism. We will discuss this in details in the following subsections.

3.4 Data Dependence

In our framework we formulated a metric called *data dependence count* to partition the programs such that the data dependence between threads are minimized.

Our thread partitioning algorithm works in multiple passes. In the first pass, the compiler builds the CFG and also finds out the data dependence information. It does the traditional data flow analysis and calculates the *read/write* sets [1] for every instruction. We have implemented an intraprocedural pointer analysis to have an improved data dependence information. The pointer analysis helps us in getting more

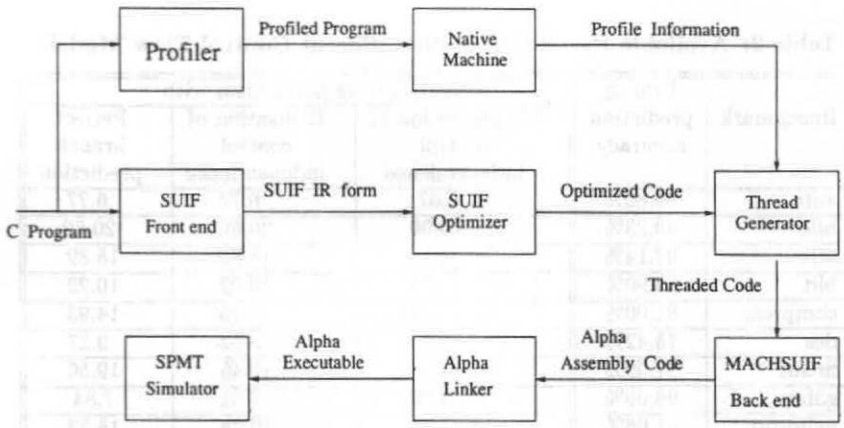


Figure 1: The Layout of the Compiler and Simulator Framework

precise read/write sets. After calculating the *read/write* sets for every instruction, data flow analysis is performed and for every variable in the read set of an instruction, the set of reaching definitions [1] are determined.

The *data dependence count* (DDC) is the weighted count of the number of data dependence arcs coming into a basic block from other blocks as shown in Figure 2. This models the extent of data dependences this block has on other blocks. If the dependence count is small, then this block is more or less data independent from other blocks and it may be beneficial to begin a thread at the beginning of that basic block. While counting the data dependence arcs, the compiler gives more weights to the arcs coming from blocks that belong to the threads closer to the block under consideration. The dependences from distant threads are likely to be resolved earlier and hence the current thread is less likely to wait for the data generated in that thread. Moreover, we give less weightage to the data dependence arcs coming from the less likely paths. The advantage of using this metric are twofold. First of all, it is much simple to compute. Also we found it more accurate than other sequential execution based modeling in the presence of out-of-order execution inside each thread.

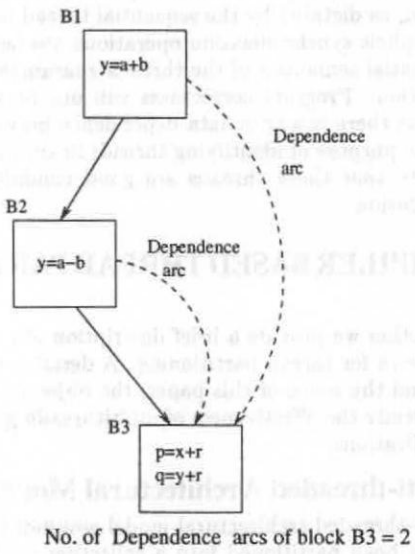


Figure 2: The Data Dependence Arcs

3.5 Program Partitioning

This subsection describes the partitioning algorithm. The compiler partitions the CFG into multiple threads, and also specifies the points in the program from which a particular thread can be initiated. In the partitioning algorithm, I have used the basic blocks as the granularity of partitioning, i.e., either all the instructions inside a basic block are included in a thread or none of them are included. In other words, we do not split a basic block across multiple threads. From every basic block, A, the compiler looks ahead until the basic block B, which is control independent of A and decides which future threads could be initiated from A. Also, at this point the compiler decides which basic blocks in the path between A and B can be included in the current thread, i.e. the thread containing A. To maintain load balancing between the threads, it uses a lower limit and an upper limit for the number of instructions that can be executed in one thread. It also selects speculative threads based upon the profiling data. It selects the most likely path that the program will take for going from basic block A to its next control independent point B. The compiler partitions the program such that the execution in the most likely path be optimized. The thread will continue execution in the speculated path and if it finds the speculation to be incorrect at later point, it will take the correct path. However, there is no need to abort the threads that are spawned at the control independent point of this thread.

Several cases may arise when we look inside the most likely path between the basic blocks A and B. These are shown in Figure 3. The likely path between the basic blocks A and B are shown by thick arrow. In Figure 3(a), basic blocks A and B are not very far and also by including the instructions executed in the likely path between A and B, (including B) in thread 1, the size of thread 1 is not going to violate the upper limit. So the compiler does not spawn a new thread at B. Rather the compiler includes all blocks between A and B in thread 1 and looks beyond B to find the next potential thread starting point. In figure 3 (b), B is not too close to A and yet not too far from A. So B is a potential thread starting point. So the compiler marks B as the starting

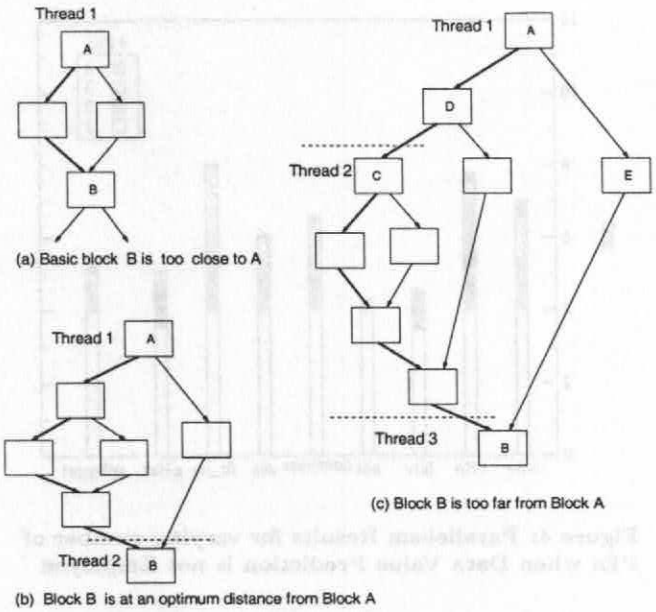


Figure 3: Different Cases in Program Partitioning

point of thread 2 and forms a thread at B. Now it checks the data dependence between the thread containing A and the thread containing B according to the data dependence distance or the data dependence count. If it is found that the total completion time of threads 1 and 2 (where thread 2 is spawned from the beginning of B), is less than the completion time if the two threads are executed sequentially one after another, then it spawns thread 2 from A.

In figure 3(c), A and B are very far apart, as far as the most likely path between them are concerned. So, starting a new thread at B and including all the blocks till B in thread 1 is not efficient. First of all, the size of thread 1 will become very large and moreover there may exist potential threads inside the likely path between A and B. So the compiler looks inside the likely path between A and B and tries to partition it further. In case of 3(c) it is found that basic block C is a starting point of thread 2 and this thread is speculatively spawned from A, before the actual direction of the branch is resolved. Thread 3, which starts from basic block B is spawned from somewhere inside thread 2.

The compiler also checks the paths that are not the likely paths and partitions them as well. If at run-time, control goes into those unlikely paths, then the threads spawned speculatively are aborted. But the threads that are not control dependent on the aborted threads need not be aborted. For example, consider Figure 3 (c). If from A, instead of following the most likely path, the control goes to basic block D, when both threads 2 and and 3 have been spawned, thread 2, would be aborted, but not thread 3, as B is control independent of A. Moreover in the path containing D, there can be spawning of thread 3 as well, and this spawning should be ignored during execution because thread 3 has already been spawned.

In our compiler framework, the loops are treated as a special case of control dependence. For loops the compiler checks the dependence between two iterations of the loops, and if it is found that spawning another thread for the next iteration is profitable, then the thread is spawned. It may also happen that, instead of spawning from the beginning of the loop for the next iteration, the compiler spawn the next iteration from somewhere inside the loop. The large body of the loops may be further partitioned into multiple threads as described above. While partitioning the loops, we use profile information on the number of loop iterations. Typically the compiler does not want to execute small loop body in parallel. However, if the number of iterations is large then the compiler would spawn the iterations as separate threads. Otherwise the size of the thread will become very large.

3.6 Implementation in the SUIF Platform

Our thread partitioning algorithm has been implemented on the SUIF-MachSUIF platform [9]. All of the compiler analysis and thread partitioning are done at the high-level intermediate representation (IR) of SUIF. We have chosen the SUIF platform to implement our compiler system because it provides a modular and flexible infrastructure to develop compiler optimizations. SUIF first translates high-level source code into an IR, and then performs code optimization through several independent passes on that IR. While transforming high-level programs into IR, SUIF retains all of the relevant information from the high level source program. This is particularly helpful for carrying out optimization such as profiling and pointer analysis. Moreover, the instructions in the SUIF IR are very close to the assembly level instructions; thus, the estimation of thread sizes done at the IR level remains valid in the final assembly level as well. In SUIF, it is possible to annotate the instructions with necessary information like data dependence, and use them in separate passes afterwards. Also, the SUIF package contains many optimization modules, which improve the quality of the code produced. We have used the MachSUIF [17] framework to generate Alpha assembly code from the SUIF IR.

4. EXPERIMENTAL STUDY

In order to see how much of the parallelism measured in Section 2.2 can potentially be tapped by our compiler-directed multithreading approach, we enhanced our software tool (TAPE) along the lines of the simulation environment used in [15] to study parallelism in general-purpose applications. The number of PEs, issue size per PE, etc., are parameterized. It models a perfect instruction cache and data cache. The code executed in the supervisor mode are unavailable to the simulator, and are therefore not taken into account in the measurements. Furthermore, the simulator does not overlap the execution of threads that precede and succeed a system call. For these measurements, each PE has an issue width of 4 instructions per cycle, an instruction window of 32 entries, and can perform out-of-order execution.

When encountering a conditional branch instruction in a thread, its PE consults a branch predictor for a prediction. If the prediction is incorrect, the immediate control-independent point in the program is determined. If this point is within the thread, then subsequent threads are not squashed. Branch predictions are done using a 2-level Pap scheme [20], with a direct-mapped 16K-entry Branch History Table, a pattern size of 6, and 3-bit saturating counters in the Pattern History Table entries.

A data value predictor is implemented in the simulator. This predictor is a hybrid of a stride predictor and a 2-level predictor [19]. The first level of the predictor has 16K entries, and is direct-mapped. Data value prediction is carried out only for those instructions that produce a single register result. Thus branch instructions, store instructions, nops, and double-precision instructions are not considered for data value prediction.

4.1 Code Explosion Due to Thread Information

As mentioned earlier, for embedded system applications, it is important that our thread partitioning does not have a significant impact on the code size. Table 2 gives the increase in code size because of including thread information in the program binary. For each benchmark program, the table provides the number of static instructions for Alpha ISA, the number of instructions with annotation, and the percentage of instructions with annotation. From the table, we can see that the code size expansion ranges from 0.12% to 2.69%, with an average of 0.82%, which is very insignificant. The number of instructions that are annotated does not depend on the ISA; rather it depends on the program characteristics. For annotating an instruction, we need 16/32 bits to specify an instruction address and 2 bits to specify the type of annotation. Note that for embedded processors where memory size is usually smaller, the number of bits required to specify the instruction address will be even less. Also, we can use special hardware and have relative addressing scheme to specify the instruction address in the annotations, thereby reducing the space requirements further.

Benchmark	Static Instruction Count	Additional Instrs for Conveying Thread Info	Code Expansion Factor
auto	1359	5	0.37%
bffo	1339	6	0.45%
bilv	1671	2	0.12%
blit	1495	8	0.54%
compress	2190	59	2.69%
des	2007	8	0.40%
g3fax	1561	17	1.09%
ucbqsort	1787	16	0.90%
Average	1676.12	15.12	0.82%

Table 3: Increase in code size of the benchmark programs because of including thread information

4.2 Parallelism Without Data Value Prediction

Our first set of multithreading studies were done without employing data value prediction. Figure 4 presents the overall parallelism in terms of instructions per cycle (IPC) obtained in these experiments, with different number of processing elements (PEs). All benchmarks are simulated till hundred million instructions unless the programs get completed before that.

From Figure 2, we can see that across all benchmark programs, there is notable speedup with 4 PEs, except for `bilv`, `blit`, `g3fax`, and `ucbqsort`. Among these, `g3fax` did not show any parallelism in the measurements in Section 2. Even for the ones that show promise, the parallelism saturates when the number of PEs reaches 2 or 3. Thus, without data value prediction, multithreading has limited use for these embedded applications.

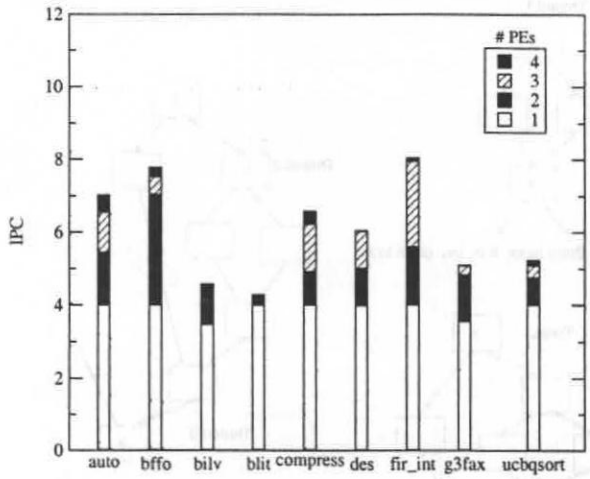


Figure 4: Parallelism Results for varying number of PEs when Data Value Prediction is not Employed

4.3 Parallelism With Data Value Prediction

Next, we present the parallelism values obtained when data value prediction was employed in the multithreaded processor. Figure 5 presents the parallelism values obtained in these experiments, with data value prediction. For ease of comparison, the results from Figure 2 are reproduced alongside.

When data value prediction is employed, two of the four benchmarks that did not show much parallelism—`blit` and `ucbqsort`—show a marked improvement. In addition, `bffo` shows further improvement. The most notable speedup is seen for `blit`. On the other extreme, `bilv` does not show any noticeable speedup with multi-threading, even when data value prediction is employed. Most programs have substantial speedups with multithreading. By and large, incorporating data value prediction helps to reduce the effects of inter-thread data dependences, thereby providing notable speedups. Thus, we can see that multithreading is quite effective for embedded systems programs when the processor employs data value prediction, which is quite encouraging.

4.4 Importance of Non-loop Threads

The experimental measurements conducted so far included threads that are loop-centric (iterations of loops) as well as non-loops. In the next set of measurements, we measure the parallelism obtained when only loop-centric threads are employed. Figure 6 presents these results. For each benchmark, two bars are given. The first corresponds to using all kind of threads, and the second corresponds to using only loop-centric threads. Among the benchmarks, only a single program—`bffo`—benefits from using only loop-centric threads. For most of the programs, restricting to loop-centric threads results in less parallelism being exploited. This demonstrates the importance of a multithreading frame-

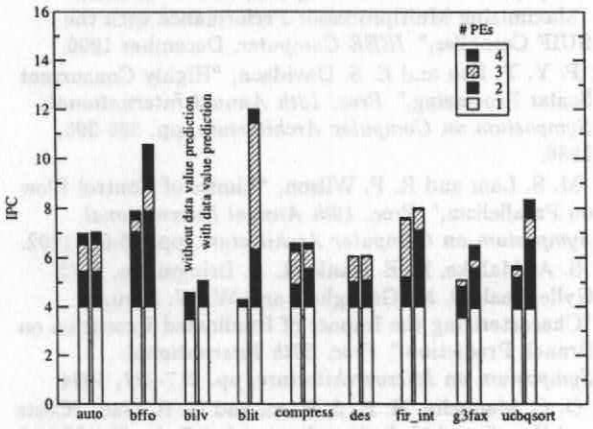


Figure 5: Parallelism Results for varying number of PEs when Data Value Prediction is Employed

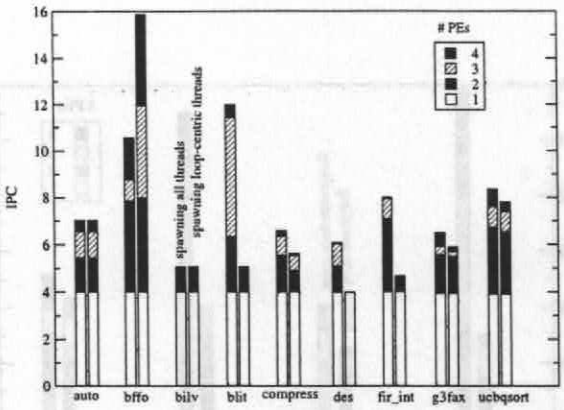


Figure 6: Parallelism Results for varying number of PEs when Using Different Types of Threads

work that supports loop-centric as well as other kind of threads.

4.5 Effect of Selective Loop Unrolling

Some of the embedded applications are dominated by very small loops, whose iterations are fairly independent. If each iteration of such a loop is partitioned as a single thread, then each thread becomes very small, and the thread overhead becomes too much. On the other hand, if the entire loop is made a single thread, then we fail to exploit the inter-iteration parallelism present in these loops. In order to deal with this problem, we experimented with selective unrolling of loops. Selective unrolling is very important to keep the code expansion factor low. We introduced a selective unrolling pass in our compiler framework; currently this pass unrolls simple "`for`" loops with fixed upper and lower bounds. The use of selective unrolling showed substantial potential for 4 of the benchmarks—`auto`, `compress`, `fir_int`, and `ucbqsort`. The code expansion due to selective loop unrolling for these 4 benchmarks were 58.8%, 15.1%, 33.11%, and 7.0%, respectively. The improvements in thread-level parallelism are shown in Figure 5. For each of these 4 benchmarks, 2 bars are shown, the first indicating the parallelism before loop unrolling and the second indicating the parallelism after loop unrolling. Among these two benchmarks, `auto` and `fir_int` show remarkable improvement due to selective loop unrolling.

5. DISCUSSION AND CONCLUSIONS

Embedded processor designs are constrained by power consumption, code size, and die size limitations. Nevertheless, the performance expected from them has been steadily increasing. Today's embedded processors incorporate a variety of techniques that are used for general-purpose processor design.

To obtain high performance in embedded system applications, it is important to handle both loop-terminating branches and other conditional branches in an efficient manner. Although traditional branch prediction provides somewhat reasonable prediction accuracies for embedded system application programs, a substantial increase in performance for next-generation systems requires more effective ways of dealing with conditional branches. Recognizing control-independent regions, and performing speculative execution along multiple (independent) flows of control have the potential to extract the large amounts of parallelism that are available at a distance. Given the increasing interest in multithreading for general-purpose processors, we expect that future embedded processors will also attempt to execute multiple threads in one way or another.

This paper investigated the applicability of multithreading in embedded system applications. We partitioned programs from the Powerstone benchmark suite, a collection of programs from the embedded systems area, into multiple threads. While performing the partitioning, the compiler not only considers control independence information, but also considers data dependence information and profile-based information on the most likely control flow paths. We performed several measurements with these compiler-generated threads. Our measurements show that a majority of the benchmarks programs are able to get substantial increase in parallelism when up to 4 threads are executed in parallel, provided data value speculation is used to break inter-thread data dependences. Our measurements also show that most of the benchmark programs require non-loop threads also, in addition to loop-centric threads. Results from these simulations indicate that future processors can speed up the execution of embedded system program by using multithreading.

A major advantage of speculative multithreading multithreading is backward compatibility with existing processors. That is, an existing executable program for the original (single-threaded) embedded processor forms legal single-thread code for a multithreading embedded processor. This feature is very important from the commercial point of view, because of customers' strong preference to have the ability to run the old binaries in the new machine (although those binaries can not benefit from the new machine's multithreading features).

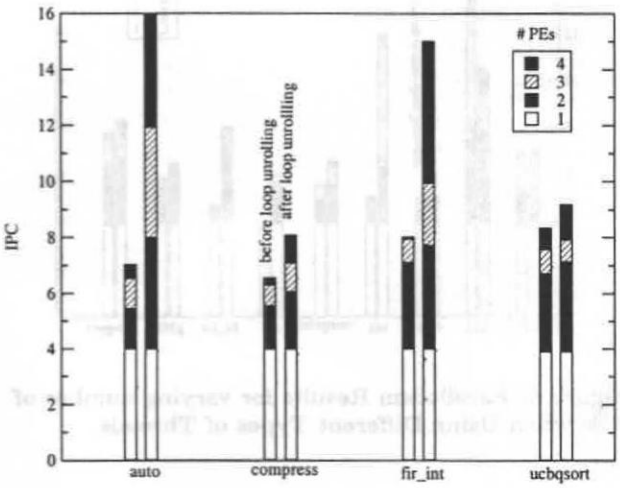


Figure 7: Parallelism Results for varying number of PEs when Selective Loop Unrolling is Employed

Acknowledgements

This work was supported by the U.S. National Science Foundation (NSF) through a CAREER grant (MIP 9702569), a regular grant (CCR 0073582), and an REU grant (EIA 9912218).

6. REFERENCES

[1] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[2] H. E. Bal and M. Haines, "Approaches for Integrating Task and Data Parallelism," *IEEE Concurrency*, July-September 1998.

[3] A. Bhowmik and M. Franklin, "A Characterization of Control Independence in Programs," *Proceedings of Workshop on Workload Characterization*, 1999.

[4] R. Cytron, J Ferrante, B. Rosen, M. Wegman, and F. Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph," *ACM Transactions on Programming Languages and Systems*, Vol. 13, No. 4, pp. 451-490, October 1991.

[5] W. J. Dally and S. Lacy, "VLSI Architecture: Past, Present, and Future," *Proceedings of Advanced Research in VLSI Conference*, 1999.

[6] P. Dubey, K. O'Brien, K. M. O'Brien, and C. Barton, "Single-Program Speculative Multithreading (SPSM) Architecture: Compiler-assisted Fine-Grained Multithreading," *Proc. International Conference on Parallel Architecture and Compilation Techniques (PACT '95)*, 1995.

[7] M. Emami, "A Practical Interprocedural Alias Analysis For an Optimizing/Parallelizing C Compiler," *Masters Thesis*, School of Computer Science. McGill University, Montreal, 1993.

[8] M. Franklin, "The Multiscalar Architecture," *Ph.D. Thesis, Technical Report 1196*, Computer Sciences Department, University of Wisconsin-Madison, 1993.

[9] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S. W. Liao, E. Bugnion, and M. S. Lam, "Maximizing Multiprocessor Performance with the SUIF Compiler," *IEEE Computer*, December 1996.

[10] P. Y. T. Hsu and E. S. Davidson, "Highly Concurrent Scalar Processing," *Proc. 13th Annual International Symposium on Computer Architecture*, pp. 386-395, 1986.

[11] M. S. Lam and R. P. Wilson, "Limits of Control Flow on Parallelism," *Proc. 19th Annual International Symposium on Computer Architecture*, pp. 46-57, 1992.

[12] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher, and W. W. Hwu, "Characterizing the Impact of Predicated Execution on Branch Prediction," *Proc. 27th International Symposium on Microarchitecture*, pp. 217-227, 1994.

[13] O. C. Maquelin, H. H. J. Hum, and G. R. Gao, "Costs and Benefits of Multithreading with Off-the-Shelf RISC Processors," *Proceedings of the First International EURO-PAR Conference* (Seif Haridi, Khayri Ali, and Peter Magnusson, eds.), no. 966 in Lecture Notes in Computer Science, Stockholm, Sweden, pp. 117-128, Springer-Verlag, August 29-31, 1995.

[14] K. Olukotun, et al, "A Chip-Multiprocessor Architecture with Speculative Multithreading," *IEEE Transactions on Computers*, September 1999.

[15] J. Oplinger and D. Heine and M. S. Lam. In Search of Speculative Thread-Level Parallelism. *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 1989.

[16] J. Scott, L. H. Lee, A. Chin, J. Arends, and B. Moyer, "Designing the MCORE M3 CPU Architecture," *Proc. International Conference on Computer Design (ICCD)*, 1999.

[17] M. D. Smith, G. Holloway, "An Introduction to Machine SUIF and Its Portable Libraries for Analysis and Optimization".

[18] K. B. Theobald, G. R. Gao, and L. J. Hendren, "On the Limits of Program Parallelism and its Smoothability," *Proc. 25th Annual International Symposium on Microarchitecture (MICRO-25)*, pp. 10-19, 1992.

[19] K. Wang and M. Franklin, "Highly Accurate Data Value Prediction using Hybrid Predictors", *Proceedings of 13th IEEE/ACM International Symposium on Microarchitecture*, pp. 281-90, Dec. 1997.

[20] T-Y. Yeh and Y. N. Patt, "Alternative Implementations of Two-Level Adaptive Branch Prediction," *Proc. 19th Annual International Symposium on Computer Architecture*, pp. 124-134, 1992.

Prefetching in an Intelligent Memory Architecture Using a Helper Thread *

Yan Solihin[†], **Jaejin Lee**[‡], and **Josep Torrellas**[†]
[†]University of Illinois at Urbana-Champaign
[‡]Michigan State University
{solihin,torrellas}@cs.uiuc.edu
jlee@cse.msu.edu
http://iacoma.cs.uiuc.edu/flexram

Abstract

Data prefetching is a popular technique for tolerating long memory access latencies. In this paper, we introduce a novel type of prefetching: memory-side correlation prefetching implemented in a user-level thread. The prefetching thread runs on a general-purpose processor embedded in the main memory. By allocating the correlation table in main memory, we can afford the large space required by the table. In addition, the scheme can be supported with few modifications to the L2 cache and no modification to the main processor core. We introduce a new organization of the correlation table and a new prefetching algorithm that enable fast and accurate far-ahead prefetching with high coverage. Overall, our evaluation shows that the algorithm effectively prefetches irregular applications, speeding up three applications by an average of 1.28. Furthermore, our scheme can work synergistically with a conventional processor-side prefetcher to deliver an average speedup of 1.36.

1 Introduction

Data prefetching is a popular technique to tolerate long memory access latencies. There have been many proposals using a *helper thread* to help prefetching for the *main thread*, such as [12, 15]. These proposals have focused on either SMT or CMP platforms. In this paper, we propose a prefetching thread scheme that is suitable for implementation in an *Intelligent Memory Architecture* (IMA). In IMA, the memory system is augmented with one or more memory processors. The nature of the problems in IMA is quite different than in SMT or CMP platforms. First, in SMT/CMP, *Processor-Side* prefetching is used, while in IMA, *Memory-Side* prefetching is used, because prefetch requests are generated by the processor in the main memory. Secondly, communication between the threads is cheap in SMT/CMP, while it is expensive in IMA. Thus, a suitable prefetching scheme is one that operates autonomously and that can be effective with coarse-grain communication between the prefetching and the main threads. In this work, we implement the prefetcher as a user-level thread that can prefetch irregular applications effectively using correlation prefetching algorithms. The only communication needed by the prefetching thread is the miss address stream of the main thread.

Memory-side prefetching is attractive for several reasons. First, it eliminates the overheads that prefetch requests and state bookkeeping introduce in the paths between the main processor and its caches. Secondly, it can be supported with very few modifications to the L2 cache and no modification to the main processor core. Thirdly, the prefetcher can exploit its proximity to the memory to its advantage. Memory-side prefetching has the additional attraction of riding the technology trend of increased chip integration. Indeed, popular platforms like PCs are being equipped with graphics engines in the memory system [16]. Some chipsets, like NVIDIA's nForce [13] even integrate a powerful processor in the North Bridge chip. Similar engines can be provided for prefetching, or existing graphics processors can be reused for prefetching when under-utilized. Moreover, there are proposals to integrate processing logic in DRAM chips, such as IRAM [8].

Using an engine for memory-side prefetching has been proposed elsewhere [1, 2, 4, 13, 14, 16, 18]. However, in most cases, these engines perform either very simple operations or highly-specific operations, such as prefetching linked data structures [4, 18]. Instead, what we would like, is a very flexible, general-purpose prefetcher.

While a memory-side prefetcher can support a variety of prefetching algorithms, one type that is particularly suitable is Correlation Prefetching [1, 3, 5, 11]. Correlation prefetching relies on correlation of miss addresses to predict and prefetch future misses based on the current state. Because the only information the prefetch thread needs is the miss address stream, correlation prefetching is suitable for an IMA platform.

In the past, general correlation prefetching has been supported by hardware controllers that require a large dedicated hardware table structure [1, 3, 5, 11]. In all but one case, these controllers have been placed between the L1 and L2 caches or between the L1 and the processor. While effective, the approach has a very high hardware cost. Furthermore, it does not prefetch far enough and tends to have a low coverage.

*This work was supported in part by the National Science Foundation under grants CCR-9970488, EIA-0081307, and EIA-0072102, by DARPA under grant F30602-01-C-0078, and by NCSA, Michigan State University, and gifts from IBM and Intel.

This paper introduces a novel prefetching scheme where memory-side correlation prefetching algorithms are implemented in software by using a user-level thread. The algorithms run on a general-purpose processor in the main memory system. The scheme allows prefetching algorithms to evolve with the applications, even after the computer system is shipped. In addition, the system can be supported with few modifications to the L2 cache, and no modifications to the main processor core.

We introduce a new organization of the correlation table and a new correlation prefetching algorithm that enable fast and far-ahead prefetching, with high coverage and accuracy. By allocating the correlation table in main memory, we can afford the large space required by the table. We demonstrate that the software algorithm can effectively prefetch data for irregular applications. Indeed, our scheme speeds up three SPECInt2000 applications by an average of 1.28. We also show that our scheme can work synergistically with a conventional processor-side prefetcher to deliver an average speedup of 1.36.

The rest of the paper is organized as follows: Section 2 discusses memory-side prefetching and correlation prefetching; Section 3 presents our design; Section 4 discusses our evaluation setup; Section 5 evaluates our design; and Section 6 concludes.

2 Related Issues

2.1 Memory-Side Prefetching

Memory-Side prefetching occurs when prefetching is initiated by one or a set of engines that reside in or beside the main memory (definitely beyond any memory bus). Chip manufacturers have integrated hardwired controllers that probably recognize very simple sequences like strides, such as NVIDIA's DASP engine in the North Bridge chip [13] and Intel's prefetch cache in its i860 chipset.

In this paper, we propose to use a simple general-purpose memory processor for memory-side prefetching. Although this idea is applicable to a generic memory system, we will illustrate it on a PC-like memory system depicted in Figure 1-(a). The memory processor can be placed in several places, such as in the North Bridge (Memory Controller) chip (1), or in the DRAM chips (2). The advantages of the first case are that it is simple to support, because the DRAM interface is not modified, and that the memory processor can be employed for other uses, such as a graphics engine. The second case, although more complicated to support, has the advantage of lower memory access latency and higher memory bandwidth due to higher integration. In this paper, we study the performance potential of the DRAM case.

Memory- and processor-side prefetching are not the same as *Push* and *Pull* (or *on-demand*) prefetching [18], respectively. *Push* prefetch occurs when prefetched data is sent to a cache or processor that has not requested it, while *pull* prefetch is the opposite. Clearly, a memory prefetcher can act as a *pull* prefetcher, by simply storing the prefetched data in a local buffer and supplying it to the processor on demand. In general, however, memory-side prefetching is most interesting when it performs *push* prefetching to the caches of the processor, because it can hide a larger fraction of memory access latency.

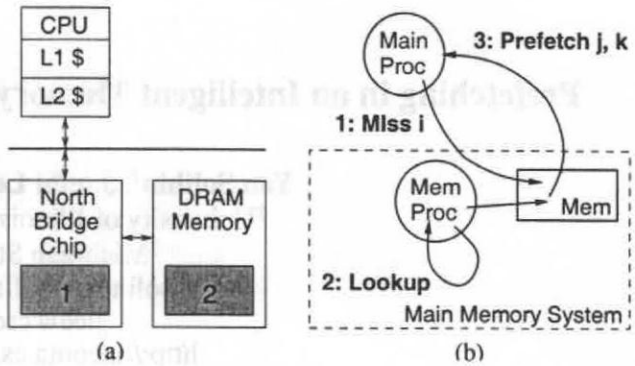


Figure 1: Architecture of the system (a), and actions of the prefetches (b).

In our system, the memory processor observes the requests from the main processor that reach main memory. Based on them, and after examining some internal state, the memory processor prefetches other lines that it expects the main processor to need in the future (Figure 1-(b)).

In this paper, we concentrate on *push prefetching* into the L2 cache. Since the memory processor only sees L2 cache miss streams, it aims to eliminate L2 cache misses by pushing the prefetched data into the L2 cache. L2 cache miss penalty is the largest component of memory access latency, and it is the hardest to hide, even by an out-of-order processor.

Our scheme is inexpensive to support. The main processor core does not need to be modified at all. The L2 cache needs to have the following supports. First, as in many other systems [4, 7], the L2 cache controller has to be able to accept lines from the memory system that it has not requested. To do so, the L2 has to assign unused *Miss Status Handling Registers* (MSHRs) [10] to such lines. Secondly, if the L2 has a pending request for the same line when a prefetch arrives, the prefetch simply steals the MSHR and updates the cache as if it were the reply. Finally, a prefetched line arriving at L2 is dropped in the following cases: the L2 cache already has a copy of the line, the write back queue has a copy of the line because the L2 is trying to write it back to memory, all MSHRs are full, or all the lines in the set where the prefetch line wants to go are in pending state.

2.2 Correlation Prefetching

Correlation Prefetching uses the current state of the reference or miss stream to predict and prefetch future misses. Two popular correlation schemes are the *Stride-Based* and *Pair-Based* schemes. The former tries to find a stride pattern in the miss stream and prefetch all the locations that would be accessed if the pattern continues in the future. The lat-

ter tries to identify a correlation between pairs of misses, for example between a miss and its immediate successor. It basically records a sequence of miss addresses in a table, and later when it encounters the head of the sequence, it looks up the table and prefetches the rest of the sequence. What makes pair-based schemes attractive is their general applicability, i.e. they work for any miss sequences that repeat. This is true for regular applications and for a wide range of irregular applications such as those that operate on sparse matrices and linked data structures. Furthermore, the schemes can be employed without any compiler support or changes in the application binaries.

Pair-based correlation prefetching has only been studied using a hardware implementation of prefetch engines [1, 3, 5, 11, 17], usually by placing the engine between the L1 and L2 cache [3, 5, 11, 17]. These studies have demonstrated the applicability of pair-based correlation prefetching on a wide variety of applications. However, they also reveal shortcomings of the approach. One critical problem is that to be effective, it needs large storage space to match the footprints of the applications. One and two Megabytes of dedicated on-chip SRAM tables have been proposed [5, 11], while some applications with larger footprints even need a 7.6 MB off-chip SRAM table [11]. Furthermore, it does not prefetch far enough and has low coverage (unless it is tightly coupled to the main processor and uses more fine grain information [11]). For example, for each miss, Joseph and Grunwald only store *immediate* successors [5]. The coverage is low because it needs one miss to trigger the prefetcher to prefetch the successor of the miss. At best only half of the misses can be eliminated. This scheme uses a wide table that stores many successors per miss and continuously rebuilds the table to increase the coverage. However, it causes excessive useless prefetches.

3 Proposed Scheme

Pair-based correlation prefetching is suitable for our memory-side prefetching system to support because it has general applicability and can be supported inexpensively. We show that shortcomings of the current correlation prefetching schemes can be eliminated by improving the correlation algorithms and implementing them in software. The algorithms described are implemented in a prefetching thread running on the memory processor. The code for the prefetching thread is written in C and hand-optimized for minimal prefetch response and occupancy time.

In the following sections, we discuss the concepts (Section 3.1), the architecture (Section 3.2), pair-based correlation prefetching algorithms (Section 3.3), and conventional processor-side prefetching (Section 3.4).

3.1 Concepts

Prefetching algorithms are implemented as a user-level helper thread that we call *prefetching thread*. The actions of the memory processor are determined by the behavior of the prefetching thread that we implement. The operation of the prefetching thread can be conceptually divided into two phases: *learning* and *prefetching*. In the learning phase, the prefetching thread records the L2 read and write miss patterns that it observes in a correlation table, one miss at a time. In the prefetching phase, every time that the prefetching thread sees a miss, it looks up the correlation table and prefetches several memory lines to the L2 cache of the main processor. No action is taken on a write-back memory access. In practice, as in [5], we found that combining the learning and prefetching phases enables the correlation table to quickly learn new patterns and provides the best performance in most cases (Figure 2).

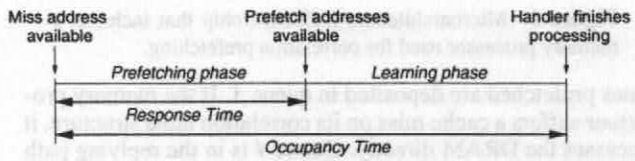


Figure 2: Timing of the prefetching thread.

The prefetching algorithm can be characterized by its *response time* and *occupancy time* (Figure 2). The response time is defined as the time beginning when the prefetching thread obtains a miss address until the prefetching thread produces the prefetch addresses. The occupancy time is the time the prefetching thread is busy and cannot process another miss address. As can be seen in the figure, the prefetching phase is always executed before the learning phase to minimize the response time. For the software implementation to be viable, the occupancy time has to be smaller than the average time between two consecutive L2 cache misses. Also, for best performance, the response time needs to be as small as possible.

By using a prefetching thread that stores the correlation table in the main memory, we eliminate the high hardware cost required by the table in the traditional implementation. We further address the inadequacies of traditional correlation prefetching, namely low prefetching coverage, and not prefetching far enough, by improving the correlation algorithms (Section 3.3).

3.2 Architecture of the System

When we integrate the memory processor in the DRAM chips, the DRAM chips and possibly the DRAM interface need to be modified. Extra complexities in handling multiple DRAM chips must also be addressed. Our goal in this paper is to study the performance potential of this case. Consequently, we abstract away the implementation complexity of integrating the processor in the DRAM by assuming a single chip main memory with a single memory processor in it (Figure 3).

The key communication occurs through queues *1*, *2*, and *3*. Miss requests from the main processor are deposited in queues *1* and then in *2*. In the learning phase, the memory processor uses the entries in queue *2* to build its state. In the prefetching phase, the memory processor uses the entries in queue *2* and its state to generate addresses to prefetch. The

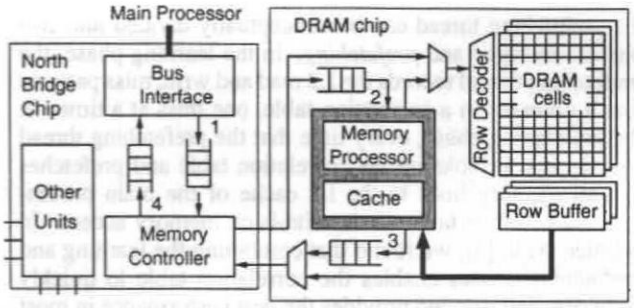


Figure 3: Microarchitecure a DRAM chip that includes a memory processor used for correlation prefetching.

lines prefetched are deposited in queue *3*. If the memory processor suffers a cache miss on its correlation table structure, it accesses the DRAM directly. Queue *4* is in the replying path from memory to the main processor.

3.3 Pair-Based Correlation Algorithms

We now discuss the pair-based correlation prefetching algorithms. We consider two different organizations for the correlation table: a basic one that does not allow data replication and a more advanced one that allows replication. Their use gives rise to different algorithms. We consider them in turn.

Pair-Based Algorithms with Basic Table Organization

Each row in this table stores the tag of the miss address, and the addresses of a set of *immediate* successor misses stored in MRU order. We consider two algorithms that use this basic organization: *Base* and *Chain*.

Base follows the scheme proposed by Joseph and Grunwald [5]. For any given miss, *Base* is only interested in prefetching *immediate* successor misses. The parameters of the algorithm are the number of immediate successors predicted (*NumSucc*), the number of misses that the correlation table can store predictions for (*NumRows*), and the associativity of the correlation table (*Assoc*).

Base is illustrated in Figure 4-(a). It shows two snapshots of the correlation table at the point that the corresponding miss trace has been consumed (i and ii). In the example, *NumSucc* is 2, *NumRows* is 4, and *Assoc* is 1. Within a row, successors are replaced using LRU replacement policy. As in Joseph and Grunwald's study [5], we find that LRU replacement policy for the successors in each row works best. The figures show the successors in MRU order from left to right. In the learning phase, the processor keeps a pointer to the row of the last miss observed. When a miss occurs, its address is placed as one of the immediate successors of the last miss, and a new row is allocated for the new miss unless an entry for the address already exists. In the prefetching phase (iii), when a miss is observed, the processor finds the corresponding row and prefetches all the *NumSucc* immediate successors, starting from the MRU one.

Since *Base* only prefetches immediate successors, its coverage and latency hiding capabilities are limited. To improve this, we propose the *Chain* algorithm, which for every miss prefetches multiple levels of successors. The algorithm takes one extra parameter called *NumLevels*, which is the number of levels of successors prefetched. The algorithm is illustrated in Figure 4-(b).

In the learning phase, *Chain* is identical to *Base* (i and ii). However, *Chain* does more work in the prefetching phase (iii). After prefetching the row of immediate successors, it takes the most recently-used successor among them and indexes the correlation table with its address. If the entry is found, it prefetches all *NumSucc* successors there. Then, it takes the most recently used successor in that row and repeats the process for *NumLevels-1* times. As an example, suppose that a miss on line *a* occurs (iii). The memory processor first prefetches *d* and *b*. Then, it takes the MRU entry *d*, looks-up the table, and prefetches *d*'s successor, *c*.

While improving the coverage and far-ahead prefetching capability over *Base*, *Chain* has two limitations. One limitation is that the response time of the algorithm is high. To issue prefetches in response to a miss, it needs to make *NumLevels* accesses to different rows in the table, each possibly involving a low-associative search and potentially causing a cache miss. The second limitation is that it does not prefetch the *correct MRU* successors of each level of successors. Instead, it only prefetches successors found along the MRU path.

Pair-Based Algorithms with Replicated Table Organization

Each row in this table stores the tag of the miss address, and *NumLevels* levels of successors. Each level contains *NumSucc* addresses, which are MRU-ordered.

We propose a a new algorithm called *Replicated* that exploits this table organization. Replicated takes the same parameters as *Chain*. In the learning phase, *NumLevels* pointers to the table are kept for efficient access, pointing to the rows for the address of the last miss, second last, and so on. When a miss occurs, its address is recorded in the correct position of MRU successors of the last few misses by using these pointers. Figures 4-(c) illustrates the algorithm. In the example, *NumSucc* is 2, *NumRows* is 4, *Assoc* is 1, and *NumLevels* is 2. The figure shows two snapshots of the correlation table in the learning phase at the point where the corresponding miss trace has been consumed (i and ii). The figure also shows the position of the two pointers, and the algorithm in prefetching phase (iii).

Note that this organization solves the two problems of *Chain*. First, the response time is much shorter. We can prefetch several levels of successors with a single row access, possibly with only one cache miss. In fact, we shift some computation from the prefetching phase, which is the critical phase, to the learning phase. Now the learning phase needs to update several rows in the table. However, the rows are most likely still in the cache and, since we keep the pointers to the entries of last few miss addresses, the associative search is avoided. Secondly, by grouping together all the successors from a given level, we can identify the correct MRU successors from that level, yielding higher accuracy.

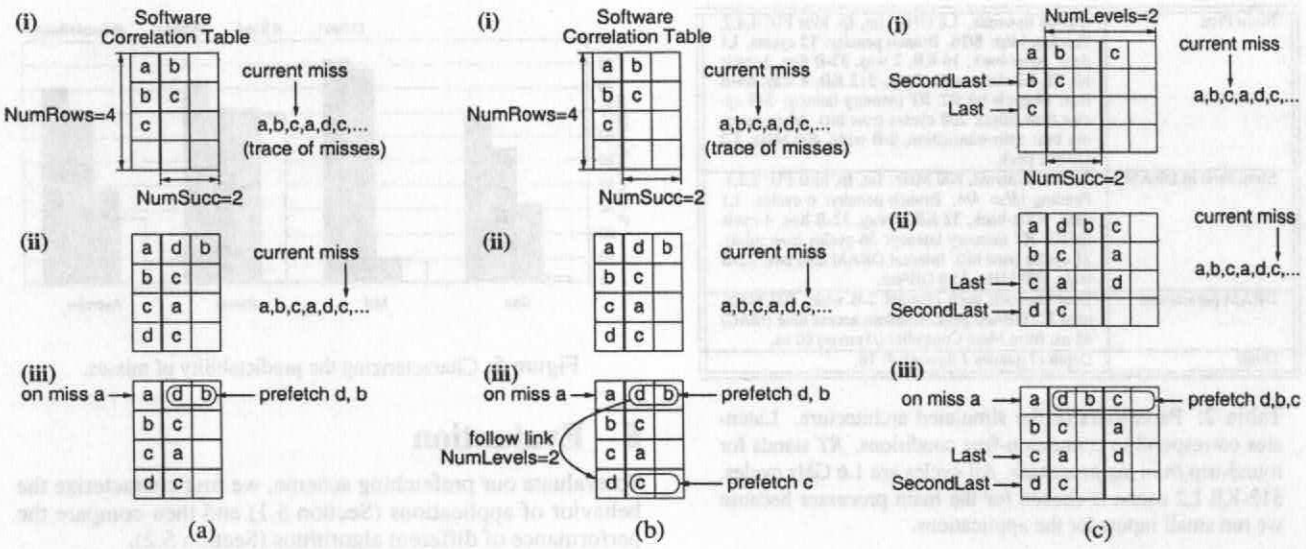


Figure 4: Pair-based correlation algorithms: *Base* (a), *Chain* (b), and *Replicated* (c).

Characteristics	*Base*	*Chain*	*Replicated*
Levels of successors prefetched	1	*NumLevels*	*NumLevels*
Full MRU ordering for each level?	Yes	No	Yes
Num. row accesses in the prefetching phase (SEARCH)	1	*NumLevels*	1
Num. row accesses in the learning phase (NO SEARCH)	1	1	*NumLevels*
Response Time	Low	High	Low
Space requirement (for constant number of prefetches)	x	x	$NumLevels \times x$

Table 1: Comparing the different pair-based algorithms.

Algorithm Comparison

Table 1 compares the three pair-based schemes. From the table, we see that *Replicated* algorithm tries to solve problems in current correlation prefetching algorithms: it looks far ahead by prefetching several levels of successors, thereby improving coverage, while keeping high accuracy by prefetching the correct MRU successors in each level. Its only shortcoming is its high space requirements for the correlation table. Fortunately, this is a minor issue, since the table is allocated in the main memory.

The response time is better with the *Replicated* algorithm than with the *Chain* algorithm. The handler in *Replicated* runs very efficiently because cache lines are well utilized. Note that all the correlation algorithms could be implemented in hardware. However, *Replicated* is very suitable for a software implementation because it has a low response time, far-ahead prefetching capability, and uses cache lines well.

3.4 Conventional Prefetching

Previous studies found that placing a stride-based prefetcher as a front end of a pair-based prefetcher makes pair-based prefetching more effective [3, 17]. We exploit this finding by including processor-side prefetching in the form of a hardware multi-stream sequential prefetcher at the L1 cache. The prefetcher has similar capabilities to stream buffers [6], except that the prefetch lines are put directly in the L1 cache.

In our system, we assume that the memory controller can distinguish the prefetches issued by the processor-side prefetcher from regular misses. The memory controller chooses not to pass such prefetches to the memory processor. As a result, in general, the processor-side prefetcher targets the regular misses while the memory-side prefetcher targets the irregular ones.

4 Evaluation Environment

Applications. To evaluate our prefetching scheme, we use three mostly irregular memory-intensive applications from the SPECInt2000 suite. Irregular applications are hardly amenable to compiler-based prefetching. Consequently, they are the obvious target for the type of prefetching that we propose. We choose *Gap*, *Mcf*, and *Parser*. *Gap* uses a subset of the test input set, *Mcf* uses the test input set, and *Parser* uses a subset of the train input set.

Simulation Environment. The evaluation is performed using execution-driven simulation. Our environment is based on an extension to MINT that supports dynamic superscalar processor models with register renaming, branch prediction, and non-blocking memory operations [9].

The architecture modeled is that of a high-end PC with a

Main Proc	6-issue dynamic, 1.6 GHz. Int, fp, ld/st FU: 4,4,2. Pending ld/st: 8/16. Branch penalty: 12 cycles. L1 data: write-back, 16 KB, 2 way, 32-B line, 3-cycle hit RT. L2 data: write-back, 512 KB, 4 way, 64-B line, 19-cycle hit RT. RT memory latency: 243 cycles (row miss), 208 cycles (row hit). Main memory bus: split-transaction, 8-B wide, 400 MHz, 3.2 GB/sec peak.
Mem Proc in DRAM	2-issue dynamic, 800 MHz. Int, fp, ld/st FU: 2,2,1. Pending ld/st: 4/4. Branch penalty: 6 cycles. L1 data: write-back, 32 KB, 2 way, 32-B line, 4-cycle hit RT. RT memory latency: 56 cycles (row miss), 21 cycles (row hit). Internal DRAM data bus: 32-B wide, 800 MHz, 25.6 GB/sec.
DRAM parameters	Dual channel; each channel 2-B wide, 800 MHz; total 3.2 GB/sec peak. Random access time (*tRAC*) 45 ns; from Mem Controller (*tSystem*) 60 ns.
Other	Depth of queues *1* through *4*: 16.

Table 2: Parameters of the simulated architecture. Latencies correspond to contention-free conditions. *RT* stands for round-trip *from the processor*. All cycles are 1.6 GHz cycles. 512-KB L2 cache is chosen for the main processor because we run small inputs for the applications.

memory processor that is integrated in the DRAM, following the microarchitecture of Figure 3. Table 2 shows the parameters used for each component of the architecture. The architecture is modeled cycle by cycle, including contention effects.

In the simulation, both the application thread and the prefetching thread are run simultaneously. We model the contention between the two threads on memory subsystems that are shared (memory controller, DRAM channels, DRAM banks, etc.). The simulation includes all overheads incurred by running the two threads on different processors.

Algorithm Parameters. Table 3 shows the default parameter values that we use for the algorithms described in Section 3.2. For the *Base* algorithm, we use the values similar to what Joseph and Grunwald use for their system [5] to make the comparison easier. For all the algorithms, we use *NumRows* = 64K, which results in a table of size 1.3 MBytes, 0.66 MBytes, and 1.8 MBytes for *Base*, *Chain*, and *Repl*, respectively. These sizes are very tolerable, since the table is a plain software data structure that is stored in main memory, is dynamically allocated, and is cached by the memory processor.

The conventional prefetching discussed in Section 3.4 takes two parameters: the number of streams it is able to prefetch simultaneously (*NumSeq*) and the number of prefetches that it issues per miss in a sequence observed (*NumPref*). We implement this algorithm in hardware in the L1 cache (*Conven4*) and also in software running on the memory processor (*Seq1* and *Seq4*).

Algorithm	Label	Parameter Values
[illegible]	*Base*	*NumSucc* = 4, *Assoc* = 4
Chain	*Chain*	*NumSucc* = 2, *Assoc* = 2, *NumLevels* = 3
Replicated	*Repl*	*NumSucc* = 2, *Assoc* = 2, *NumLevels* = 3
Conventional 1-Stream	*Seq1*	*NumSeq* = 1, *NumPref* = 6
Conventional 4-Stream	*Seq4*	*NumSeq* = 4, *NumPref* = 6
Conventional 4-Stream	*Conven4*	*NumSeq* = 4, *NumPref* = 6

Table 3: Parameter values used in the algorithms.

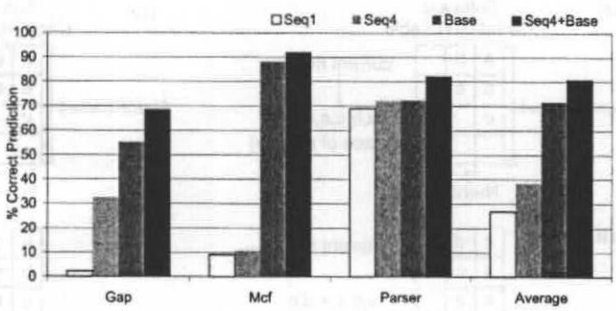


Figure 5: Characterizing the predictability of misses.

5 Evaluation

To evaluate our prefetching scheme, we first characterize the behavior of applications (Section 5.1) and then compare the performance of different algorithms (Section 5.2).

5.1 Characterizing Application Behavior

For memory-side correlation prefetching to be effective, the miss address streams have to be predictable. In this experiment, we record the fraction of L2 cache misses that are correctly predicted. For a sequential scheme, this means that the upcoming address exactly matches the one predicted, while for a pair-based scheme, the upcoming address matches one of the predicted successors. The thread does not perform prefetching here and it only observes the addresses of all L2 cache misses.

In our experiments, shown in Figure 5, we record the fraction of L2 cache misses that are correctly predicted. We try stride-based schemes that detect up to one stream (*Seq1*) and four streams (*Seq4*), the *Base* algorithm, and the combination.

The figure shows that the miss stream is largely predictable, with *Seq4*, *Base*, and *Seq4+Base* correctly predicting roughly 40%, 70%, and 80% of the misses on average, respectively. However, the predictability of each application differs. For example, *Mcf* does not have sequential patterns, while *Parser* has mostly sequential patterns, and *Gap* is mixed.

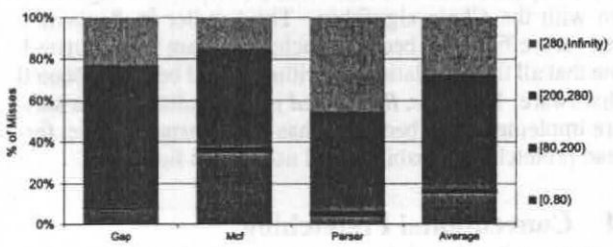


Figure 6: Characterizing the time between consecutive misses.

Seq4 always outperform *Seq1*, indicating that multiple

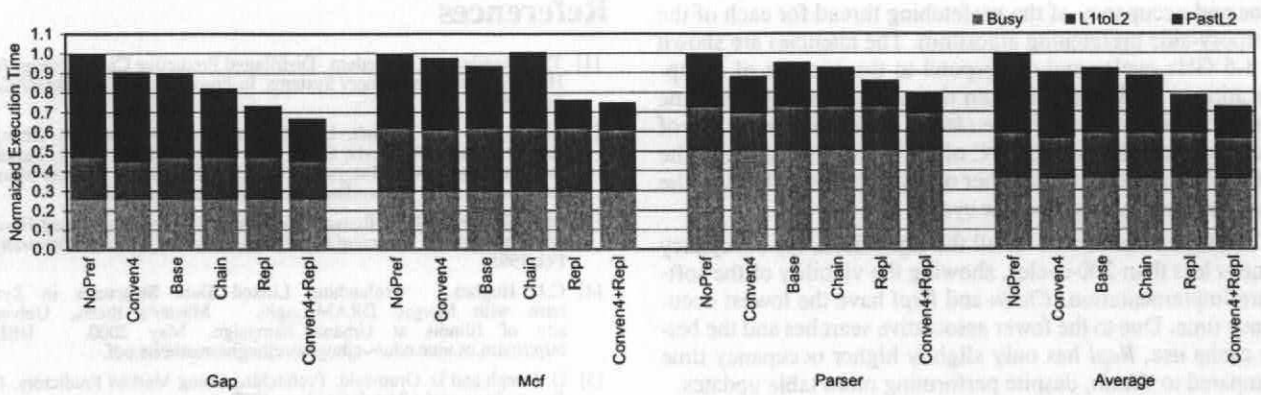


Figure 7: Execution time of the different algorithms.

stream support is necessary for a sequential scheme. The figure shows that in all applications, *Base* is almost as good as the combination *Seq4+Base*. This is because a correlation table is able to detect both sequential and irregular patterns, as long as the patterns repeat. Once the table learns a pattern, it can predict it effectively. However, it is still beneficial to have a multi-stream sequential prefetcher at the processor-side for several reasons: it does not need learning, it can be cheaply implemented, and it can hide the full memory latency if integrated with the L1 cache. Furthermore, it splits the misses into regular and irregular streams, and by tackling the regular one, it removes some load from the memory prefetcher.

We now consider the time between misses. Figure 6 classifies the misses according to the number of cycles between two consecutive misses arriving at the memory. The misses are grouped in bins corresponding to [0,80) 1.6 GHz processor cycles, [80,200), etc. The most significant bins in the figure are [200,280), [280,∞), and [0,80), which contribute on average to 54%, 28%, and 18% of all miss distances. The misses with distances between 200 and 280 are critical as they are both frequent and hard to hide even with out-of-order processors. Furthermore, since the round-trip memory latency is between 208 and 243 cycles, dependent misses are likely to fall in this bin. This characterization suggests that, to be on the safe side, occupancy time of the prefetching algorithm should be less than 200 cycles.

The [0,80) bin contains misses that may not give enough time for our prefetching thread to respond. Fortunately, these misses are not frequent and are likely to be overlapped with each other or with computation. Thus, they harm the performance much less than the bin size implies.

5.2 Comparing the Different Algorithms

Figure 7 compares the execution time of the applications in different cases: no prefetching (*NoPref*), hardware processor-side L1 prefetching as shown in Table 3 (*Conven4*), different software memory-side prefetching schemes as shown in Table 3 (*Base*, *Chain*, and *Repl*), and the combination of *Conven4* and *Repl* (*Conven4+Repl*). For each application and the average, the bars are normalized to *NoPref*. They are broken down into miss stall time past the L2 cache (*PastL2*), miss stall time between the L1 and L2 caches (*L1toL2*), and the remaining time (*Busy*) that represents processor computation plus various pipeline stalls.

On average, the *PastL2* time is the most significant component of the execution time, contributing about 40%, while *Busy* and *L1toL2* follow with 35% and 25%, respectively. Thus, although our software scheme can only target L2 cache misses, we are targeting the main performance bottleneck.

The conventional scheme (*Conven4*) performs well on applications with some sequential patterns, such as *Gap* and *Parser*, but is ineffective in the application that has purely irregular patterns (*Mcf*). On average, *Conven4* reduces the execution time by 10%.

The pair-based schemes show mixed performance. The *Base* scheme, modeled after Joseph and Grunwald's, shows limited speedups because it does not prefetch far enough. *Chain* performs slightly better than *Base*, but is limited by inaccuracy and high response time. *Repl* is able to reduce the execution time significantly. It outperforms both *Base* and *Chain* in all applications. Its impact comes from the nice properties of the *Replicated* algorithm, as discussed in Section 3.

The combined scheme (*Conven4+Repl*) performs the best. Its impact is significant: it removes on average 60% of *PastL2* stall time, providing an average speedup of 1.36. Compared to processor-side prefetching only (*Conven4*) with an average speedup of 1.11, and memory-side prefetching only (*Repl*) with an average speedup of 1.28, there is a clear synergistic effect in the combined scheme. Memory-side prefetching helps processor-side prefetching in irregular patterns, while processor-side prefetching helps in regular patterns.

Workload of the Prefetching Thread

We can gain further insight by examining the work load of the prefetching thread. Figure 8 shows the average response

time and occupancy of the prefetching thread for each of the memory-side prefetching algorithm. The latencies are shown in 1.6 GHz cycles and correspond to the average of all applications. Each bar is broken down into computation time (*Busy*) and memory stall time (*Mem*). The numbers on top of each bar show the average IPC of the prefetching thread. The IPC is calculated as the number of instructions divided by the number of memory processor cycles.

The figure shows that for all the algorithms, the occupancy time is less than 200 cycles, showing the viability of the software implementation. *Chain* and *Repl* have the lowest occupancy time. Due to the fewer associative searches and the better cache use, *Repl* has only slightly higher occupancy time compared to *Chain*, despite performing more table updates.

The response time is very important for prefetching effectiveness. The figure shows that *Repl* has the lowest response time. its value is around 30 cycles.

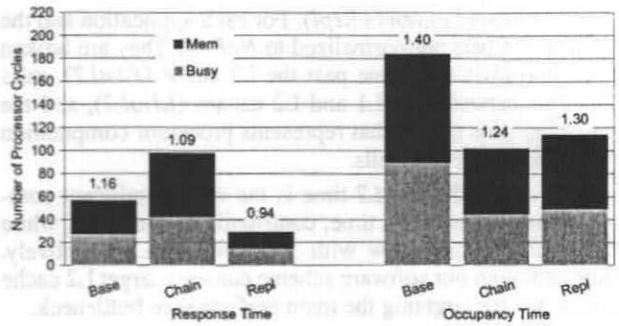


Figure 8: Response and occupancy time of the prefetching thread for each of the prefetching algorithm.

6 Conclusions

This paper introduced memory-side correlation-based prefetching implemented in a user-level thread. The scheme runs on a general-purpose processor in the main memory. The scheme can be supported with few modifications to the L2 cache and no modification to the main processor. We introduced a new organization of the correlation table and a new correlation prefetching algorithm that enable fast and accurate far-ahead prefetching with high coverage. Overall, our scheme effectively prefetched irregular applications, speeding up three SPECInt2000 applications by an average of 1.28. Furthermore, our scheme can work synergistically with a conventional processor-side prefetcher to deliver an average speedup of 1.36.

Acknowledgement

We thank James Tuck, Jose F. Martinez, Jose Renau, and Michael Huang for contributions to this work.

References

[1] T. Alexander and G. Kedem. Distributed Predictive Cache Design for High Performance Memory Systems. In *Proceedings of the 2nd HPCA*, Feb 1996.

[2] J.B. Carter, W.C. Hsieh, L.B. Stoller, M.R. Swanson, L. Zhang, E.L. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M.A. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a Smarter Memory Controller. In *Proceedings the 5th HPCA*, January 1999.

[3] M.J. Charney and A.P.Reeves. Generalized Correlation Based Hardware Prefetching. *Technical Report EE-CEG-95-1, Cornell University*, Feb 1995.

[4] C.J. Hughes. Prefetching Linked Data Structures in Systems with Merged DRAM-Logic. Master's thesis, University of Illinois at Urbana-Champaign, May 2000. URL: http://rsim.cs.uiuc.edu/~cjhughes/cjhughesmsthesis.pdf.

[5] D. Joseph and D. Grunwald. Prefetching Using Markov Predictors. In *Proceedings of the 24th ISCA*, June 1997.

[6] N.P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th ISCA*, pages 388–397, 1990.

[7] D. Koufaty and J. Torrellas. Comparing Data Forwarding and Prefetching for Communication-Induced Misses in Shared-Memory MPs. In *Proceedings of the ICS*, July 1998.

[8] C. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanovic, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick. Scalable Processors in the Billion-Transistor Era: IRAM. *IEEE Computer*, September 1997.

[9] V. Krishnan and J. Torrellas. An Execution-Driven Framework for Fast and Accurate Simulation of Superscalar Processors. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 1998.

[10] D. Kroft. Lockup-free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of the 8th ISCA*, pages 87–85, 1981.

[11] A.-C. Lai, C. Fide, and B. Falsafi. Dead-Block Prediction and Dead-Block Correlating Prefetchers. In *Proceedings of the 28th ISCA*, 2001.

[12] C.-K. Luk. Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors. In *Proceedings of the 28th ISCA*, 2001.

[13] NVIDIA. http://www.nvidia.com.

[14] R. Cooksey, D. Colarelli, and D. Grunwald. Content-based Prefetching: Initial Results. In *The 2nd Workshop on Intelligent Memory Systems*, Nov 2000.

[15] A. Roth and G.S. Sohi. Speculative Data-Driven Multithreading. In *Proceedings of the 7th HPCA*, pages 37–48, Jan 2001.

[16] Sony Computer Entertainment Inc. http://www.sony.com.

[17] T. Sherwood, S. Sair, and B. Calder. Predictor-Directed Stream Buffers. In *Proceedings of the 33th MICRO*, Dec 2000.

[18] C.-L. Yang and A.R.Lebeck. Push vs. Pull: Data Movement for Linked Data Structures. In *Proceedings of the 2000 ICS*, May 2000.

Multi-Threading For Latency

John P. Shen

It looks like multithreading is here to stay with the recent introduction of the "Hyper-Threading" technology by Intel for Xeon family server processors. This talk will start with Intel's Hyper-Threading technology and move on to cover ongoing research activities in Intel Labs related to Hyper-Threading. More specifically, the focus is on Speculative Precomputation techniques for both Itanium and IA32 machines, and how to leverage multithreading resources to achieve better latency for single-threaded applications. This talk concludes by suggesting a paradigm of leveraging TLP to achieve better MLP in order to realize a new form of ILP.

Bio:

John P. Shen is the Director of the Microarchitecture Lab at Intel, whose mission is to develop new and innovative microarchitecture techniques for possible adoption by future Itanium and IA32 microprocessor products. This lab has researchers located in Santa Clara CA, Hillsboro OR, and Austin TX, and works closely with product groups. Prior to joining Intel in 2000, for over 18 years John Shen was on the faculty of the Electrical and Computer Engineering Department of Carnegie Mellon University, where he headed up the Carnegie Mellon Microarchitecture Research Team (CMuART). He is currently writing a textbook on "Fundamentals of Superscalar Processor Design" which will be published by McGraw-Hill in 2002.