IMPLICITLY-MULTITHREADED PROCESSORS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Il Park

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2003

ACKNOWLEDGMENTS

Vijay's policy for a student to earn a Ph.D. is for the student to learn something that he can use during his lifetime. I have learned from Vijay how to find, analyze, and solve problems. However, the best gift I got from Vijay is something I had lost for a long time since I left my undergraduate school, which is "being confident." I dare say I finally found someone I really respect from the bottom of my heart as an advisor and as a friend.

As I stand at a turning point, I want to thank my family for their unconditional support through the whole process. My mother has been my best friend and my mental supporter. My grandmother has been my biggest fan with endless love. Now, it is my turn to give my family some response.

I also want to thank my whole architecture group at Purdue. Chong has been my closest friend, and we went through happy and painful times together. Chen-Yong's wide knowledge always excites me. Mike's deep knowledge about the circuit area has taught me a lot along with his calmness. I also want to give Chad my special thanks for giving me a hand with the writing process and providing lots of fun in our school lives. An-Chow always has been a man to offer his kindness and deep knowledge with no hesitation. I also express my appreciation to Jahangir, Zeshan, Ankit, Kailash, Jin-Yi, Ethan, and Yen for their friendship and influence on me.

Finally, I would like to thank my friend Dong-Ki and his wife for their friendship. I cannot imagine how my life looks like without him and her. When I suffered most, they helped me most. I believe that when I am most happy, they will be too.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABSTRACT

Park, Il. Ph.D., Purdue University, August, 2003. Implicitly-Multithreaded Processors. Major Professor: T. N. Vijaykumar.

Simultaneous Multithreading (SMT) is proposed to improve pipeline throughput by overlapping execution of multiple threads. However, SMT cannot improve single-thread performance. To improve single-thread performance, I propose the *Implicitly-Multi-Threaded* (IMT) *architecture* to execute compiler-specified speculative threads on to a modified SMT pipeline. IMT reduces hardware complexity by relying on the compiler to select suitable thread spawning points and to orchestrate inter-thread register communication. This study shows that a naive mapping of even optimized compiler-specified threads onto SMT performs only comparably to an aggressive superscalar; a naive IMT (N-IMT) inefficiently shares SMT's resources among threads irrespective of resource availability, thread resource usage, and inter-thread dependence. Optimized IMT (O-IMT) proposes key microarchitectural optimizations to alleviate these inefficiencies in N-IMT.

I propose three primary optimizations and two secondary optimizations. The three primary optimizations are: (1) resource- and dependence-based fetch policy to fetch and execute suitable instructions, (2) context multiplexing to improve utilization and map as many threads to a single context as allowed by availability of resources, and (3) early thread-invocation to hide thread start-up overhead by overlapping one thread's invocation with other threads' execution. Two secondary optimizations are: (1) speculatively releasing register values to avoid the implementation and performance issues of N-IMT's thread-level squashing and (2) two-phase commit to reduce register pressure by freeing some registers at instruction commit, before the thread commits.

Using SPEC2K benchmarks and execution-driven simulation, this study shows the performance comparison among an aggressive superscalar, N-IMT, O-IMT, previously-proposed Threaded Multipath Execution (TME), and Dynamically MultiThreaded (DMT) Processors. The results indicate that N-IMT outperforms DMT, but outperforms neither an aggressive superscalar nor TME. With three primary microarchitectural mechanisms, O-IMT improves performance by considerable speed-up over an aggressive superscalar and

TME. Even though two secondary optimizations do not increase the O-IMT's speed-up significantly on average, they significantly improve some specific benchmarks' performance.

1 INTRODUCTION

As CMOS technology continually improves, chips are able to hold more and more transistors. However, wire delays have been failed to scale with transistors, and this trend will continue into the future as long as we will use CMOS technology. Unfortunately, there are lots of programs that inherently do not have enough *instruction-level parallelism* (ILP) to exploit the higher number of transistors and to hide wire delays with useful work. For instance, more than half of SPEC 2K benchmarks do not have enough ILP to exploit all of the on-chip transistors available today.

Architects are now exploring *thread-level parallelism* (TLP) to exploit the continuing improvements in CMOS technology to deliver higher performance. Chip MultiProcessor (CMP) is proposed to mitigate wire delays within a chip and to improve overall throughput by running multiple (either multiprogrammed or explicitly parallel) threads simultaneously on multiple cores in a chip [27,16]. Simultaneous multithreading (SMT) [29] is proposed to improve pipeline throughput by overlapping multiple (either multiprogrammed or explicitly parallel) threads on a single wide-issue superscalar. The proposed Alpha 21464, the recently-announced IBM Power5, and the hyperthreaded Pentium 4 currently in production [15] are examples of SMT processors. However, these architectures do not speed up a sequential program when the program is not explicitly threaded. Unfortunately, it is not easy for a compiler to generate explicitly parallelized threads from a single sequential program. For instance, a compiler cannot easily handle indirect memory accesses and indirect calls, and most C programs have these problems.

Speculative threading has been a popular approach to speed up sequential programs. Previous proposals use software [12,23,19,13,24,25,8] or hardware [1,17] to peel off potentially-dependent threads from a single sequential program. While speculative threading executes potentially-dependent threads speculatively, it uses hardware to enforce the sequential execution semantics later but before commit. However, the majority of these proposals use CMP-based hardware platforms to support speculative threading [12,23,19,13,24,25,8,17]. Unlike SMT, which is a centralized architecture that has a single core on a chip, CMP is a distributed architecture with multiple cores on a chip. Because of this architectural characteristic of CMP's, the previous proposals that uses CMP require special customized hardware to support register communication and memory disambiguation across separate processing cores or pipelines. The difficult design requirement of such extra hardware greatly reduces the chance of realizing speculative threading proposals into commercial products.

Recently, researchers have also advocated using SMT's multithreading support to improve the execution time of a single sequential program. Examples include Threaded Multipath Execution (TME) [31] and Dynamically MultiThreaded (DMT) processors [1].

I propose the *Implicitly-MultiThreaded* (IMT) processor to utilize SMT's architectural support for multithreading by executing speculative threads extracted from a sequential program. IMT executes compiler-specified speculative threads from a sequential program on a wide-issue SMT pipeline. IMT is based on the fundamental observation that Multiscalar's execution model — compiler-specified speculative threads [10,23] — can be decoupled from the processor organization — distributed processing cores. Multiscalar employs sophisticated specialized hardware, the Register Ring [4] and Address Resolution Buffer [11], which are strongly coupled to the distributed organization. In contrast, IMT proposes to map speculative threads onto generic SMT hardware.

IMT goes one step further by proposing a novel microarchitectural optimization to support multithreading even in one context at SMT. A context at SMT is defined as a conceptual hardware bundle to which a single thread runs. This optimization greatly alleviates the load imbalance, which is one of the biggest overhead of speculative threading, and it also enables us to use even genuine superscalar to run speculative threads (i.e., Multiscalar's compiler-generated speculative threads) with only minor hardware modification.

IMT differs in many key respects from prior proposals, such as TME, DMT, and slice-based precomputation for speculative threading on SMT. TME and slice-based precomputation do map their execution onto SMT, but they execute multiple threads in only the infrequent cases of branch mispredictions and cache misses. In contrast, IMT invokes threads in even the common cases of correct branch predictions and cache hits, better utilizing SMT resources. Rather than using compiler-specified threads as in IMT, DMT creates threads in hardware during run-time. Because of the lack of compile-time information and flexibility, DMT's threads frequently incur dependence stalls that prohibit them from extracting thread-level parallelism effectively. In addition, DMT's threads are inordinately long, requiring fast, frequent searches through thousands of instructions held in custom trace buffers that are difficult to implement efficiently. None of these proposals consider any optimization for alleviating the load imbalance in order to maximize the context resource usage. Rather these proposals take contexts for granted.

IMT modestly modifies SMT to perform the traditional tasks of fetch, register rename, and memory dependence enforcement for speculative threads. IMT invokes threads in program order but fetches instructions out of program order by interleaving earlier and later threads. Out-of-order fetch allows independent instructions in later threads to enter the pipeline early and overlap with the processing of earlier threads' dependent instructions. By appropriately setting up the rename tables, IMT forces later threads' instructions, which are register dependent on earlier threads' yet-to-be-fetched instructions, to wait until the earlier instructions execute. Because IMT needs to enforce memory dependences across its threads, loads and stores from one thread search other threads' loads and stores in the load/store queue to enforce inter-thread memory dependences.

Unfortunately, a naive mapping of compiler-specified speculative threads onto SMT performs poorly. Despite using an advanced compiler [30] to generate threads, a naive IMT (N-IMT) implementation performs only comparably to an aggressive superscalar. N-IMT's key shortcoming is its indiscriminate approach to fetching/executing instructions from threads without accounting for resource availability, thread resource usage, and inter-thread dependence information. The resulting poor utilization of pipeline resources (e.g., issue queue, load/store queues, and register file) in N-IMT negates the advantages of speculative threading.

This dissertation identifies the key inefficiencies in N-IMT and proposes optimized IMT (O-IMT), which has three primary and two secondary microarchitectural optimizations necessary to alleviate the inefficiencies of N-IMT. The three primary optimizations are:

- **Novel fetch policy:** Because the choice of which thread to fetch from every cycle fundamentally impacts performance, IMT carefully controls fetch via a resource- and dependence-based thread fetch policy. The policy employs a highly accurate (~97%) *dynamic resource predictor (DRP)* to gauge dynamic resource (physical registers, load/store queue slots, and active list entries) availability to avoid thread squashes due to lack of resources midway through execution. The policy also employs *inter-thread dependence heuristic* (*ITDH*) to avoid the delay of earlier threads' instructions in favor of fetching and front-end processing later threads that are data-dependent on earlier threads anyway. In contrast, Multiscalar statically partitions its resources and fetches from as many threads as the number of cores. TME, DMT, and N-IMT use variations of ICOUNT [28] or round-robin fetch policies that do not account for resource availability and result in suboptimal performance.

- **Multiplexing hardware contexts to bring more suitable instructions**: As in TME and DMT, N-IMT assigns a single thread to each SMT context [28] consisting of an active list and a load/store queue. Because many programs have short-running threads and SMT implementations are likely to have only a few (e.g., 2-8) contexts, such an assignment severely limits the number of instructions in flight. Unfortunately, a brute-force increase in thread size would result in an increase in misspeculation frequency and in the number of instructions discarded per misspeculation [30]. To obviate the need for larger threads, O-IMT multiplexes the hardware contexts by mapping as many contiguous threads onto a single context as allowed by the resources. While others have alluded to overlapping one thread's wait-to-commit time with another's execution [24,8], multiplexing overlaps multiple threads by *simultaneously* executing them.
- **Hiding thread start-up delay to increase overlap among suitable instructions**: Speculatively-threaded processors incur the delay of setting up register rename tables at thread start-up to ensure proper register value communication between earlier threads and a newly-invoked thread. As in TME, N-IMT incurs extra start-up delay prior to thread invocation. Because the compiler-specified inter-thread register dependence information is available well before the thread starts, O-IMT hides the delay by overlapping rename table set-up with previous thread execution. Other proposals for speculative threading, including DMT and Multiscalar, do not address this issue.

The two secondary optimizations are:

- **Speculative releasing:** Upon a branch misprediction within a thread, IMT does thread-level squashing, which squashes all subsequent instructions only within the thread and not later threads, saving later instructions. Much like Multiscalar, N-IMT disallows communication of speculative register values across threads, delaying values until intra-thread speculation is resolved [3]. While this strategy enables thread-level squashing by guaranteeing that an intra-thread squash does not affect later threads, it causes considerable performance loss by delaying values and an implementation issue of releasing values from instructions that are not in the pipeline anymore. O-IMT avoids the performance loss and implementation issue by speculatively communicating values.
- **Reducing Register pressure**: Out-of-order fetch, employed by IMT and others, overlaps instructions farther than in-order fetch, increasing physical register pressure. O-IMT employs a two-phase commit strategy in which an instruction commits within its thread freeing instruction resources, and threads commit in global order freeing thread resources. Two-phase commit alleviates register pressure by freeing some registers at

instruction commit, *before* the thread commits. Multiscalar and TME do not address this issue. DMT reduces resource pressure by employing prohibitively large custom instruction trace buffers (holding thousands of instructions and all register and memory data for them) and retiring instructions from the pipeline and active list speculatively. In the case of any misspeculation, DMT searches the trace buffer and selectively rolls back all relevant data. Unfortunately, frequent associative searches through such large buffers are slow and impractical.

Using the SPEC2000 benchmarks, results show that N-IMT actually degrades performance in integer benchmarks by 3% on average, and it improves performance negligibly in floating-point benchmarks relative to a comparable baseline superscalar with comparable hardware resources. In contrast, O-IMT achieves average speedups of 20% and 29% in the integer and floating-point benchmarks, respectively, over superscalar. The results also indicate that TME and DMT are on average not competitive relative to a comparable superscalar.

While the techniques I propose in this dissertation are essential for IMT to perform well, they can also help improve performance in distributed microarchitectures such as [23]. However, I focus on SMT platform in this study.

The rest of this dissertation is organized as follows. Chapter 2 shows the background of this research. Chapter 3 briefly explains compiler-specified threads. Chapter 4 describes implementing N-IMT on SMT and Chapter 5 proposes key microarchitecture optimizations to alleviate the inefficiency of N-IMT. Chapter 6 presents experimental results and Chapter 7 derives conclusions. Chapter 8 discusses the future direction of the research.

2 BACKGROUND

Simultaneous Multithreading (SMT) [29] has been proposed to improve pipeline throughput by overlapping execution of multiple (either multiprogrammed or explicitly parallel) threads on a single wide-issue processor. Figure 1 compares SMT with superscalar and Multithreaded processors [2]. SMT can issue multiple instructions from multiple threads each cycle, which is different from Multithreaded processors that execute instructions from one thread on a given cycle even though processors have hardware states for multiple threads at the same time.

Threaded Multiple Path Execution (TME) [31], Dynamic Multithreading (DMT) [1], slice-based recomputation [32,22], and Simultaneous Subordinate Microthreading (SSMT) [6] are earlier proposals to improve the performance of single application on SMT hardware. These machines utilize SMT's multithreaded hardware only for problematic but infrequent cases of branch mispredictions and/or cache misses, only when there are spare contexts to use. TME fetches instructions from both paths of hard-to-predict branches. When TME spawns a new thread, it incurs extra cycles to set up the rename tables, and employs an extra dedicated bus for a bus-based write-through scheme to copy rename maps. Slice-based recomputation and SSMT use helper threads (speculative slice

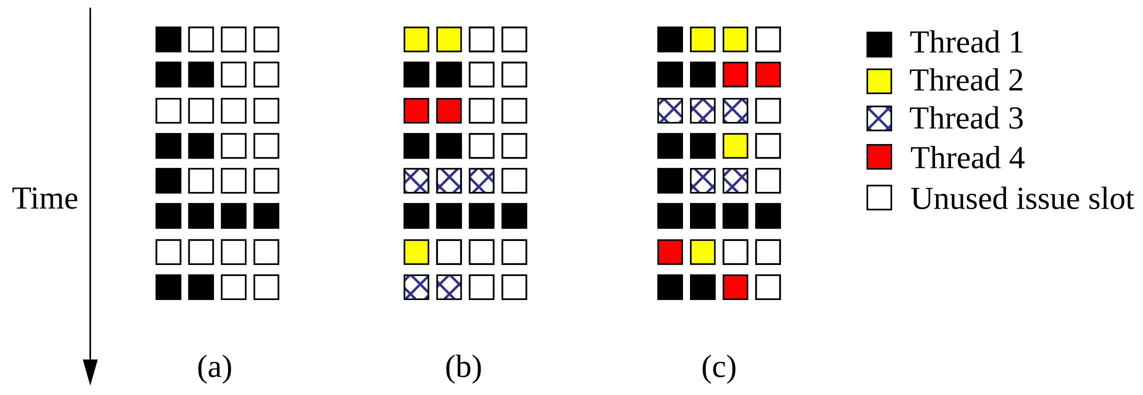


Fig. 1. Utilizing issue bandwidth: (a) superscalar, (b) multithreaded processor, (c) SMT

or micro instructions) designed for problem instructions, which are in the critical path and cause delay due to cache misses or branch mispredictions. Executing helper threads before executing those problem instructions from the main threads will bring the effect of prefetching and give more information for branch predictions in the main threads.

Another proposal, Dynamic Multithreading (DMT) also uses SMT as the underlying architecture [1]. To handle inter-thread dependences, DMT resorts to value prediction by using substantially more hardware for an aggressive copying mechanism to set up rename tables magically within a cycle, an entire extra pipeline for selective recovery from mis-speculations, and a large trace buffer to hold thousands of instructions in flight.

There are other proposals to execute speculative threads on distributed architectures such as Multiscalar [12,23], Multiplex [19], Hydra [13], Stampede [24,25], Speculative NUMA [8], SUN Microsystems MAJC [27], and others [17,16]. These proposals employ Chip MultiProcessor (CMP) as the underlying architecture. As a result, these architectures have multiple processor cores in a single chip, and each core has dedicated hardware resources including the pipeline.

While CMP statically partitions and allocates all hardware resources, such as fetch unit (including fetch bandwidth and caches) and execution unit (including functional units and physical registers), SMT shares them for all threads in flight. So, under the assumption of the same number of transistors used, SMT may have higher Instructions Per Cycle (IPC) than SMP does, a distributed architecture SMP has a clock-speed advantage over the centralized architecture SMT.

3 COMPILER-SPECIFIED SPECULATIVE THREADING

Speculatively threaded architectures may use the hardware [1,17] or compiler [23,13,25] to partition a sequential program into multiple implicit threads. IMT uses Multiscalar's compiler-specified speculative threads. The Multiscalar compiler employs several heuristics to optimize thread selection [30]. The compiler forms reasonably-sized threads without exceeding the number of targets emanating from a thread. To the extent possible, the compiler exploits loop parallelism by capturing entire loop bodies into threads, avoids inter-thread control-flow mispredictions by enclosing both if and else paths of a branch within a thread, and reduces inter-thread register dependences. Typical threads contain 10-20 instructions in integer programs, and 30-100 instructions in floating-point programs. These instruction counts give an idea of the order of magnitude of resources needed and overheads incurred per thread, and help understand the optimizations introduced in this dissertation.

The compiler provides summary information of a thread's register and control-flow dependences in the *thread descriptor*. In the descriptor, the compiler identifies: (1) The set of registers live into the thread via the *use mask*, and the set of registers written in at least one of the control-flow paths through the thread via the *create mask*. (2) The possible control-flow exits out of the thread via the *targets*. The compiler also annotates the instructions to specify each instance of the dependence summarized in the descriptor. An instruction that is the last write to an architectural register in all the possible control flow paths is annotated with *forward* bits. Instructions that lead to a target are annotated with *stop* bits.

Figure 2 shows an example thread. The thread shown here has two targets: B1 and B5. The branch at the bottom of B4 is annotated with stop bits (shown by S). The create mask contains r1, r2, and r3. r2 is read before written in B3. r4, r5, and r6 are read and never written. Hence r2, r4, r5, and r6 are in the use mask. In B2 and B3, the writes to r3 are the last write in the both path B1B2B4 and B1B3B4, and these instructions are annotated with forward bits (shown by F). However, there are cases where forward bits are not sufficient. For instance, in the figure, the write to r1 in B1 is not the last write in the path B1B2B4 but

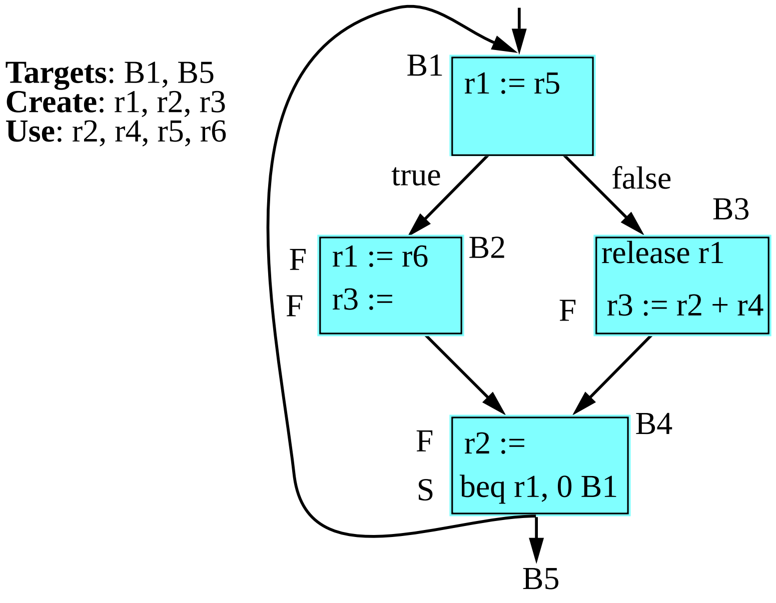


Fig. 2. Compiler-specified speculative threads: an example.

it is in the path B1B3B4. To handle this case, the compiler inserts a *release* instruction in B3. In Section 4.3, I explain how the hardware uses forward and release instructions to implement inter-thread register communication.

Typical threads in integer programs contain 10-20 instructions, and 30-100 instructions in floating point programs. These instruction counts give an idea of the order of magnitude of resources needed and overhead incurred per thread, and help understand the following sections.

In contrast to IMT, the prior proposals for speculative threading on SMT, DMT [1] and TME [31] create threads in hardware. DMT spawns a new thread when the fetch unit reaches a function call or a backward branch. The start address of the new thread is the address *after* the call or backward branch. DMT's threading has two weaknesses: First, DMT cannot exploit any parallelism across inner loop iterations, although the largest opportunity for exploiting parallelism resides in inner loop iterations. Second, the threads are inordinately long, of the order of thousands of instructions, and require large custom trace buffers to hold their speculative state. In TME, threads are simply created upon identifying an unpredictable branch. However, TME's opportunity for extracting thread-level parallelism is severely limited due to targeting the uncommon cases of unpredictable branches.

4 IMPLICITLY-MULTITHREADED PROCESSORS

This study proposes the *Implicitly-MultiThreaded* (IMT) architecture to utilize SMT's support for multithreading by executing compiler-specified speculative threads. Figure 3 illustrates how a single application can be partitioned into speculative threads by compiler and how those threads can be mapped into SMT's shared pipeline. IMT exploits *implicit* parallelism, as opposed to programmer-specified, *explicit* parallelism exploited by conventional SMT and multiprocessors. Like the Multiscalar architecture, IMT predicts and spawns the threads in program order with the help of compile time information, and it maps the threads to execution resources with the earliest thread as the *non-speculative* (head) thread, followed by subsequent *speculative* threads [23]. IMT leverages the conventional register renaming to honor the inter-thread control-flow and register dependences specified by the compiler. IMT uses the load/store queue (LSQ) to enforce inter-thread memory dependences. Upon completion, IMT commits the threads in program order.

SMT places instructions from all threads in a single *issue queue* in which instructions wait until source operands become available enabling out-of-order issue. As each instruction issues out of the issue queue, it stays in its thread's private *active list* and commits from the active list in the thread's program order, enabling precise interrupts. SMT conceptually bundles all the per-thread resources such as the active list, load/store queue and register renaming logic into a *hardware context*, and allows as many threads as there are contexts. SMT shares the functional units, physical registers, issue queue, and memory hierarchy among all the contexts.

Figure 4 depicts the anatomy of an IMT processor based on an SMT pipeline. IMT uses the rename tables for register renaming, the issue queue for out-of-order scheduling, per-context load/store queue for memory dependences, and the active list for instruction reordering prior to commit. As in SMT, IMT shares the functional units, physical registers, issue queue, and memory hierarchy among all contexts. This research presents two variations of IMT processors, mapping compiler-optimized threads [30] onto the SMT pipeline: (1) a naive IMT (N-IMT) that performs comparably to an aggressive superscalar, and

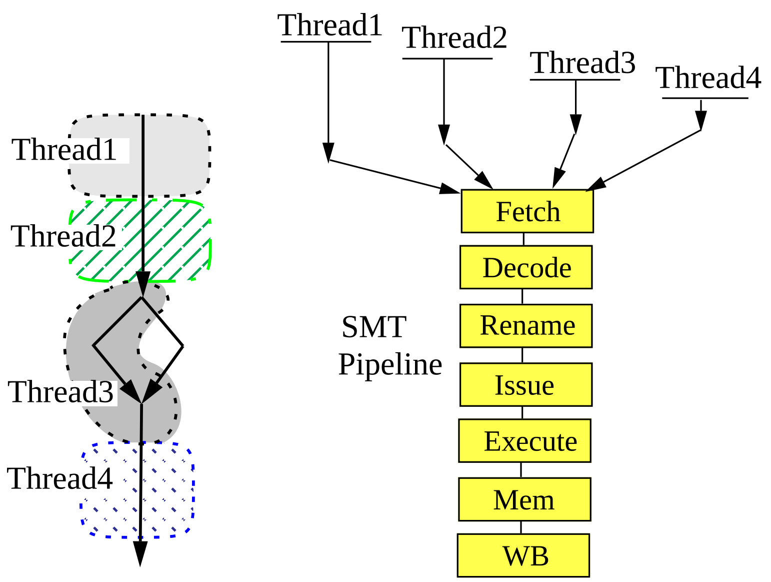


Fig. 3. The IMT concept.

(2) an optimized IMT (O-IMT) that uses novel microarchitectural techniques to significantly improve performance.

The rest of this chapter is organized as follows. I first explain the thread invocation in N-IMT. Then I show how N-IMT does register communication and memory disambiguation. Finally, I explain the thread execution and thread completion in N-IMT.

4.1 Thread Invocation

IMT invokes threads in program order by predicting the next thread from among the targets of the previous thread, using a thread predictor like Multiscalar. Using the predicted target number, IMT obtains the next thread's start PC from the previous thread's descriptor. Like Multiscalar, IMT caches thread descriptors in a descriptor cache. Although IMT invokes threads in program order, it fetches later threads' instructions out of order before fetching all of earlier threads' instructions, interleaving instructions from multiple threads. To decide which thread to fetch instructions from every cycle, IMT consults the fetch policy.

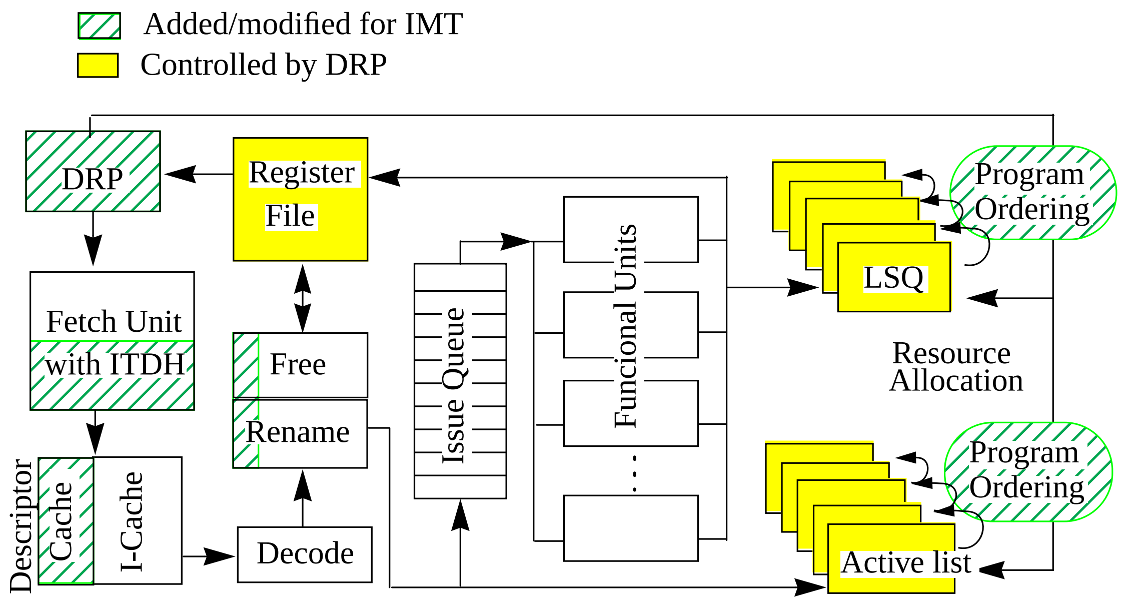


Fig. 4. The anatomy of an IMT processor.

4.1.1 Instruction fetch policy

The base IMT processor, N-IMT, uses an unmodified ICOUNT policy [28], in which the thread with the least number of instructions in decode, rename, and the issue queue is chosen to fetch instructions from every cycle. The rationale is that the thread that has the fewest instructions is the one whose instructions are flowing through the pipeline with the fewest stalls. By choosing the "best" thread each cycle, the fetch policy can minimize the issue queue clog and maximize the overall system throughput. Previous study examined diverse fetch policies and concluded that ICOUNT was the best among them [28].

However, unlike SMT's independent threads, IMT's threads are potentially dependent. This difference has a big implication to fetch policy, and it is expected that ICOUNT may not be the best fetch policy for IMT-like machines.

4.1.2 Mapping threads onto contexts

Multiscalar targets Chip MultiProcessor (CMP) as a base hardware platform. Multiscalar assigns one thread to each core in a chip. Therefore, Multiscalar simultaneously executes as many threads as the number of cores in a chip.

As mentioned before, SMT conceptually bundles all the per-thread resources such as the active list, load/store queue and register renaming logic into a hardware context, and allows as many threads as there are contexts. Therefore, the context of SMT has similar meaning as the core of CMP. N-IMT assigns one thread to a context, much like prior proposals including Multiscalar.

Previous study shows that the load imbalance is one of the biggest inefficiency (overhead) of speculative threading machines [19]. Because N-IMT maps threads to contexts in the same way Multiscalar maps threads to cores, it is possible to expect that N-IMT may suffer from the similar inefficiency. Section 5.1.2 will discuss the cause and effect of such inefficiency in detail and propose the microarchitectural optimization to alleviate the inefficiency.

4.2 Control Flow Speculation

In a conventional superscalar or SMT, the control flow is predicted at the instruction-level by a branch predictor that chooses one of two possible outcomes, namely taken or not-taken. Unlike SMT/superscalar, N-IMT has two different types of control flow predictions, which are instruction-level speculation and thread-level speculation. While IMT's instruction-level speculation is the same as SMT's control flow speculation, thread-level speculation is different from SMT's control speculation. Like Multiscalar implementation, N-IMT employs two different types of branch predictors [3]. A conventional branch predictor, called *intra-branch predictor,* is used for speculating the instruction-level control flow. A modified branch predictor, called *inter-branch predictor,* is used only for speculating the thread-level control flow.

In the inter-branch predictor, the next thread has to be predicted before reaching the end of the current thread. The thread generated by the Multiscalar compiler has N (> 2) possible control flow edges and therefore N successors. (The binaries I used for this research have four possible successors for each thread.) Therefore, the inter-branch predictor chooses from among multiple control flow targets, and the hardware has to be built to keep N targets for each control flow point. The inter-branch prediction table is shared across threads, and the predictor generates one outcome per cycle.

The intra-branch predictor is the same as the conventional branch predictor with one exception. Even though threads are invoked in-order and instructions are fetched in-order within a thread, threads are executed out of program order. As a result, instructions are fetched out of program order with respect to instructions from other threads. Therefore, IMT's global branch history cannot be managed in the same way as the conventional superscalar's. When a thread is activated (not invoked) for fetching, the branch history register from the inter-branch predictor is copied to the intra-branch predictor's global branch history register if the intra-branch predictor uses a global predictor. This copying warms up the global prediction within a thread and alleviates the disadvantage of a non-contiguous branch history due to out-of-order fetching. The intra-branch prediction table is shared across threads, and its branch history registers are maintained separately per thread. To utilize the fetching bandwidth fully, the intra-branch predictor is designed to make as many predictions as the number of ports in the L1 instruction cache.

4.2.1 Thread-level squash

As thread execution proceeds, the front-end predicts branches, as mentioned in Section 4.1. In N-IMT's threads, a branch could transfer control-flow to another thread (inter-thread control flow), or a non-sequential instruction within the thread (intra-thread control flow). Mispredictions of the two cases are not handled in the same manner. On an intra-thread branch misprediction, N-IMT selectively squashes only within the thread, keeping later threads in the pipeline. Using SMT's ability to squash instructions from a specific thread, N-IMT squashes only the thread's later instructions. Superscalar can achieve a similar effect by selectively squashing only incorrect instructions. By leveraging thread-level granularity, however, N-IMT's thread-level squashing is significantly simpler than superscalar's selective squashing [21] and does not require substantial hardware like other machines [1].

The main problem with internal squashing is that if incorrect register or memory values from an incorrect branch path would have been consumed by later threads, then later threads would have to be squashed even if they had been correctly predicted. To avoid such squashes, N-IMT disallows communication of speculative register values across threads, delaying values until intra-thread speculation is resolved, in the same way Multiscalar does [3]. This strategy enables thread-level squashing by guaranteeing that an intra-thread squash does not affect later threads. This thread-level squashing mechanism gives the advantage to N-IMT over superscalar by allowing N-IMT to avoid squashing instructions from later threads in flight in the event of intra-branch misprediction.

The inter-branch prediction is resolved when executing the last dynamic instruction of each thread. Upon resolving the last dynamic instruction's control flow in the current thread, N-IMT verifies the inter-thread prediction for the next thread and either allows the next thread to commit or squashes later threads on a thread misprediction. Section 4.6 discusses inter-thread branch mispredictions with other related activities.

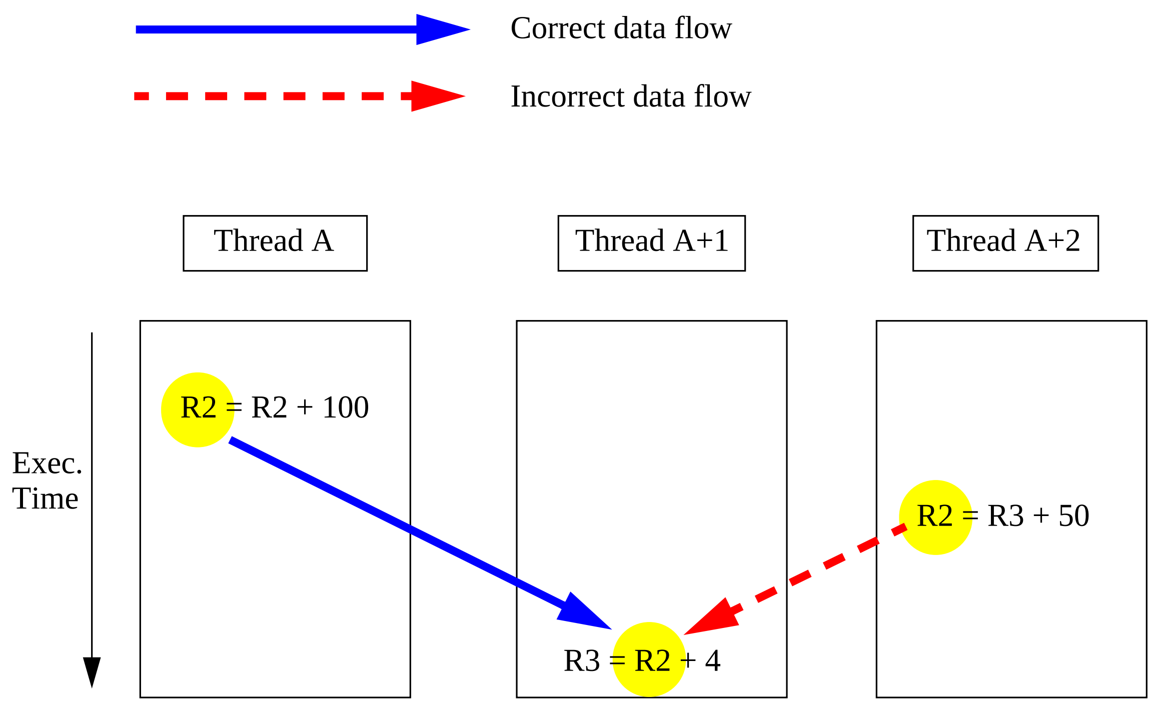
4.3 Register Communication through Renaming

SMT's register rename table links register value producers to consumers using the fact that SMT fetches instructions in program order. N-IMT's out-of-order fetch raises two issues in linking producers in earlier threads to consumers in later threads. First, N-IMT has to ensure that the rename maps for earlier threads' source registers are not clobbered by later threads. Second, N-IMT has to ensure that later threads' consumer instructions obtain the correct rename maps and wait for the yet-to-be fetched earlier threads' producer instructions.

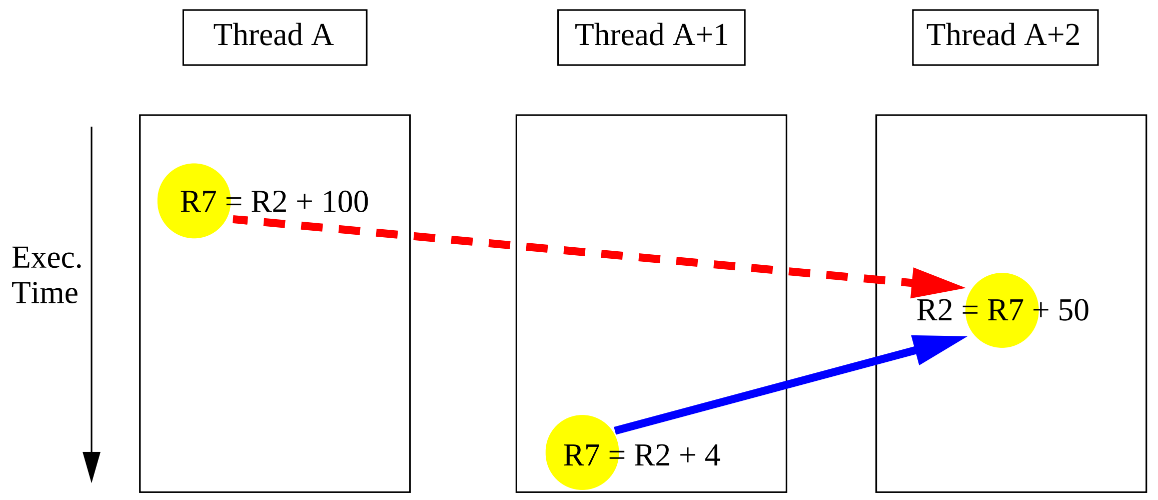
Figure 5 illustrates examples of these two issues for N-IMT's out-of-order fetch. In Figure 5 (a), the consumer *R2* in *thread A+1* should get the value from the producer *R2* in *thread A*. Because N-IMT fetches and renames the producer *R2* in *thread A+2* before it fetches and renames the consumer *R2* in *thread A+1*, the conventional renaming incorrectly links the consumer *R2* in thread *A+1* with the producer *R2* in *thread A+2*, not with the correct producer *R2* in *thread A*. In Figure 5 (b), the consumer *R7* in *thread A+2* should get the latest value from the producer *R7* in *thread A+1*. Because N-IMT fetches and renames the consumer *R7* in *thread A+2* before it fetches and renames the producer *R7* in *thread A+1*, the conventional renaming incorrectly links the consumer *R7* in thread *A+2* with the producer *R7* in *thread A*, not with the correct producer *R7* in *thread A*+1.

While others [1,17] employ hardware-intensive value prediction or complicated recovery to address these issues, N-IMT uses the create mask and use mask (Chapter 3), and existing rename tables. Although Multiscalar proposed the use of create mask to aid in inter-thread register communication, Multiscalar does not leverage conventional rename tables for this purpose. Figure 6 shows an example of how IMT handles rename map table when a thread is invoked.

For the first issue, before fetching any instruction from the next thread, N-IMT copies the rename maps, corresponding to the current (most recent) thread's use and create mask registers, from the *master rename table* to the current thread's *local rename table*. At this point, the master table reflects the program's register state up to the current thread beginning. The next thread modifies the master table, but not the current thread's local table. Later, when the current thread's instructions are fetched, they use the maps in the local table. In addition to use-mask registers' maps, the local table also copies the maps corresponding to create-mask registers from master rename table before the current thread updates the master rename table with any new mapping. Figure 6 shows that the current thread A copies maps not only for use-mask registers (R10:P102, R11:105, R12:P110,



(a) Earlier threads' source register map should not be clobbered by later threads.



(b) Later thread's consumer should wait for yet-to-be fetched earlier thread's producer.

Fig. 5. Issues of register renaming in out-of-order fetch machine.

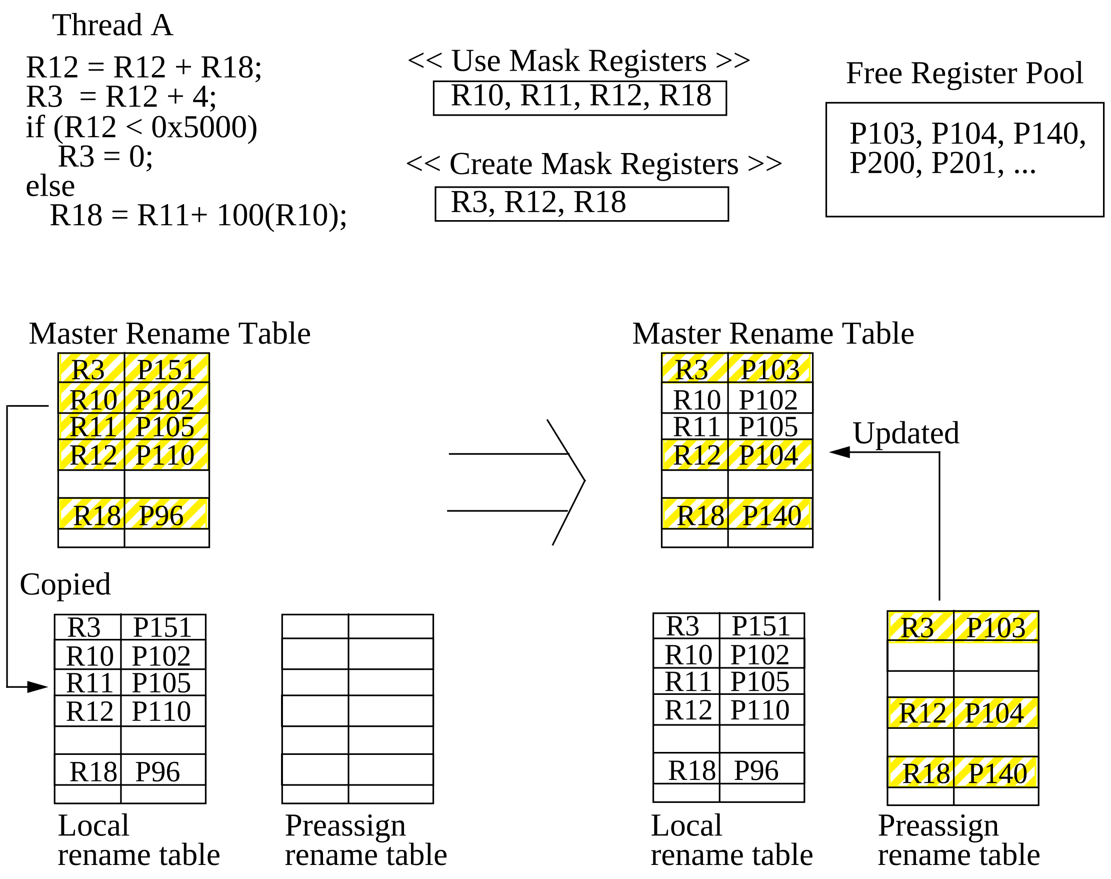


Fig. 6. Example of handling register rename map tables.

R18:P96), but also for create mask registers (R3:P151) from master table to local rename table. Conventional pipelines also perform such copying, on an instruction-by-instruction basis, to checkpoint rename maps for branch misprediction recovery. I explain the reason for copying the create-mask registers' maps at the end of this subsection.

For the second issue, a thread's create mask gives N-IMT a prior knowledge of the thread's yet-to-be fetched instructions. Upon invoking a thread, N-IMT *preallocates* and *preassigns* physical registers for all the create mask registers (e.g., map architectural create mask register R3 to preallocated physical register P103, register R12 to P104, and register R18 to P140 in Figure 6). N-IMT modifies the master table with the preassignments and marks the physical registers busy. Because N-IMT invokes threads in program order, the master table is updated in program order and provides the correct maps for later threads. Additionally, N-IMT allocates another *preassign rename table* and updates the table with the create mask's preassigned maps for later use. In Figure 6, the current thread A has new maps for create mask registers in the preassign rename table. Instructions use the local (and *not* master) table both to get their source rename maps and to put their destination rename maps. If an instruction's source is a use mask register, the local table provides the rename map for the register. An instruction that is data dependent on an earlier-thread instruction waits until the corresponding preassigned physical register is ready (or bypassed). Data-independent instructions proceed without waiting, much as in SMT. When the earlier-thread's forward or release eventually completes execution, its preassigned register gets the value, allowing all waiting instructions to proceed.

If an instruction is neither a forward nor release (Chapter 3), it does *not* use the preassigned physical register for its destination; instead, it newly allocates a physical register for the destination. A forward or release uses the preassigned physical register in the preassign table as its destination. Thereby, forwards and releases correctly bind the values live at the thread end to the preassigned registers. Forwards write their results in the preassigned physical registers. Releases copy values from the physical registers given by the local table to the preassigned physical registers. Thus, forwards and releases allow all waiting instructions to proceed. By copying the create mask maps at thread start-up from the master table to the local table, the local table holds the latest rename map for the create-mask registers irrespective of whether the thread actually writes to the create-mask registers or not. Therefore, releases copy the correct values by referring the local rename table when the thread actually does not write to the create-mask registers.

4.4 Memory Disambiguation through Load/Store Queue

N-IMT imposes program order in the load/store queues to enforce memory dependencies *within* and *across* threads. Memory accesses from a thread search its context's load/store queue (LSQ) to honor memory dependencies. If there is no match in the local load/store queue, accesses proceed to search other context's load/store queues. The non-speculative thread's loads do not search other contexts, but its stores search later contexts to identify and squash premature loads. Speculative threads' loads search in earlier contexts for previous matching stores, and stores search in later contexts for later premature loads. Thus, N-IMT uses the LSQ to achieve the same functionality as ARB's [11].

Figure 7 illustrates how to honor memory dependences through the LSQ. Dashed arrows are the N-IMT's extra search to other threads' LSQs. In this example, red arrow finds the premature load *load B2* in thread 2, and N-IMT squashes thread 2 and any following threads. Meanwhile, *load A1* from thread 1 needs to search previous threads' LSQ (thread 0 in this example) to find the latest store to the same address *A1*.

Unlike a load which is serviced out of program order, a store should be sent to the memory in program order when the store commits. The store queue entry will be cleared when the store commits. However, there is a gap between the time the store commits and the time it finishes updating the memory when the store is a cache miss in L1. This gap can vary with a range of a few cycles to hundreds of cycles. The store information will reside in the MSHRs (Miss Status Holding Registers) until the store finishes updating the memory. Therefore, all loads will get the latest store value from the MSHRs if there is any.

Multiple searches through LSQs from different contexts cause two design challenges. First, searching multiple LSQs to find the most recent store value requires extra cycles that impact load hit latency. Second, this searching also makes the hit latency variable because of the uncertainty of the latest store's existence and the port contention. For high performance, superscalar processors speculatively schedule instructions dependent on the load with the assumption that the load is a cache hit. Variable hit latencies may complicate such a scheduling mechanism. To avoid complicating the scheduler, N-IMT foregoes early-scheduling for the instructions that are dependent on the load.

However, there is one important exception to this rule. The most critical access is the load from the non-speculative thread (the head thread). This load does not need to search any previous segment because no previous segment exists. Therefore, the hit latency for this load is constant. Although the load from the non-speculative thread still has to search the MSHRs, this search also exists in the base case and it does not make the hit latency

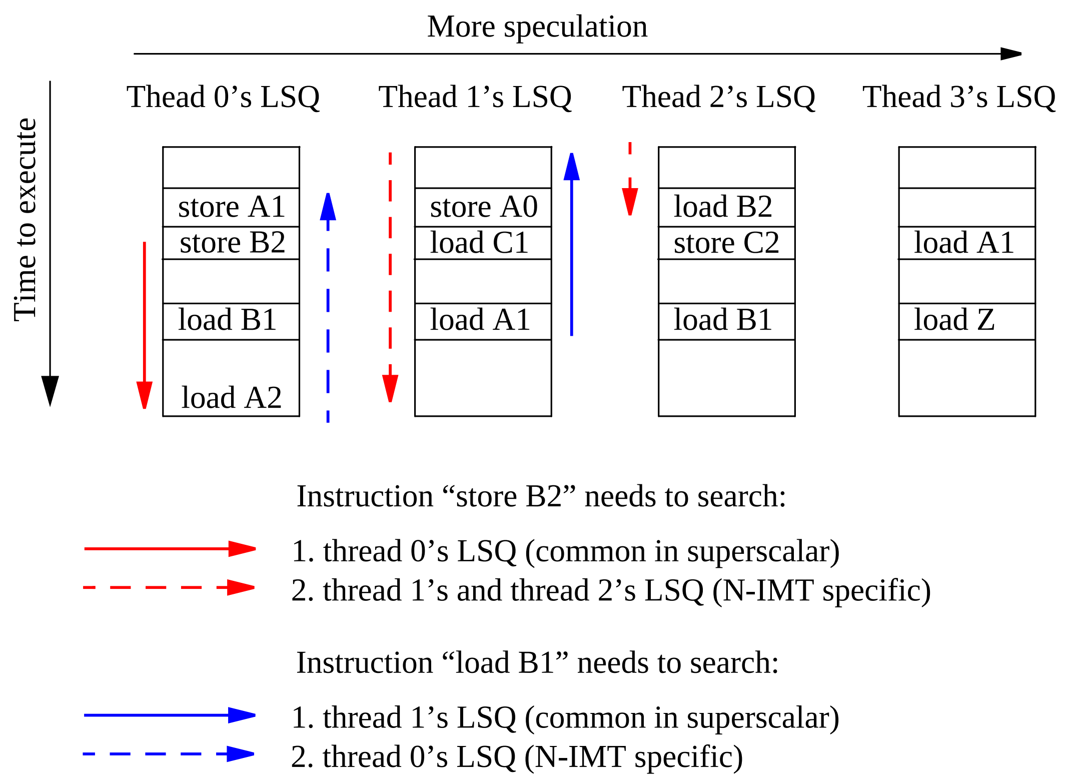

More speculation
Time to execute
Thead 0’s LSQ
Thead 1’s LSQ
Thead 2’s LSQ
Thead 3’s LSQ
store A1
store B2
load B1
load A2
store A0
load C1
load A1
load B2
store C2
load B1
load A1
load Z
Instruction “store B2” needs to search:
1. thread 0’s LSQ (common in superscalar)
2. thread 1’s and thread 2’s LSQ (N-IMT specific)
Instruction “load B1” needs to search:
1. thread 1’s LSQ (common in superscalar)
2. thread 0’s LSQ (N-IMT specific)

Fig. 7. Example of honoring memory dependences via load/store queue

variable. Therefore, N-IMT keeps performing early-scheduling for the load-dependent instructions if the load is from the non-speculative thread.

Search latency incurred by speculative threads’ loads is hidden under N-IMT’s instruction-level and thread-level parallelism. To avoid being jeopardized by speculative threads’ too many searches, N-IMT gives less-speculative threads higher priority of LSQ port access. In the example of Figure 7, it is possible that *load A1* from thread 1 never be able to finish searching due to port contention while thread 1 is speculative. However, when thread 1 finally becomes a non-speculative thread, thread 1 has the highest priority for accessing LSQ ports and so it is guaranteed to finish searching. As mentioned above, on a memory dependence violation, N-IMT squashes the offending threads. N-IMT can avoid such squashes via well-known memory dependence synchronization techniques [18].

4.5 Speculative status overflow

Even though some physical registers can be freed when instructions commit, the load/store queue cannot be freed at instruction commit if the memory instruction is not from the head thread (i.e., non-speculative thread). So, due to this limitation of reusing load/store queue slots, it is possible that no slot in the load/store queue will be available for a speculative thread at some point. If this situation happens, no more instruction can be fetched from the thread until the thread finally becomes the head thread so that it can free and reuse the load/store queue slots. This is more likely to happen with floating-point application's larger threads, rather than with integer application's smaller threads.

4.6 Thread Completion

When a thread completes, N-IMT verifies the next-thread prediction, and frees the thread's processor resources. N-IMT flags thread completion on executing an instruction annotated with stop bits (mentioned in Chapter 3). Upon resolving the stop instruction's control flow, N-IMT verifies the next-thread prediction and either allows the next thread to commit or squashes later threads on thread misprediction. If squashed, the later threads' instructions free their physical registers and load/store queue slots, as squashed instructions do in SMT. If allowed to commit, a thread continues execution until the stop instruction performs instruction commit, and then the thread commits in program order after previous threads commit.

Thread commits free the processor resources occupied by the threads. Freeing active list slots and load/store queue slots is straightforward. Thread commit frees the physical registers not freed by instruction commits and removes the instructions left in the active list.

4.7 Precise Interrupt

When a thread has an interrupt or exception, it immediately squashes subsequent instructions within the thread and also squashes all following threads. N-IMT neither fetches instructions from the thread nor activates any thread until the instruction with the interrupt or exception becomes the oldest non-speculative instruction in the pipeline. After servicing the interrupt, N-IMT fetches instructions from the thread again and also activates new threads for fetching. By using this approach, N-IMT supports precise interrupts while still supporting speculative execution.

5 OPTIMIZATIONS

Section 5.1 proposes and discusses three primary optimizations, which are R&D-based fetch policy, multiplexing hardware contexts, and hiding thread start-up overhead. Then, Section 5.2 proposes and discusses two secondary optimizations, which are speculative releasing and reducing register pressure.

5.1 Primary Optimizations

5.1.1 Resource- & dependence-based fetch policy

As mentioned in Section 4.1.1, N-IMT, uses an unmodified ICOUNT policy [28]. I make the observation that the ICOUNT policy may be suboptimal for a processor in which threads exhibit control-flow and data dependence, and resources are relinquished in program (and not thread) order. For instance, later (program-order) threads may result in resource (e.g., physical registers, issue queue and load/store queue entries) starvation in earlier threads, forcing the later threads to squash and relinquish the resources for use by earlier threads. Unfortunately, frequent thread squashing due to indiscriminate resource allocation without regards to demand incurs high overhead. Moreover, treating (control- and data-) dependent and independent threads alike is suboptimal. Fetching and executing instructions from later threads that are dependent on earlier threads may be counter-productive because it increases inter-thread dependence delays by taking away front-end fetch and processing bandwidth from earlier threads. Finally, dependent instructions from later threads exacerbate issue queue contention because they remain in the queue until the dependences are resolved.

To mitigate the above shortcomings, O-IMT employs a novel Resource- and Dependence-based fetch policy (R&D-based fetch policy) that is bimodal. In the “dependent mode”, the policy biases fetch towards the non-speculative thread when the threads are likely to be dependent, fetching sequentially to the highest extent possible. In the “independent mode”, the policy uses ICOUNT when the threads are potentially independent, enhancing overlap among multiple threads. Because loop iterations are typically independent, the policy employs an *Inter-Thread Dependence Heuristic* (ITDH) to identify loop

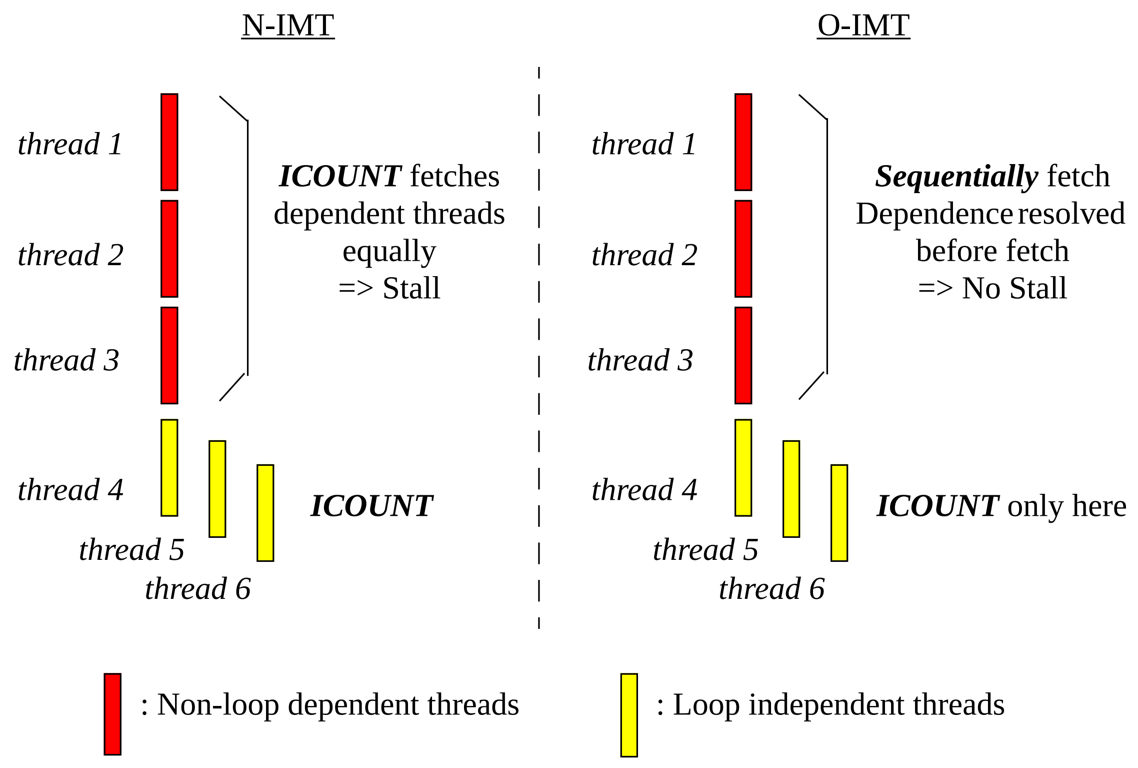

N-IMT
O-IMT
thread 1
thread 2
thread 3
thread 4
thread 5
thread 6
ICOUNT fetches dependent threads equally => Stall
ICOUNT
Sequentially fetch Dependence resolved before fetch => No Stall
ICOUNT only here
: Non-loop dependent threads
: Loop independent threads

Fig. 8. Dependence-based fetch.

iterations for the independent mode, otherwise considering threads to be dependent. ITDH predicts that subsequent threads are loop iterations if the next two threads' start PCs are the same as the non-speculative (head) thread's start PC.

Figure 8 illustrates the effect of using the dependence-based fetch for O-IMT, compared to N-IMT that always uses ICOUNT fetch regardless of inter-thread dependence. The figure uses an example code that has non-loop dependent threads and loop independent threads together. N-IMT uses ICOUNT fetch policy to fetch instructions equally even from dependent threads. N-IMT has lots of stalls in executing instructions from *thread 3* because *thread 3* is dependent on *thread 1* and *thread 2*, and it cannot make progress until *thread 1* and *thread 2* are done.

In contrast to N-IMT, O-IMT sequentially fetches instructions from threads when ITDH considers the threads to be dependent. Therefore, O-IMT does not have stalls that happen in N-IMT due to the fact that N-IMT blindly fetches from dependent threads equally. When ITDH considers threads (i.e., *thread 4*, *thread 5*, and *thread 6*) to be loop iterations, O-IMT uses ICOUNT to fetch from threads to maximize the thread-level paral-

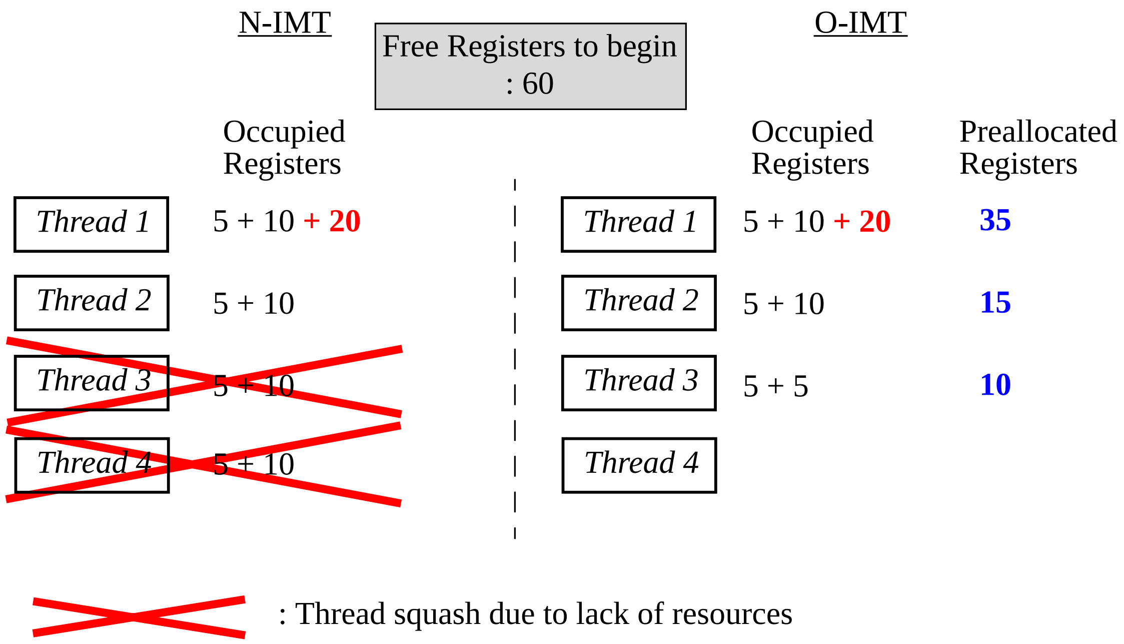
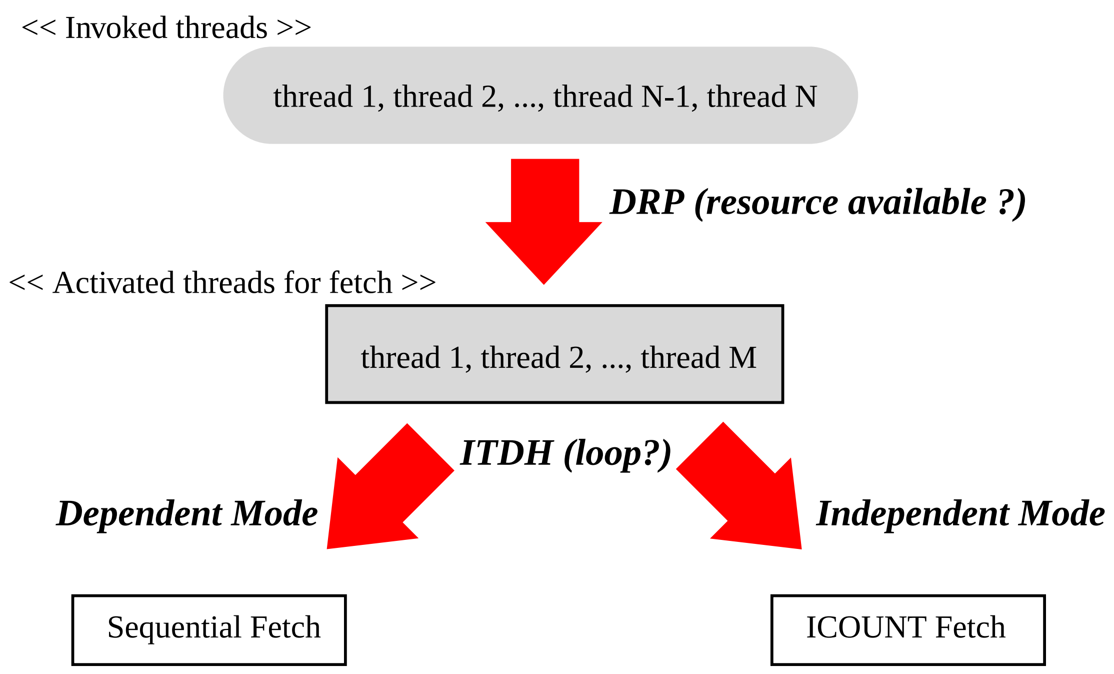


Fig. 9. Resource-based fetch: an example with registers.

lelism across loop iterations. As a result, N-IMT does not distinguish dependencies of threads for fetching, but O-IMT does.

To reduce resource contention among threads, the policy employs a *Dynamic Resource Predictor (DRP)* to initiate fetch from an invoked thread *only* if the available hardware resources such as physical registers, active list entries, and load/store queue entries exceed the predicted demand by the thread. DRP dynamically monitors the threads activity and allows fetch to be initiated from newly invoked threads when resources become available (either by thread commit or by instruction commit, as explained in Section 4.6). Section 5.1.2 will describe the implementation and issues of DRP in detail.

Figure 9 illustrates the performance impact of the resource-based fetch by using registers as an example. Physical registers are shared and occupied by different threads as soon as instructions are fetched and renamed. The figure assumes 60 physical registers to begin with. N-IMT fetches equally from different threads. Later, each thread occupies 15 registers and so N-IMT runs out of registers. However, *thread 1*, which is the head thread (the non-speculative thread), needs 20 more registers. To make overall progress or to avoid deadlock, N-IMT has to squash younger threads to take resources back for the head thread. In the figure, N-IMT has to squash *thread 3* and *thread 4*. While *thread 3* and



<< Invoked threads >>
thread 1, thread 2, ..., thread N-1, thread N
DRP (resource available ?)
<< Activated threads for fetch >>
thread 1, thread 2, ..., thread M
ITDH (loop?)
Dependent Mode
Independent Mode
Sequential Fetch
ICOUNT Fetch

Fig. 10. Combining Resource-based and dependence-based fetch mechanism.

thread 4 have been competing for fetch and processing bandwidth with *thread 1* and *thread 2* and slow down the execution of *thread 1* and *thread 2*, they are eventually squashed anyway.

O-IMT avoids such inefficiency by predicting resource demand and preallocating resources for each thread before fetching. In the figure, O-IMT uses DRP to predict the demand for physical registers and preallocates registers for each thread. Unlike N-IMT, O-IMT does not fetch any instructions from *thread 4* because it knows that there are not sufficient registers for *thread 4*. As a result, O-IMT does not delay the execution of older threads (i.e., *thread 1* and *thread 2*) for useless execution of *thread 4*, and it avoids thread squashes.

Figure 10 illustrates how the resource-based fetch mechanism and dependence-based fetch mechanism work together. Among invoked threads, namely from *thread 1* (the oldest thread) to *thread N* (the youngest thread), DRP chooses the group of threads to activate for fetching only from *thread1* to *thread M*, which are the biggest group of threads composed of under the assumptions as follows: (1) the group of activated threads is the subset of the group of invoked threads, (2) the group of activated threads should include the oldest thread, (3) the activated threads in the group should be contiguous, and (4) the size of the group of activated threads should be limited by the resource demand of threads and the resource availability in the pipeline. Then ITDH decides how to fetch among the activated threads (i.e., *thread 1*, *thread2*,... *thread M*). If the threads are loop iterations, ITDH chooses ICOUNT to fetch from among the threads. Otherwise, the policy biases fetch towards the non-speculative thread to fetch from.

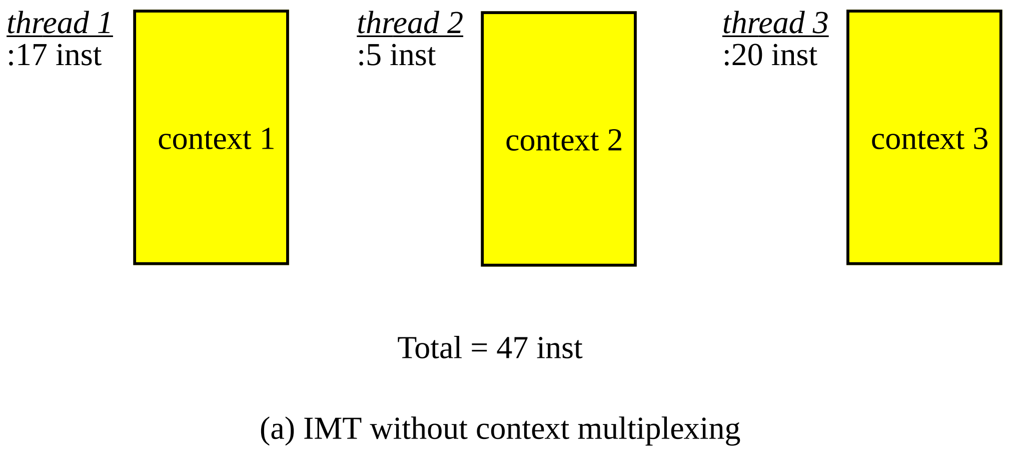
O-IMT's R&D-based fetch policy increases instruction throughput by choosing suitable instructions, thus making room for earlier threads when necessary. The policy alleviates inter-thread data dependence by processing producer instructions earlier and decreasing instruction execution stalls, thereby reducing pipeline resource contention.

In contrast to O-IMT, prior proposals for speculative threading using SMT (e.g., TME and DMT) do not address these issues in their fetch policies. TME uses biased-ICOUNT, a variant of SMT's ICOUNT which, as discussed above, does not consider resource availability and loop-level independence. DMT uses a variant of round-robin fetch policy without accounting for resource availability or independence. The policy statically partitions DMT's two fetch ports, and allocates one port for the non-speculative thread and the other for speculative threads in a round-robin manner. To alleviate resource pressure, DMT employs prohibitively large custom trace buffers to hold thousands of instructions, allowing DMT to retire instructions speculatively from the active list and to free resources so that DMT can make forward progress. However, DMT requires searching through the buffers upon committing and misspeculation recovery. Unfortunately, allowing frequent (associative) searches through large custom trace buffers is prohibitively slow and impractical.

5.1.2 Multiplexing hardware contexts

Much like prior proposals, N-IMT assigns one thread to a context. Because many programs have short threads [30] and real SMT implementations are bound to have only a few (e.g., 2-8) contexts, this approach often leads to insufficient instruction overlap. Larger threads, however, increase both the probability of control-flow or data dependence misspeculation [30] and the number of instructions discarded per misspeculation, and cause speculative buffer overflow [13]. Instead, to increase instruction overlap without the unwanted side-effects of large threads, O-IMT *multiplexes* the hardware contexts by mapping as many threads as allowed (on average 3-6 threads for SPEC2K) by the resources in one context.

Figure 11 shows how O-IMT maps multiple threads to a context. Without context



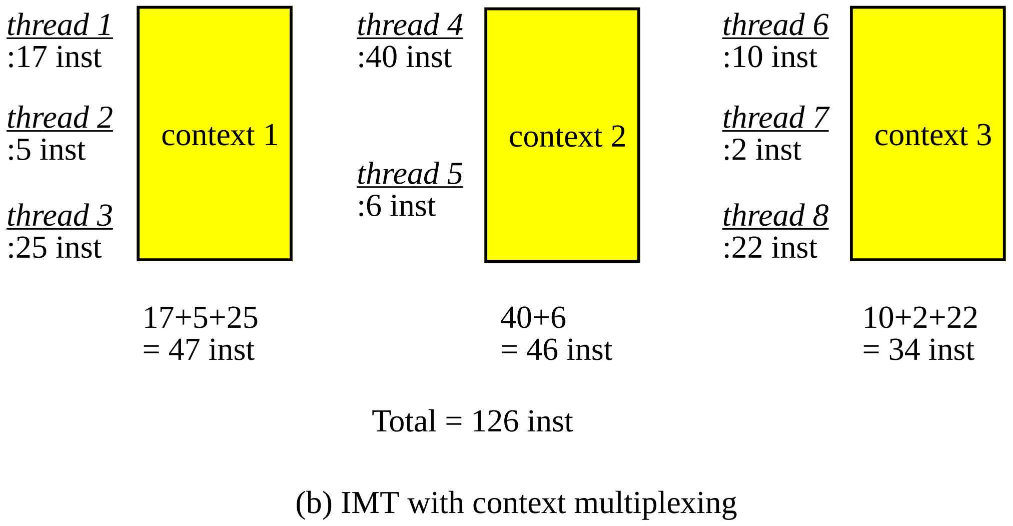


Fig. 11. Instruction overlap with context multiplexing.

multiplexing, there are only 47 in-flight instructions in the pipeline. Even though overall pipeline resources are available, IMT runs out of the context and so cannot bring more instructions into the pipeline. As opposed, IMT with context multiplexing can map *thread 1*, *thread 2*, and *thread 3* to *context 1* together. As a result, context multiplexing enables IMT to keep 126 in-flight instructions in the pipeline.

Two design complexity issues arise when mapping multiple threads to one context. First, conventional active list and load/store queue (LSQ) designs assume that instructions enter these queues in (the predicted) program order. This assumption enables the active list to be a non-searchable (potentially large) structure, and allows honoring memory dependences via an ordered (associative) search in the LSQ. If care is not taken, multiplexing would invalidate this assumption if multiple threads were to place instructions *out of program order* in the shared active list and LSQ. Such out-of-order placement would require an associative search on the active list to determine the correct instruction(s) to be removed upon commit or misspeculation. In the case of the LSQ, the requirements would be even more complicated: A memory access would have to search through the LSQ for an address match among the entries from the access's thread, and then (conceptually) repeat the search among entries from the thread preceding the access's thread, working towards older threads. Unfortunately, the active list and LSQ cannot afford these additional complications because the conventional active list is made large precisely due to the fact that the list does not have to be searched and the LSQ's ordered, associative search is already complex and time-critical.

Second, if one context has multiple *non-contiguous* threads, managing inter-thread dependence would become complicated. Two contiguous threads would be mapped to different contexts. To honor memory dependences, memory accesses would have to search other contexts holding earlier or later threads. Such searches complicate the critical LSQ.

O-IMT uses DRP to avoid the first issue. As mentioned in Section 5.1.1, DRP dynamically monitors the threads activity. Figure 12 (a) depicts an example of DRP. O-IMT indexes into the DRP table using the start PC of each thread. Each entry holds the numbers of physical registers, active list entries, and load/store queue entries used by the thread's last four execution instances. As the thread executes, the pipeline monitors a thread's resource needs. Upon thread commit, O-IMT updates the thread's DRP entry replacing the oldest instance's statistics with those of the current instance. DRP supplies the maximum among the four instances for each resource as the prediction for the next instance's resource requirement. It may seem that tracking the maximum among the last four instances of a thread may overestimate the thread's resource needs, and waste resources.

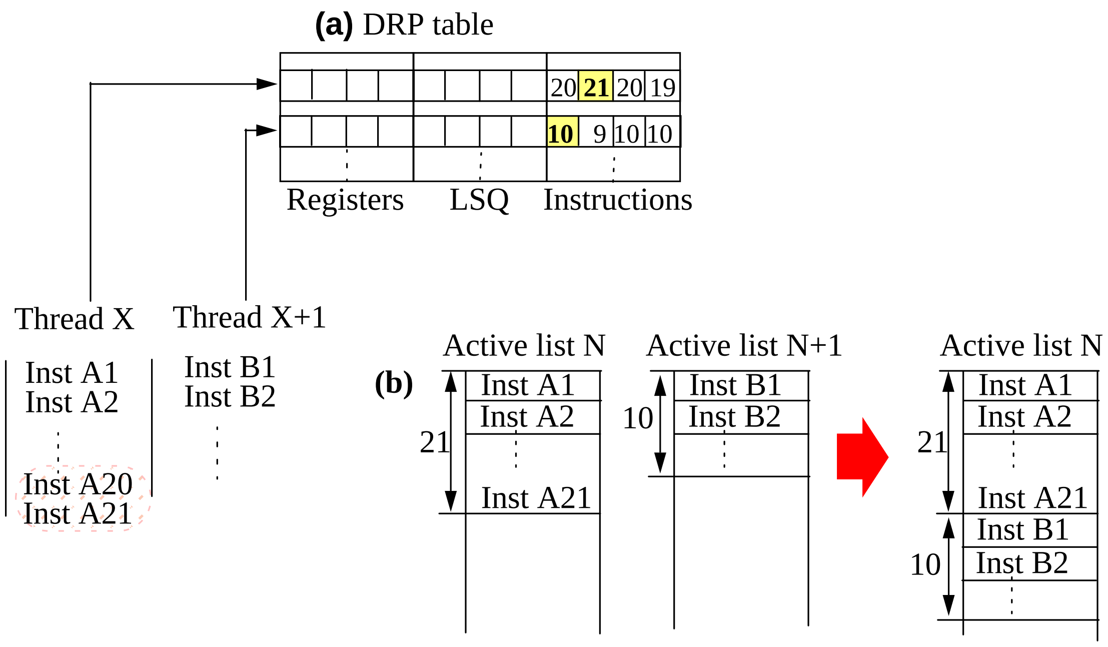


Fig. 12. Normal (successful) case of using DRP for context multiplexing: (a) DRP table. (b) Context multiplexing.

Section 6.2.1 shows the results indicating that overestimating resource usage using the maximum value among the last four instances works well in practice due to low variation in resource needs across nearby instances of a thread.

Using DRP, O-IMT avoids the first issue by placing instructions in program order. Within one context, O-IMT uses DRP's information to keep instructions in the active list in program order, despite out-of-order fetch among the threads assigned to a context. O-IMT creates a gap in the active list for the thread's yet-to-be-fetched instructions using resource prediction's gap length estimate. The next thread (invoked in program order) creates its gap after the previous thread's gaps, maintaining program order among the context's threads.

Because the gap lengths are estimates based on previous instances, it is possible that the gaps fill up before all the thread's instructions are fetched. In that case, O-IMT simply squashes later threads in the context to make room for the earlier thread. Figure 13 shows an example for that case. In such a way, DRP helps dynamically partition a context's active list so that instructions from one thread do not interfere with any other threads within the context. LSQ is handled similarly. To support multiplexing, O-IMT uses the

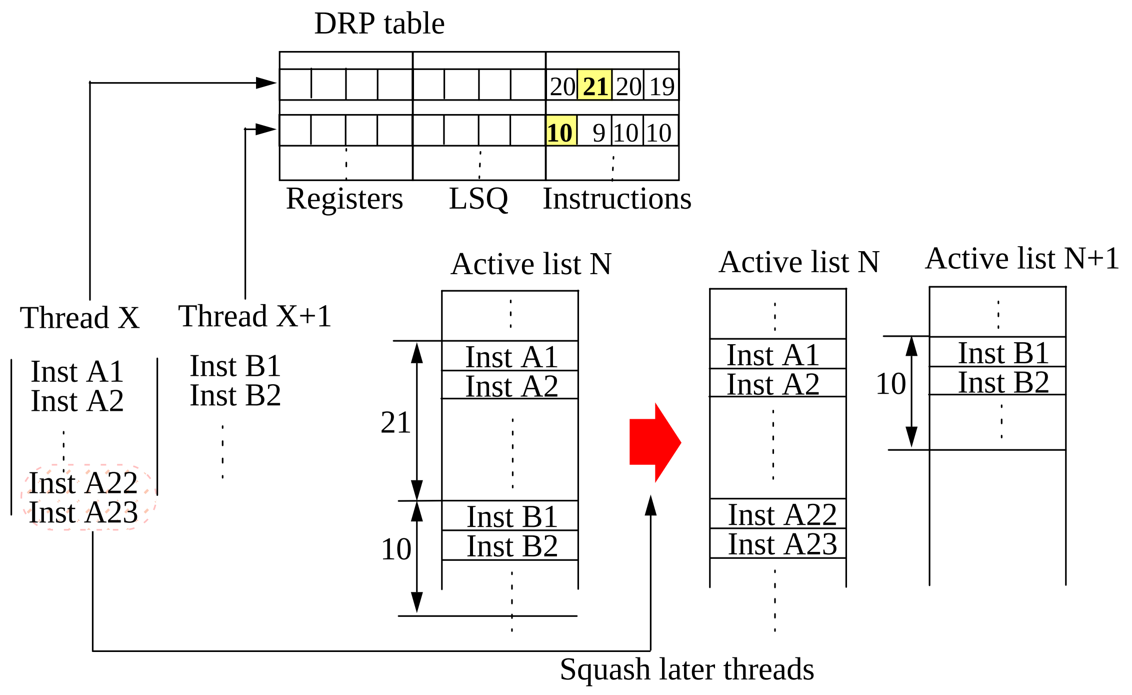


Fig. 13. Example of exceptional (unsuccessful) case of DRP prediction and effect on the context multiplexing.

DRP to create appropriately-sized gaps in the context's LSQ, similar to the active list. Thus, memory accesses from threads in a context are kept in program order in the LSQ. This approach is similar to that proposed in [7], which creates a gap in the LSQ to skip around a problematic branch.

O-IMT avoids the second issue by mapping contiguous threads to one context. Inter-thread dependences across threads within a context are treated similar to intra-thread dependence in the context, without involving other contexts. Figure 12 (b) shows how contiguous threads X and X+1 are mapped to a context. In addition to program order within contexts, O-IMT tracks the global program order among the contexts themselves for precise interrupts.

Context multiplexing further exploits the benefit of thread-level squashing. IMT with multiple threads in a context does not need to squash all subsequent instructions of the mispredicted branch within the context, but it needs to squash subsequent instructions only within the current thread of the mispredicted branch and keeps in-flight instructions from other threads in the same current context.

O-IMT' context multiplexing differs from previous proposals [24, 8] to map multiple threads onto a core and alleviate load imbalance. The schemes do not propose *simultaneously* executing multiple threads in a context, but advocate executing the next thread only *after* suspending [24] or finishing [8] the previous thread. Such serialization underutilizes resources. Moreover, these proposals do not mention the complexity issues discussed above.

5.1.3 Hiding thread start-up overhead

Even though the next thread's start PC is known, fetching instructions from the next thread has to wait until the rename tables are set up. The updating of local, and master and preassign tables must complete *before* the thread's instructions can be renamed (Section 4.3). The rate of rename table updating is limited by the rename table bandwidth. In conventional pipelines, this bandwidth matches the pipeline width and is sufficient for the peak demand. In contrast, N-IMT's requirement of updating the tables creates a burst demand that may exceed the bandwidth. Therefore, updating the tables may take a few (e.g., 2-4) cycles to finish.

O-IMT prevents the bandwidth constraint from imposing an overhead on thread start. While the current thread's instructions are fetched, O-IMT invokes the next thread, obtains the next thread's descriptor from the descriptor cache, and sets up the rename tables well before needing to fetch the next thread's instructions. O-IMT utilizes the rename table bandwidth unused by the current thread's instructions to update the three tables. For instance if in a cycle only 6 instructions are renamed but the rename tables have the bandwidth to rename 8 instructions, O-IMT uses the unused bandwidth to modify the tables. Thus, O-IMT overlaps starting up a thread with the execution of previous threads, hiding thread start-up overheads.

Thread start-up overhead exists for Multiscalar, TME, and machines like DMT. In Multiscalar, the next thread needs to set up its rename tables so that the next thread can appropriately wait for register values from previous threads. However, Multiscalar does not address this issue. TME incurs extra cycles to set up the rename tables, and employs an extra dedicated bus for a bus-based write-through scheme to copy rename maps. DMT copies not only register values but also the entire return address stack at the start of a thread. DMT does not concretely address the delay of the copying, and instead assumes the delay away using extra wires to do the copying.

5.2 Secondary Optimization

5.2.1 Speculative releasing

As mentioned in Section 4.2.1, N-IMT disallows communication of speculative register values across threads, delaying values until intra-thread speculation is resolved. However, this approach has issues of not only implementation difficulty but also performance loss.

Holding the speculative value until the speculation is resolved does not cause an implementation difficulty. The problem is to release the value when the speculation is resolved. When intra-thread speculation is resolved, it is possible that the instructions holding destination register values are already gone from the pipeline (but do not commit yet). These instructions are sitting in the active list, which is the reorder buffer in superscalar. The possible implementation is to search the active list to find instructions that need to release the destination values to later threads whenever any intra-thread speculation is resolved. However, the conventional active list is the non-searchable structure, which facilitates high clock speeds and avoids the wiring complexity. To make the active list non-searchable structure, conventional SMT (or superscalar) separates the issue queue from the active list (or reorder buffer) and searches only the issue queue to find ready-to-execute instructions. Searching the active list whenever intra-speculation is resolved adds significant complication to the conventional SMT pipeline.

From the performance point of view, the rational of holding speculative values is that the chance of squashing later threads from intra-thread misspeculation can be removed by delaying propagation of speculative register values to later threads until intra-thread speculation is resolved. However, superscalar does not have such delaying overhead and uses the produced values as soon as the values become ready. Such delaying overhead clearly introduces the inefficiency to speculative execution.

Fortunately, there have been lots of proposals to improve branch predictions since Multiscalar was proposed [3], including hybrid predictor [5]. The improvement of the branch prediction gives us chance to release speculative values with less worry about incorrect intra-thread speculation. Therefore, N-IMT releases the speculative register values to later threads as soon as the values are ready just like superscalar, and it minimizes the waiting time of dependent instructions from later threads. Section 6.3.1 shows that some benchmarks loose considerable amount of performance due to the delaying of values, and releasing speculative values alleviates the performance loss.

The problem with releasing speculative values to later threads is that register or memory values from incorrect intra-thread execution can be consumed by later threads, and later threads must be squashed even if they have been correctly predicted when the incorrect intra-thread speculation is resolved.

O-IMT annotates the rename tables and load/store queue entries to identify when values are consumed by *some* later thread. If the thread producing the values internally squashes the values and there is an indication that the values were consumed by some later thread, then all later threads are squashed. To track whether a register value is consumed by a later thread, O-IMT tags the rename maps, which are copied into the local table at thread start-up, to identify the register values that are produced by previous threads and that may be consumed by the current thread. Maps added to the local table for registers produced by the current thread are *not* tagged so. If an instruction sources a value via a tagged map, then the physical register corresponding to the map is marked as consumed by a later thread. To track whether a memory value is consumed by a later thread, later threads tag the earlier thread's load/store queue entry upon obtaining the entry's value. In the process of internal squashing, if a physical register or load/store queue entry being rolled back is marked as consumed by a later thread, O-IMT squashes *all* later threads.

IMT could keep track of which threads actually consumed the incorrect speculative values and minimize the penalty of misspeculation by squashing only contaminated threads, instead of squashing all subsequent threads whenever the value was consumed by any later thread. However, to minimize the hardware complexity, O-IMT does not choose this implementation option at this research.

5.2.2 Reducing physical register pressure

N-IMT's out-of-order fetch overlaps farther instructions than SMT's in-order fetch. As a result, more instructions are in flight in N-IMT than SMT, especially for programs with long threads (e.g., floating-point codes). To alleviate the resulting physical register pressure, O-IMT employs a two-phase instruction and thread commit strategy. Instructions commit in program order within the thread, but out of order compared to the instructions in preceding threads; threads commit in global program order. Instruction commits free some physical registers out of program order, well before the thread commits. Thread commits free the rest of the physical registers later. If there are pending exceptions, then instruction commit within the thread is frozen till all previous threads commit, maintaining precise interrupts.

Conventional pipelines free the physical register previously mapped to the same architectural register as the committing instruction's destination. Along the same lines, the create mask specifies the collective architectural destination registers of a thread. Instruction commits and thread commits free only the set of physical registers that are previously mapped to the thread's create-mask registers. Within this set, instruction commit frees its destination's previous physical register only if the register was allocated within the thread, and not some earlier thread. If the previous register was allocated by an earlier thread, then three implications follow: (1) The previous register visible to later threads must be a preallocated register in the earlier thread. (2) There is no guarantee that all use (i.e., read) of the previous register value in the earlier thread is complete. (3) Thread commit frees the register later. To identify registers allocated within the thread, O-IMT annotates the rename map (in the local table) for all non-preallocated registers, so that a later instruction with the same destination can free the register when the later instruction commits.

Freeing physical registers at instruction commit, even if previous threads may not have committed, does not cause loss of correctness due to two reasons: One, because instruction commit is in program order within the thread, there are no rollbacks of the committed instruction due to internal branch mispredictions. Two, rollbacks due to external misspeculations (thread mispredictions and inter-thread memory dependence violations) restart from the beginning of the thread and recompute the freed registers. These two reasons guarantee that the freed physical registers need not be resurrected due to rollbacks.

Because most of the previous speculative architectures [23,13,25] do not map multiple threads to one core, they may not experience register pressure. As such, the architectures do not address this issue. However, for programs with long threads, register pressure is an issue even for these architectures.

6 RESULTS

A execution-driven simulator of an out-of-order SMT pipeline with extensions is built to evaluate a base superscalar processor (using a single SMT context), and the three speculatively-threaded processors, IMT, DMT, and TME. The Multiscalar compiler [30] is used to generate optimized MIPS binaries. The superscalar, TME, and DMT experiments use the plain MIPS binaries (without Multiscalar annotations). The IMT binaries include Multiscalar's thread specifications and register communication instructions

Table 1 depicts the system configuration parameters for this study. The base pipeline assumes an eight-wide issue out-of-order SMT with eight hardware contexts. The pipeline assumes two i-cache ports for all machines including the base superscalar. To exploit the extra i-cache port for superscalar, the branch predictor allows up to two predictions per cycle. In addition to the base pipeline, O-IMT also uses a 64-entry DRP table and a 4-entry ITDH table to optimize fetch. To gauge speculative threading's potential conservatively, IMT's performance is compared against an aggressive superscalar implementation that assumes the same resources available within the SMT pipeline including the high-bandwidth of branch prediction and fetch, and the large register file. The aggressive superscalar also assumes a large active list of 1024 entries, because active lists are FIFO structures and are inherently scalable.

Table 2 shows the SPEC2K applications used in this study, and the branch prediction accuracy and superscalar IPC achieved per application. The reference input set is used for all of the benchmarks. To allow for practical simulation turnaround times, the simulator skips the first 3 billion instructions before simulating a total of 500 million instructions (plus overhead instructions, in IMT's case). Total number of cycles is used as the base metric to compare performance. In the case of IMT, the cycle counts include the overhead instructions.

The rest of the results are organized as follows. The performance of N-IMT and O-IMT are compared to the aggressive superscalar, and I break down the performance bottlenecks O-IMT optimizes. Then results on the effectiveness of O-IMT's microarchitecture optimizations are presented. Then I present O-IMT's ability to increase issue queue and

Table 1. System configuration parameters.

Processing Units		**System**	
Issue width	8	**DRP table**	64 entries (3 x 356bytes)
Issue queue	64entries		
Number of contexts	8	**ITDH**	4 program counters
Branch unit	hybrid GAg & Pag 4Kentires each, 1K-entry 4-way BTB	**L1 cache 2-port i-cache & 4-port d-cache**	64K 2-way, pipelined 2-cycle hit, 32-byte block
Mis Penalty	7cycles		
Functional units	8 integer, 8pipelined floating-point	**L2 cache**	2M 8-way, pipelined 10-cycle hit, 64-byte block
Register file	356 INT/ 356 FP	**Memory**	80 cycles
Per Context			
Active list	128 entries	**Squash buffer**	64 entries
LSQ	32 entries, 2 ports	**Thread desc. cache**	16K 2-way, 2-cycle hit

Table 2. Applications, and their branch misprediction rates and superscalar IPCs.

INT Bench.	**Branch misp. (%)**	**IPC**	**FP Bench.**	**Branch misp. (%)**	**IPC**
bzip	5.5	1.6	*ammp*	1.1	1.1
gap	2.8	3.0	*applu*	0.1	2.4
gcc	4.7	1.8	*art*	0.6	0.4
gzip	6.2	1.7	*equake*	0.5	1.0
mcf	7.6	0.3	*mesa*	2.0	2.6
parser	3.3	1.2	*mgrid*	0.8	2.3
perl	5.3	1.7	*sixtrack*	1.9	2.4
twolf	10.9	1.2	*swim*	0.1	0.9
vortex	0.6	1.9	*wupwise*	0.2	2.4
vpr	6.8	1.1			

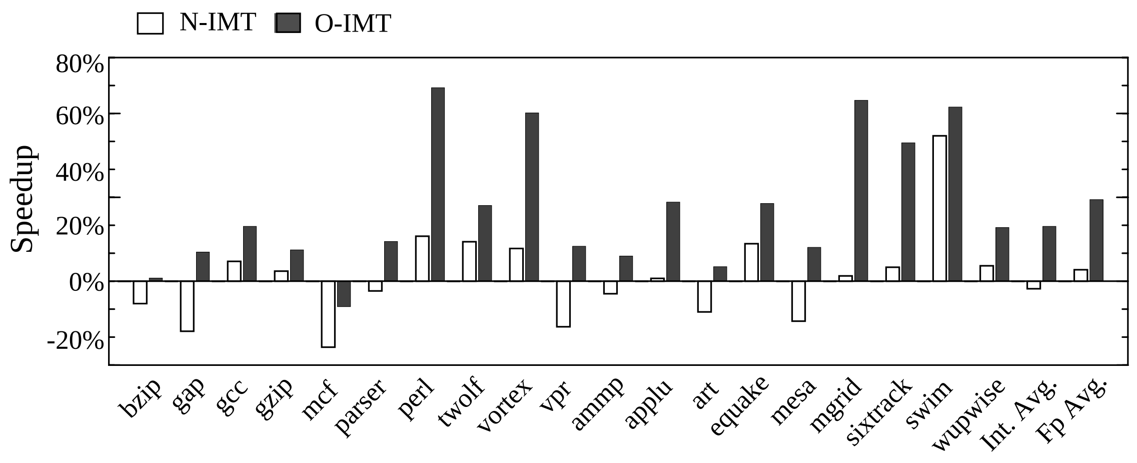


Fig. 14. Performance comparison of N-IMT and O-IMT normalized to the baseline superscalar.

load/store queue efficiency as compared to superscalar using thread-level parallelism. Finally, I compare and contrast O-IMT with TME and DMT, two prior proposals for speculative threading using SMT hardware.

6.1 Base System Results

Figure 14 motivates the need for optimizing the speculative threading performance on SMT hardware. The figure presents execution times under N-IMT and O-IMT normalized to the base superscalar. The figure indicates that N-IMT's performance is actually inferior to superscalar for integer benchmarks. N-IMT reduces performance in integer benchmarks by as much as 24% and by on average of 3% as compared to superscalar. Moreover, while the results for floating-point benchmarks vary, on average N-IMT only improves performance slightly over superscalar for these benchmarks. The figure also indicates that microarchitectural optimizations substantially benefit compiler-specified threading, enabling O-IMT to improve performance over superscalar by as much as 69% and 65% and by on average of 20% and 29% for integer and floating-point benchmarks respectively.

Figure 15 compares the key sources of execution overhead in superscalar (left bar), N-IMT (middle bar) and O-IMT (right bar). From top to bottom, the breakdown shows the overhead of squashing instructions due to branch misprediction (both within and across threads) and resource pressure (in N-IMT and O-IMT), underutilized instruction fetch bandwidth, memory waiting stalls (due to data cache misses), register data dependence stalls, and runtime instruction overhead for IMT machines.

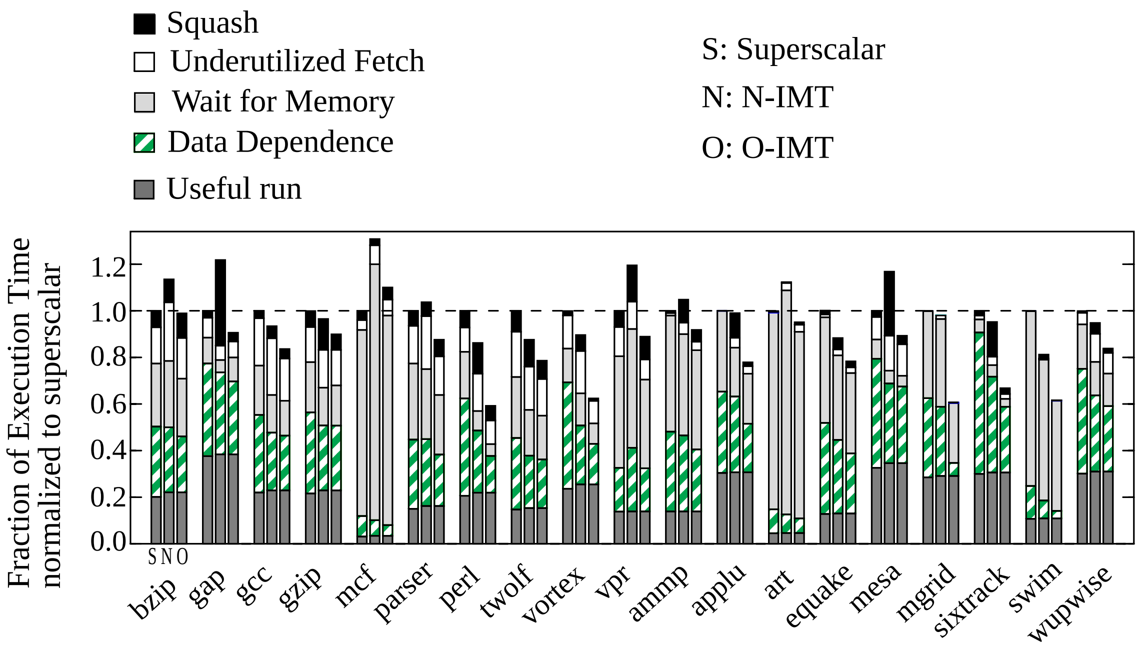


Fig. 15. Breakdown of execution into instruction execution and pipeline stalls.

Not surprisingly, the dominant execution time component in superscalar that speculative threading improves is the register data dependence stalls. The IMT machines extract parallelism across threads and increase the likelihood inserting suitable instructions (from across the threads) into the pipeline, thereby reducing data dependence stalls. Speculative threading also helps overlap latency among cache misses in benchmarks with available memory parallelism across threads, reducing memory stalls as compared to superscalar. These benchmarks most notably include *perl*, *applu*, *mgrid*, and *swim*. Finally, the cycles spent executing instructions (denoted by "useful run") across the machines are comparable, indicating that the instruction execution overhead of compiler-specified threading is negligible.

There are a number of benchmarks in which N-IMT actually reduces performance as compared to superscalar. In *gap*, *vpr*, *ammp*, and *mesa*, N-IMT simply fetches instructions indiscriminately without regards to resource availability and from the wrong threads (using round-robin) resulting in high misspeculation/squash frequency. In *mcf*, *vpr*, and *art*, N-IMT increases the data dependence or memory stalls by bringing unsuitable instructions into the pipeline. In *mcf* N-IMT increases the L1 data-cache miss ratio as compared to superscalar because later threads' cache accesses conflict with those from the non-speculative thread. In *art*, N-IMT increases the L1 data-cache miss ratio by delaying the issue of data cache misses from the non-speculative thread. Finally, in *bzip* N-IMT incurs a high thread start-up delay and increases the fraction of stalls due underutilized fetch.

The graphs also indicate that O-IMT substantially reduces the stalls as compared to N-IMT. O-IMT's resource- and dependence-based fetch policy and context multiplexing reduce data dependence and memory stalls by fetching and executing suitable instructions. Accurate resource allocation and prediction minimizes the likelihood of misspeculation and reduces squash stalls. Finally, hiding the thread start-up delay reduces the likelihood of underutilized fetch cycles by increasing the overlap among instructions. The combined effect of these optimizations results in superior performance in O-IMT as compared to superscalar and N-IMT. Section 6.2 presents detail analysis on these techniques' contributions to O-IMT's performance.

6.2 Primary Optimizations

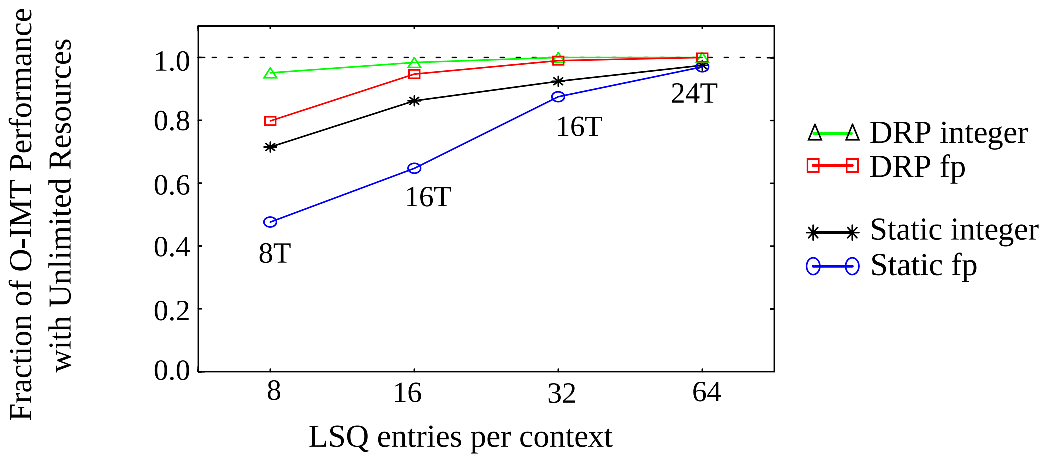
In this section, I present the result of resource allocation and prediction. Then, I present the results of O-IMT's R&D-based fetch policy and context multiplexing, which use the proposed resource allocation and prediction techniques. Then, I present the results of the optimization of hiding the thread start-up delay.

6.2.1 Resource allocation & prediction

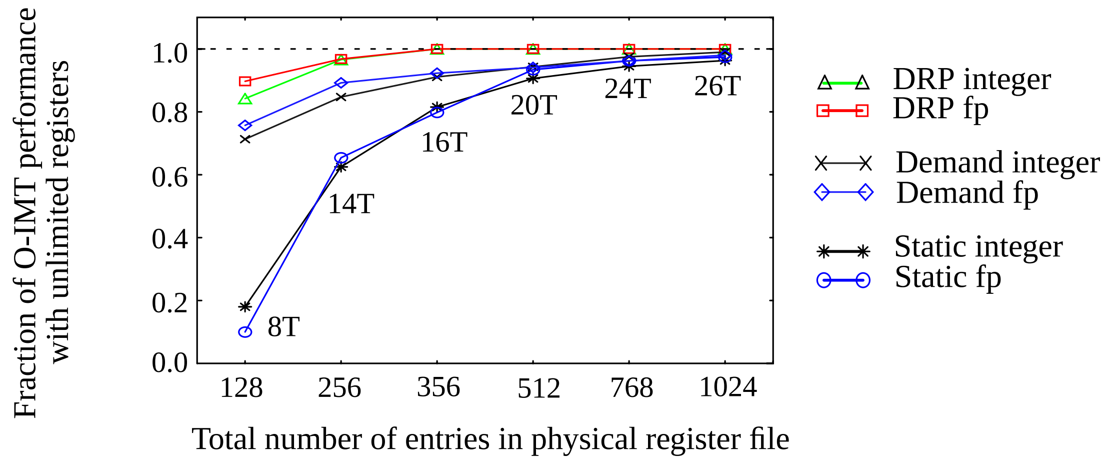
Figure 16 illustrate the need for dynamic resource allocation, and the impact of DRP's accurate prediction on performance in O-IMT. Figure 16 (a) compares performance under dynamic partitioning using DRP against static partitioning for the load/store queue (LSQ) entries, and Figure 16 (b) does for the register file. In the register file case, the figure also plots demand-based allocation of entries by threads, allowing for threads to allocate registers upon demand without partitioning or reservation. Meanwhile, in the LSQ case, the figure does not plot demand-based allocation of entries. The reason is that the LSQ entries cannot be simply shared by multiple threads because the LSQ supports the associative searches to honor the memory dependence (Section 5.1.2).

The graphs plot average performance (for integer and floating-point benchmarks separately) as a fraction of that in a system with unlimited resources. Context multiplexing allows more threads per context, thereby requiring a different (optimal) number of threads depending on the availability of resources. These graphs plot the optimal number of threads (denoted by the letter T) for every design point on the x-axis.

Figure 16 (a) indicates that DRP successfully eliminates all stalls related to a limited



(a) Impact on the load/store queue



(b) Impact on the physical register file

Fig. 16. Dynamic vs. static partitioning of resources.

number of LSQ entries in integer benchmarks with as few as 16 LSQ entries per context. In contrast, a static partitioning scheme requires as many as 64 LSQ entries to achieve the same results. Similarly, in floating-point benchmarks, DRP can eliminate virtually all LSQ stalls with 32 entries per context, whereas static partitioning would require two times as many entries per context. Moreover, static partitioning can have a severe impact on benchmark performance, reducing performance on average by 40% given 16 entries per context.

Figure 16 (b) indicates that the results for allocating registers are more dramatic. DRP allocation of registers can achieve the best performance with four times fewer registers in integer and floating-point benchmarks. Moreover, static partitioning of registers for smaller register file sizes (>256) virtually brings execution to a halt and limits performance. Demand-based allocation of registers substantially improves performance over static partitioning, allowing threads to share a large pool of registers effectively even with as few as 128 registers per integer and floating-point register files. Demand-based allocation, however, only reaches within 10% of DRP-based allocation and, much like static partitioning, requires four times as many registers to bridge the performance gap with DRP. Demand-based allocation's performance improves gradually beyond 256 registers. Register demand varies drastically across threads resulting in a slow drop in misspeculation frequency, and consequently gradual improvement in performance, with an increase in register file size.

Table 3 presents statistics on the accuracy of DRP for the dynamic allocation of registers, active list and LSQ entries. Unfortunately, demand for resources actually slightly varies even across dynamic instances of the same (static) thread. The predictors learn and predict the worst-case demand on a per-thread basis, thereby opting for over-estimating the demand in the common case. Alternatively, predictors that would target predicting the exact demand for resources may frequently under-estimate, thereby causing later threads to squash and release resources for earlier threads (Section 5.1). The table depicts the frac-

Table 3. Accuracy of dynamic resource prediction and allocation.

Benchmarks	LSQ			Registers			Active List		
	acc(%)	avg. used	avg. over	acc(%)	avg. used	avg. over	acc(%)	avg. used	avg. over
integer	99.2	7.4	0.8	97.5	15.9	3.0	98.9	17.0	2.1
floating-point	99.6	19.7	1.8	98.4	29.8	2.9	99.7	43.9	1.8

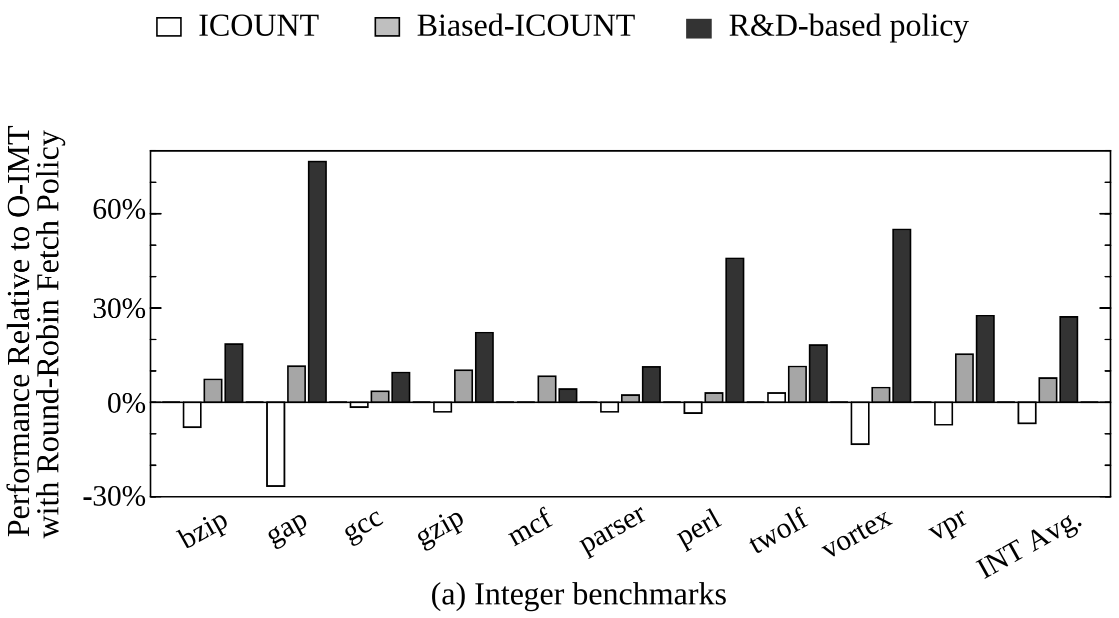
tion of the time and the amount by which DRP on average over-estimates demand. The results indicate that predicting based on the demand for the last four executed instances of a thread leads to high accuracy for (over-)estimating the resources. More importantly, the average number by which the predictors over-estimate is relatively low, indicating that there is little opportunity lost due to over-estimation.

6.2.2 Resource- & dependence-based fetch policy

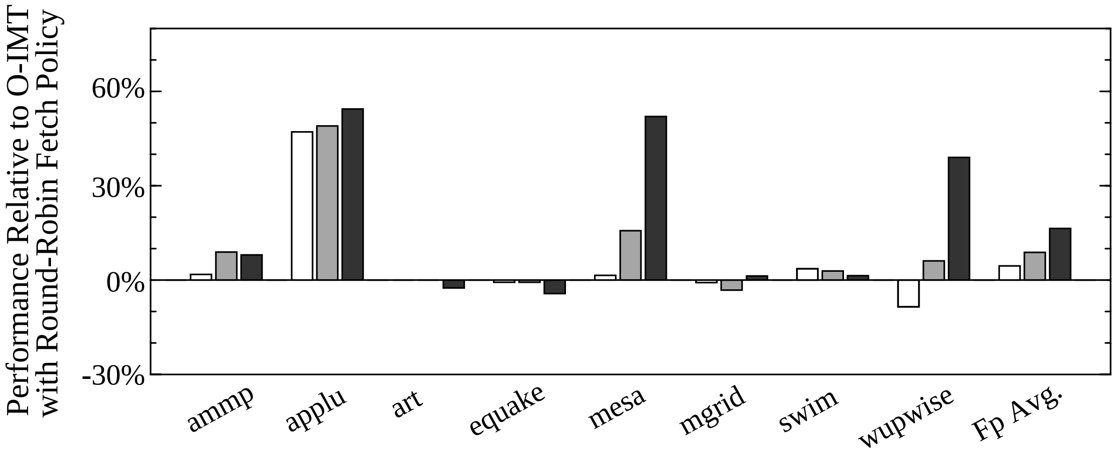
O-IMT's fetch policy gives the priority to the non-speculative (head) thread and only fetches from other threads when: (1) ITDH indicates the likelihood of parallelism and the availability of suitable instructions, and (2) DRP indicates the availability of resources based on the predicted demand. In contrast, a round-robin policy (used in DMT) would let later dependent threads hog the resources while earlier threads attempt to make forward progress, potentially reducing performance. Similarly, an ICOUNT policy [9] (used in SMT) that favors a thread with the fastest issue rate without regards to resource usage or dependence may indiscriminately allocate resources to speculative threads, leading to resource bottlenecks. Finally, a constant bias in the non-speculative thread's fetch priority in a biased-ICOUNT policy [31] (used in TME) may improve performance only slightly when resource usage and dependence across threads drastically vary.

Figure 17 shows O-IMT's performance under four different fetch policies. From left to right, the figure plots three priority-based fetch policies, ICOUNT, biased-ICOUNT, and resource- and dependence-based fetch policy. The graphs plot the performance improvement of those fetch policies over round-robin fetch policy for all benchmarks and the average of integer and floating-point programs separately.

Among integer benchmarks, *gap* shows the most benefit from employing the resource- and dependence-based fetch policy for O-IMT, and *mcf* shows the least benefit. The figure indicates that indeed in integer benchmarks, ICOUNT reduces the average performance over round-robin, because it allows speculative threads issuing at a high rate to inadvertently fetch, allocate resources, and subsequently squash. The figure also shows that without an efficient fetch policy, all the optimizations that O-IMT employs to bring more instructions (from speculative threads) to the pipeline or to increase overlap among fetched instructions can actually hurt the performance. As a result, some benchmarks including *bzip and gap* performs even worse than N-IMT (This comparison is not shown here, but it can be seen by comparing the performance degradation in this figure to that in Figure 14), which uses ICOUNT as a fetch policy. Biased-ICOUNT addresses this shortcoming in ICOUNT by biasing the priority towards the non-speculative thread by a con-



(a) Integer benchmarks



(b) Floating-point benchmarks

Fig. 17. The impact of fetch policy.

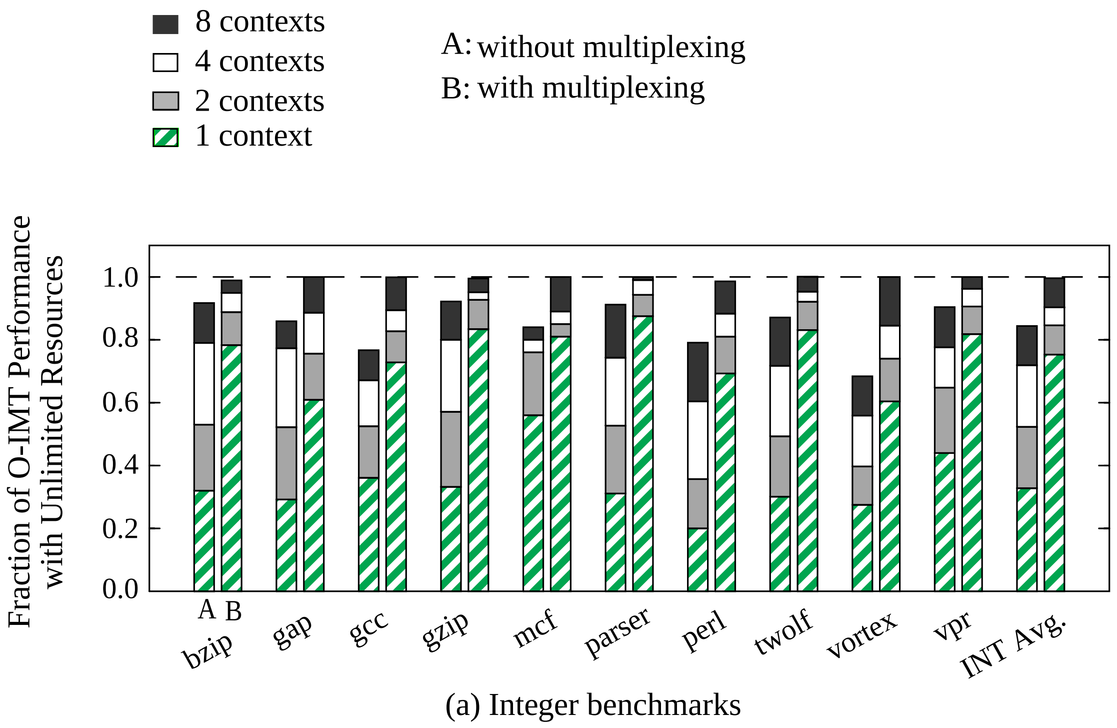
stant value, and improving performance over round-robin. O-IMT's resource- and dependence-based fetch policy significantly improves performance over round-robin by preventing later threads from fetching unless: (1) there are resources available, and (2) the threads are loop iterations and likely to be independent.

Among floating-point benchmarks, *applu* shows the most benefit from all priority-based fetch policy compared to round-robin. As a result, the floating-point benchmarks actually slightly benefit from ICOUNT and biased-ICOUNT on average. The floating-point applications exhibit a high fraction of thread-level parallelism and independence across threads. As in SMT, ICOUNT allows for the threads making the fastest rate of progress to proceed, improving performance over a round-robin policy. Biased-ICOUNT reduces the likelihood of misspeculation due to resource pressure, and as such improves performance over ICOUNT. O-IMT's fetch policy performs best by allowing the most suitable instructions to flow through the pipeline.

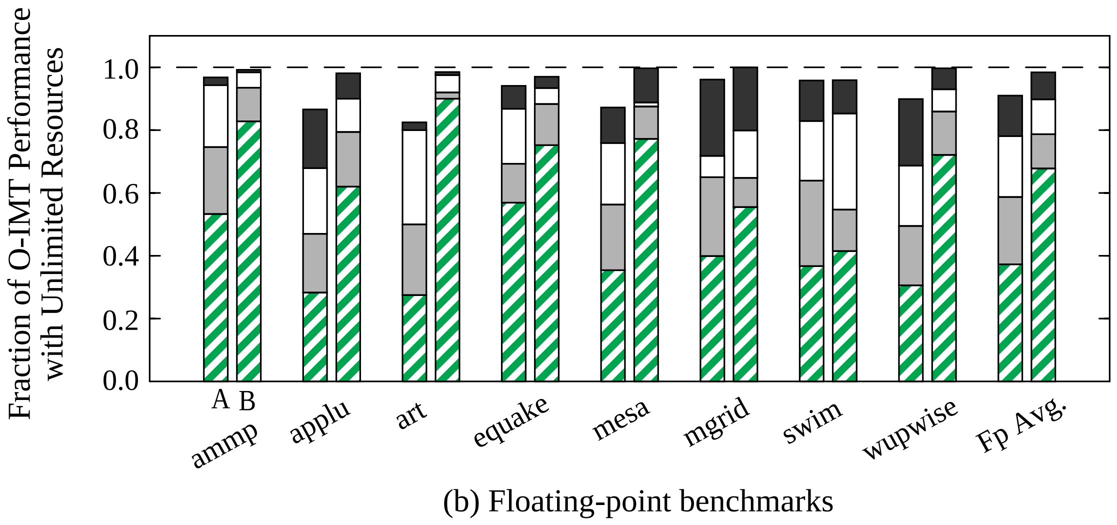
6.2.3 Multiplexing hardware contexts

Multiplexing offers two key advantages for applications with short threads. Multiple threads per context help increase the number of suitable in-flight instructions. Alternatively, multiplexing makes unused contexts available to threads across multiple applications in a multiprogrammed (SMT) environment. Figure 18 illustrates the impact of multiplexing on O-IMT's performance. The Y axis shows the fraction of O-IMT performance with different number of contexts. The X axis shows the benchmarks and the average of the integer and the floating--point programs separately. The base case is the O-IMT performance with unlimited resources including contexts. The left bar shows the performance of O-IMT without context multiplexing, and the right bar shows the performance of O-IMT with context multiplexing. To accurately gauge the overall impact on performance with an increase in available resources, the register file size are varied linearly from 132 to 356 (adding 32 registers to the base case with every context) when varying the number of contexts from one to eight.

The figure indicates that without multiplexing, neither integer nor floating-point benchmarks can reach best achievable performance even with eight hardware contexts. Moreover, performance substantially degrades (to as low as 35% in integer applications on average) when reducing the number of contexts. Without multiplexing, *perl* shows the worst performance as low as 20% of the unlimited resources case, when there is only one context. Meanwhile, with multiplexing, *perl* achieves the performance as high as 69% of the unlimited resources case.



(a) Integer benchmarks



(b) Floating-point benchmarks

Fig. 18. The impact of context multiplexing.

Multiplexing's performance impact is larger with fewer contexts because context resources are used more efficiently. Multiplexing best benefits integer benchmarks with short-running threads allowing for two contexts (e.g., as in a HyperThreaded Pentium 4 [15]) to outperform eight contexts without multiplexing. Multiplexing also benefits floating-point benchmarks, reducing the required number of contexts. Floating-point benchmarks' performance, however, scale well with an increase in the number of contexts even without multiplexing due to these benchmarks' long-running threads.

6.2.4 Hiding thread start-up overhead

Figure 19 illustrates the impact of thread start-up delay on O-IMT's performance. From left to right, the bars represent the performance of four-cycle start-up delay, four-cycle start-up delay, and O-IMT's overlap to hide the start-up delay. The base case is an O-IMT with no start-up delay. The graphs plot the performance of three different cases normalized to the performance of the base case. The figure indicates that a higher start-up delay of four cycles on average can reduce performance by 10% on average in integer benchmarks. *perl* is the one which suffers most from the thread start-up delay. The results show that in integer benchmarks with multiple-cycle start-up delay, there are enough slacks in the SMT's shared pipeline. Consequently, hiding the slacks by overlapping executions of multiple threads improves the performance.

Because of their long-running threads, the floating-point benchmarks can amortize a higher start-up delay, and as such show less performance sensitivity to start-up delay. In contrast, O-IMT's mechanism for overlapping thread start-up on average almost achieves ideal performance (incurring no start-up delay).

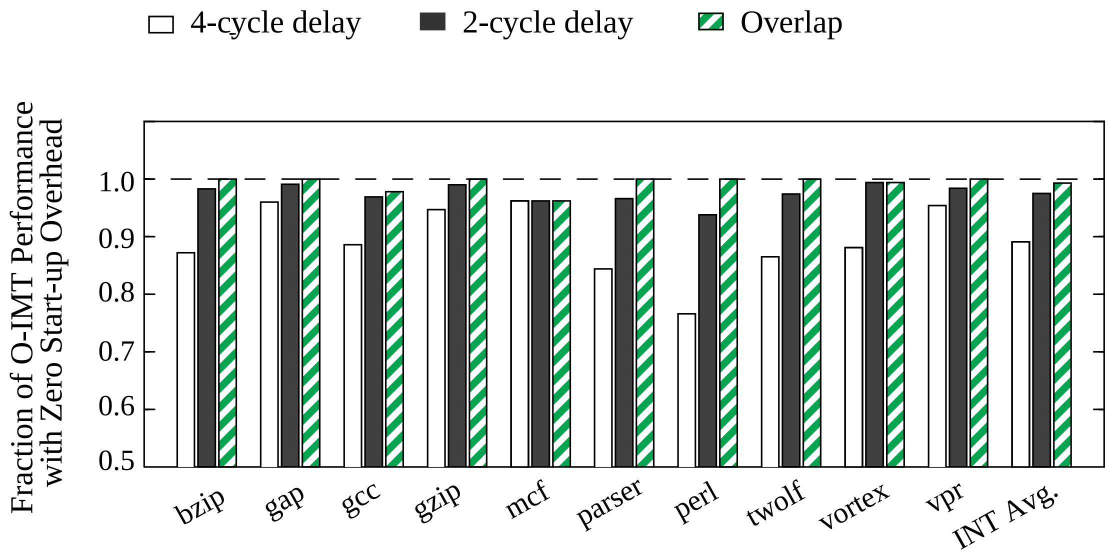
6.3 Secondary Optimizations

In this section, I will discuss the results of the secondary optimizations such as speculative releasing and reducing register pressure by two-phase commit.

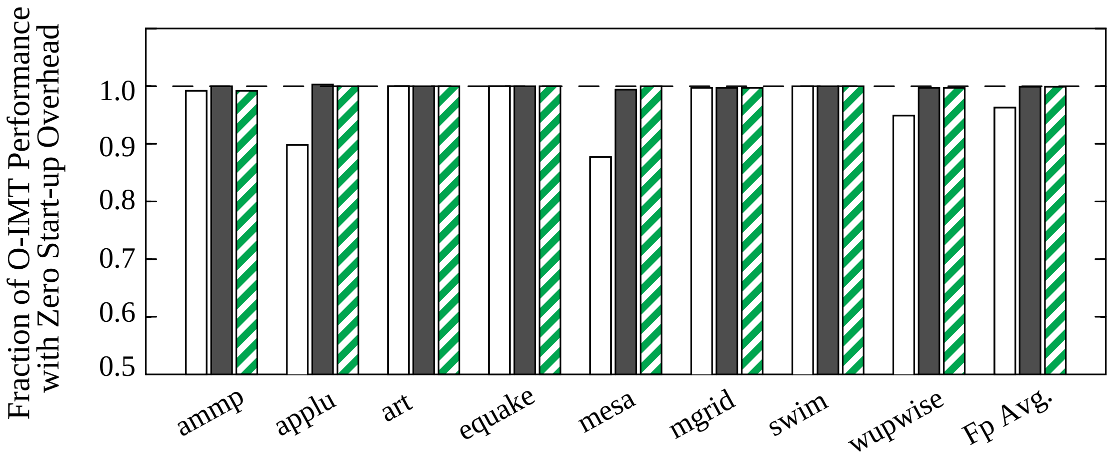
6.3.1 Speculative releasing

One of the benefits of IMT is its ability to do thread-level squashing. It means that for an intra-thread branch misprediction, IMT squashes instructions only from the current thread of the mispredicted branch, while it keeps later thread instructions intact. To isolate the intra-thread misprediction, N-IMT does not release register values to later threads until all previous internal branches are resolved. However, this approach has an implementation issue and performance issue, as mentioned in Section 6.3.1. O-IMT solves these two issues by speculatively releasing register values to later threads.

-



(a) Integer benchmarks



(b) Floating-point benchmarks

Fig. 19. The impact of start-up delay.

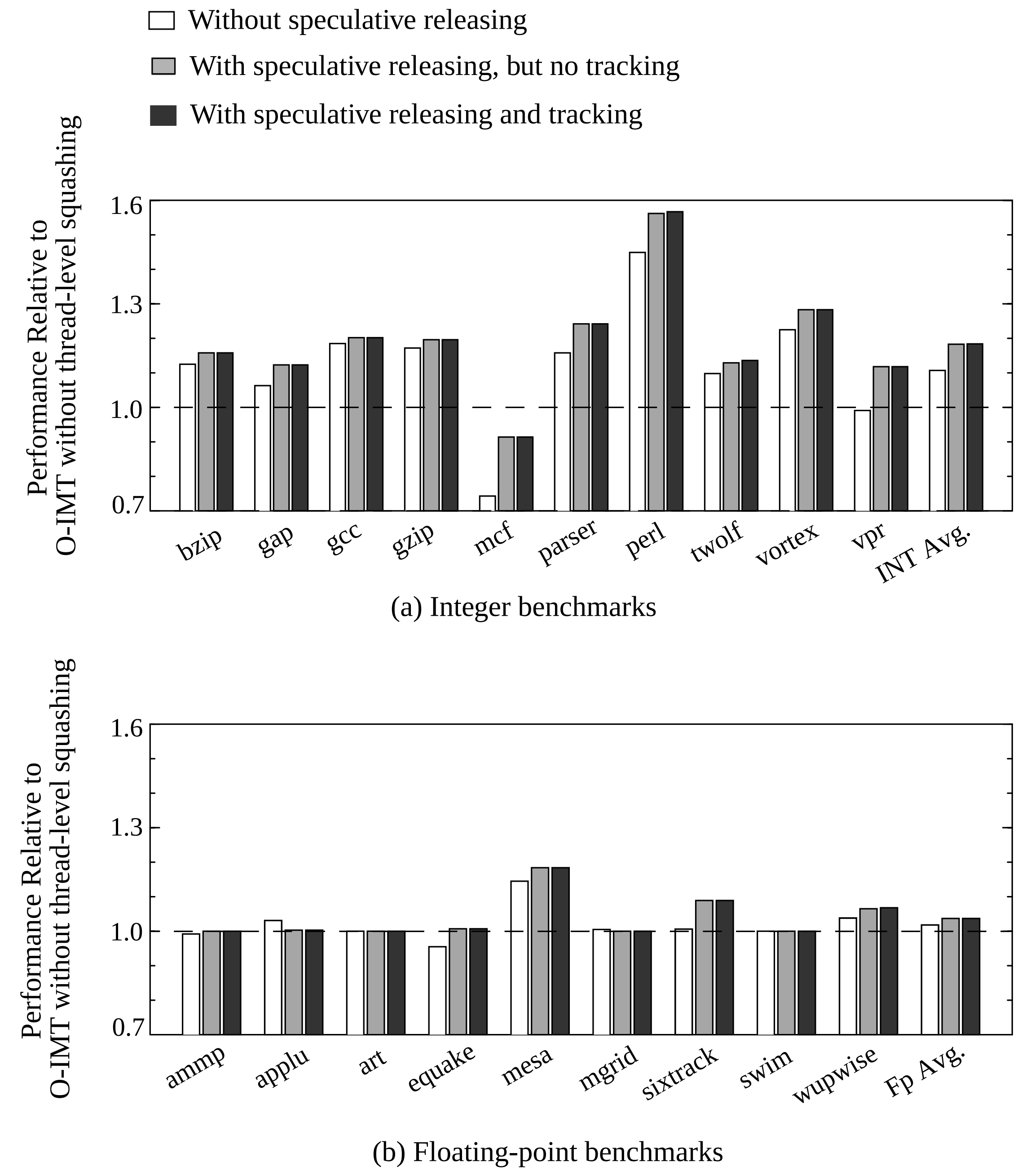


Fig. 20. The impact of thread-level squashing with speculative releasing.

Figure 20 illustrates the impact of thread-level squashing on O-IMT's performance with and without speculative releasing. The base case is an O-IMT that does not use thread-level squashing but speculatively releases the register values. From left to right, the bars represent (1) a O-IMT with thread-level squashing but without speculative releasing, (2) a O-IMT with thread-level squashing and speculative releasing but no tracking, and (3) a O-IMT with thread-level squashing, speculative releasing, and tracking. The second bar represents the performance of the optimization used for O-IMT in this dissertation. The third bar is the performance of the aggressive implementation of the optimization, which keeps tracking the threads that actually use the speculative values from the mispredicted control path and so tries to minimize the squash penalty of internal branch misprediction while releasing values speculatively (Section 5.2.1).

The figure shows the performance improvement in integer and floating-point benchmarks when using thread-level squashing. The figure also compares the impact of holding on to speculative register values — to remove the chance of squashing later threads on internal branch mispredictions — and releasing speculative register values — to minimize the waiting time of dependent instruction from later threads.

The figure indicates that thread-level squashing has a considerable impact on O-IMT's performance in integer applications due to the higher internal branch mispredictions, improving performance by 11% on average. In integer applications, thread-level squashing saves 20 instructions among 43 instructions in flight on average per branch misprediction. Moreover, releasing speculative register values further improves performance by additional 7%, indicating the low likelihood of speculative register values used by later threads with branch mispredictions occur in earlier threads.

Table 4 shows the statistics for the extra squashes due to speculative releasing. The second and fifth column show the number of intra-thread branch mispredictions occurred while executing 500 million instructions. The third and sixth columns show the number of squashes occurred due to speculative releasing without tracking and ratio of the squashes over the intra-thread branch mispredictions. The statistics show that only 14% (on average) of internal branch mispredictions cause squashing later threads due to consuming incorrect register values speculatively. While thread-level squashing does not greatly benefit floating-point applications due to extremely low internal branch misprediction rates (only 2% of performance improvement over the base case), speculative releasing adds 2% of performance improvement, resulting 4% of performance improvement over the base.

As mentioned in Section 5.2.1, tracking the threads that actually use the speculative value from the mispredicted control path may complicate the pipeline design, and I do not consider it as a design option in this dissertation. Moreover, the results in this figure show that the performance improvement from tracking over non-tracking is trivial.

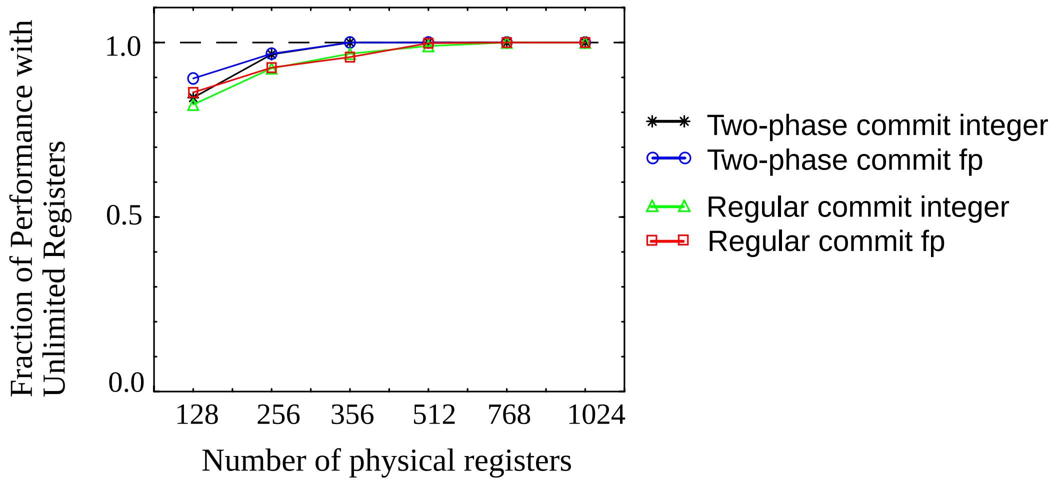


Fig. 21. The impact of two-phase commit.

6.3.2 Reducing register pressure

O-IMT allows later threads to commit their instructions out of program order and release physical registers. Figure 21 illustrates the impact of IMT's two-phase commit on relieving register pressure. The figure compares the performance of regular commit against two-phase commit in O-IMT. To show O-IMT's potential for performance improvement using two-phase commit, the numbers are normalized to an O-IMT with an

Table 4. Squashes due to speculative releasing.

INT Bench.	# Internal branch misp.	# Squash due to spec. releasing (%)	**FP Bench.**	# Internal branch misp.	# Squash due to spec. releasing (%)
bzip	4841k	1083k (22%)	*ammp*	1987807	17009 (9%)
gap	1540k	176k (11%)	*applu*	24204	5 (0%)
gcc	3591k	448k (13%)	*art*	803434	185887 (23%)
gzip	5202k	510k (10%)	*equake*	896002	3 (0%)
mcf	11858k	1293k (11%)	*mesa*	1563164	59187 (4%)
parser	4279k	1084k (25%)	*mgrid*	842	2 (0%)
perl	361k	1k (0%)	*sixtrack*	241131	1327 (1%)
twolf	8775k	489k (6%)	*swim*	24609	0 (0%)
vortex	1070k	100k (9%)	*wupwise*	403406	64479 (16%)
vpr	3648k	2571k (71%)			

unlimited number of registers. The results indicate that two-phase commit has limited advantage over regular commit in integer benchmarks and does not benefit them beyond 356 registers due to lack of register pressure. Floating-point benchmarks show similar results. With 356 registers, two-phase commit reduces register pressure over regular commit improving performance by 8% on average with maximum of 20% in *gap*, *mgrid*, *su2cor*, and *tomcatv*. Two-phase commit's advantage eventually disappears when there are enough registers to satisfy the demand for registers by threads.

6.4 Miscellaneous Results

This section first discusses the performance implication of O-IMT from the issue queue's and LSQ's point of view. Then it discusses the performance loss when O-IMT forgoes early-scheduling for the instructions that are dependent on loads, in order to avoid increasing the complexity of the scheduler design.

6.4.1 Issue queue & LSQ performance sensitivity

In SMT/superscalar pipelines, the issue queue and LSQ(s) sizes are often the key impediments to performance scalability [20]. Thread-level speculation helps increase the effectiveness of these queues of a given size by allowing suitable instructions from across the threads to enter the queues. Figure 22 illustrates improvements in superscalar and O-IMT performance with increasing number of entries in the issue queue and LSQ. The graphs indicate that as compared to a superscalar with a 32/16 entry queue pair, O-IMT can achieve the same performance with half as many queue entries. Because the issue queue and LSQ are often on the pipeline's critical path, O-IMT can actually help reduce the critical path and increases clock speed by requiring smaller queues.

The graphs also indicate that for integer applications, performance levels off with 64/32 entry queue pairs, with up to 50% performance improvement over a 16/8 entry queue pair. O-IMT maintains a 25% additional improvement in performance over superscalar by extracting thread-level parallelism. Moreover, superscalar's performance never reaches that of O-IMT's even with 256/128 entry queues. High branch misprediction frequency in integer applications ultimately limits performance even with a larger issue queue and LSQ. In O-IMT, a mispredicted branch within a thread only squashes instructions from that thread, thereby allowing suitable instructions from future threads to remain in the pipeline while a branch from an earlier thread mispredicts.

In contrast, superscalar's performance continues to scale for floating-point applications with higher levels of ILP, up to the 256/128 entry queues. O-IMT significantly

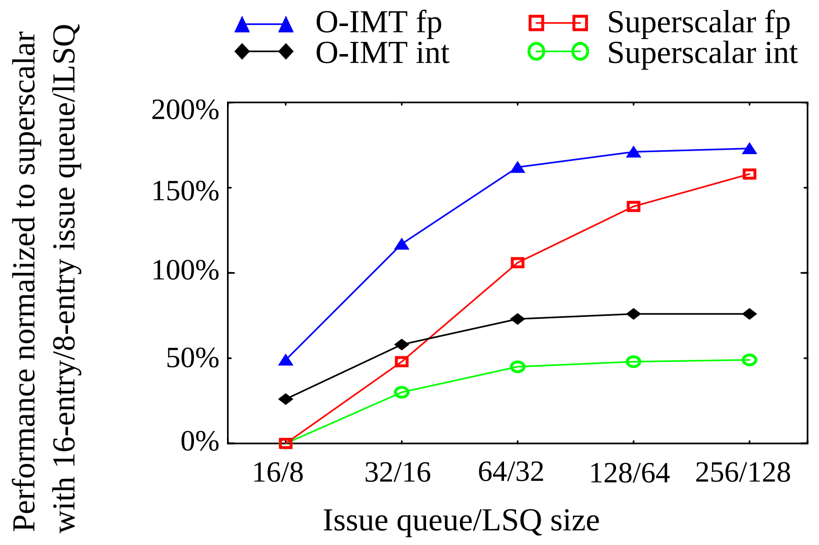


Fig. 22. Issue queue/LSQ sensitivity.

enhances queue efficiency over superscalar and achieves superscalar's performance at the 256/128 design point with less than a quarter of the queue entries. Moreover, O-IMT's performance levels off at the 64/32 design point, obviating the need for large queues to extract the available parallelism.

6.4.2 Forgoing early-scheduling for load-dependent instructions

As mentioned in Section 4.4, IMT does memory disambiguation by searching through LSQs from different contexts. Such multiple searches make the L1 hit latency variable because of the uncertainty of the latest store's existence and the port contention. Meanwhile, high performance superscalar processors speculatively schedule instructions dependent on the load with the assumption that the load is a cache hit. Variable hit latencies complicate such a scheduling mechanism. To avoid complicating the scheduler, IMT foregoes early-scheduling for the instructions that are dependent on the load. The exceptional case is for loads that are from the head context. The loads from the head context do not search any earlier context because there is no earlier context, and so their latencies are not variable.

Figure 23 shows the performance loss when O-IMT forgoes early-scheduling for the instructions that are dependent on the load. The base case is an O-IMT with the aggressive scheduler that can do early-scheduling even with variable hit latency of loads. The left bar

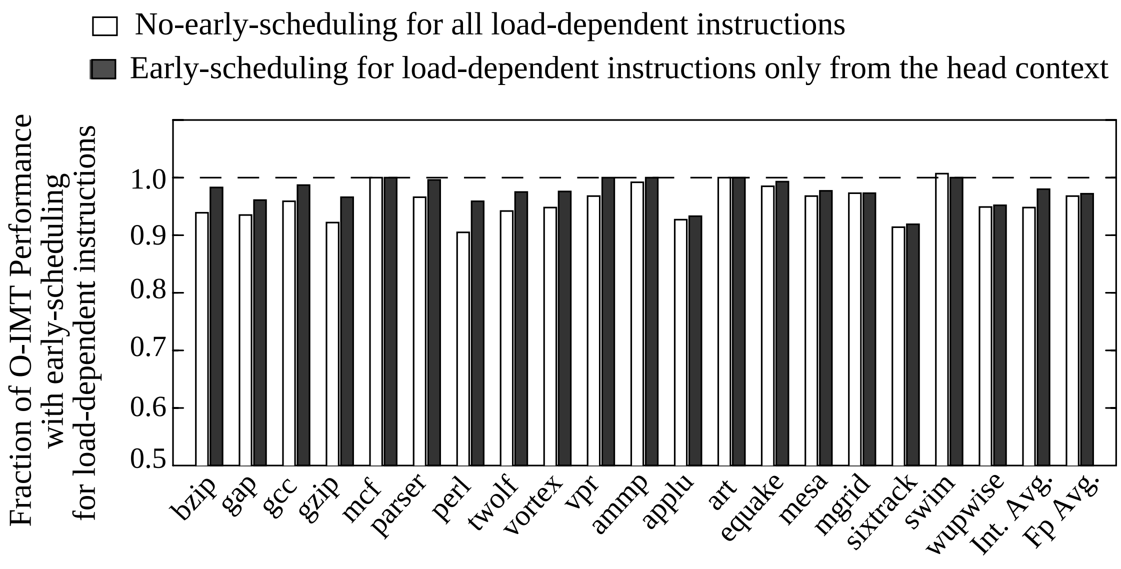


Fig. 23. Impact of early-scheduling for load-dependent instructions

shows the performance of an O-IMT that forgoes early-scheduling for all instructions that are dependent on any load. Therefore, the scheduler of this bar will be even simpler than that of conventional superscalar. The right bar shows the performance of the O-IMT that does early-scheduling for the instructions dependent on the load only if the load comes from the head context. Otherwise, the O-IMT forgoes early-scheduling. The performance of these two mechanisms is normalized to that of the base case in the figure.

The figure shows that no-early-scheduling degrades O-IMT's performance by 5% on average, as much as 10% for integer benchmarks and by 3% on average, as much as 9% for floating-point benchmarks. However, O-IMT does not need to forgo early-scheduling for the instructions dependent on the load that is from the head context. When O-IMT takes this fact into the consideration, it reduces the performance loss only to 2%. O-IMT does context multiplexing. Therefore, the head context will accommodates multiple contiguous threads including the head thread (the non-speculative thread), and the loads that are from the head context cover the majority of important loads. As a result, early-scheduling only for the instructions dependent on the load from the head context alleviates most of performance degradation due to inefficient scheduling compared to the base case.

Floating-point benchmarks show different results. *Swim* shows the most interesting results. Unlike integer benchmarks, floating-point benchmarks have lots of parallelism so that the efficient scheduling is less required. Early-scheduling also has negative impact to the performance. Early-scheduling assumes that all loads are going to be L1 hit, and it flushes out and reissues all instructions subsequent to a load when the assumption turns out to be wrong and the load is L1 miss. As a result, *swim* shows a little performance benefit by completely forgoing early-scheduling. As shown in Section 6.2.3, O-IMT's context multiplexing in floating-point benchmarks is less effective than in integer benchmarks because of bigger thread size. Therefore, in floating-point benchmarks, early-scheduling only for the instructions dependent on the load from the head context does not improve the performance of completely foregoing early-scheduling because the head context will accommodate only a few threads.

6.5 Comparison to TME & DMT

In this section, O-IMT's performance is compared against TME and DMT. The models used in this comparison for TME and DMT are quite aggressive allowing for a conservative comparison against these machines. I assume no contention for TME's mapping synchronization bus [31]. I also assume a 256-entry custom trace buffer per context (for a total of 2048 entries) with zero-cycle access penalty and selective recovery (squash) for DMT. As proposed, TME fetches from two ports using biased-ICOUNT, and DMT uses a dedicated i-cache port for the non-speculative thread and a shared i-cache port for speculative threads. I also assume an improvement over the proposed machines by allowing TME and DMT to take advantage of both i-cache ports when there are no speculative threads running. I compare these improved models against the original proposals.

Figure 24 compares speedups of the optimized TME and DMT machines, against O-IMT normalized to the baseline superscalar. Unlike O-IMT, TME and DMT reduce the average performance with respect to a comparable superscalar. TME [31] primarily exploits thread-level parallelism across unpredictable branches. Because unpredictable branches are not common, TME's opportunity for improving performance by exploiting parallelism across multiple paths is limited. TME's eagerness to invoke threads on unpredictable branches also relies on the extent to which a confidence predictor can identify unpredictable branches. A confidence predictor with low accuracy would often spawn threads on both paths, often taking away fetch bandwidth and processing bandwidth from the correct (and potentially predictable) path. An accurate confidence predictor would result in a TME machine that performs close to, or improves performance slightly over, the baseline superscalar machine. *Vpr* and *mesa* are benchmark examples in which the confidence predictor predicts accurately, allowing TME to improve performance over superscalar.

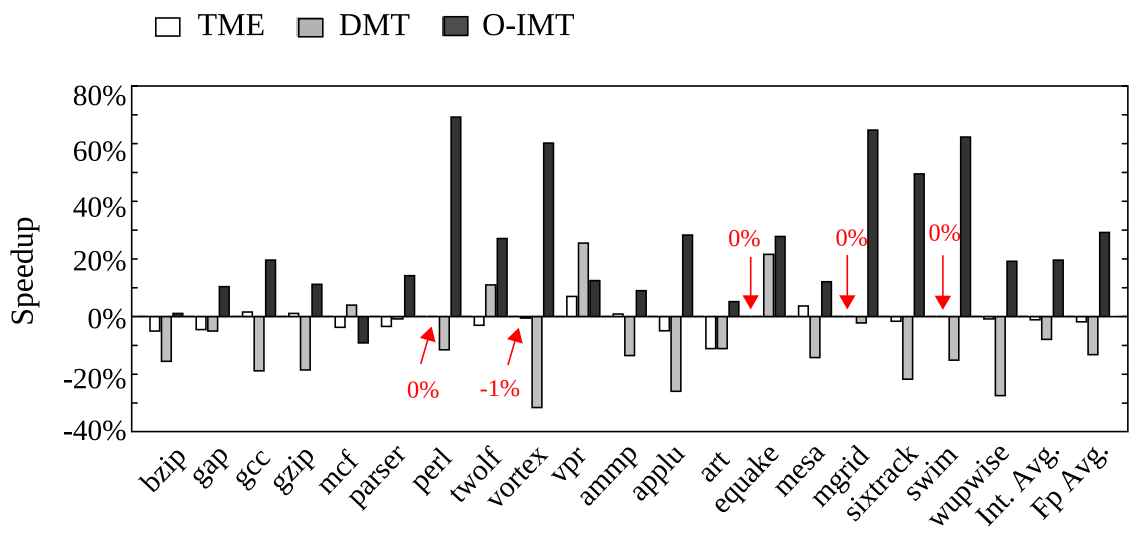


Fig. 24. Performance comparison of TME, DMT, and IMT normalized to baseline superscalar.

DMT’s poor performance is due to the following reasons. First, DMT often suffers from poor thread selection because it spawns a new thread when the fetch unit reaches a function call or a backward branch, and selects the new thread to include instructions after the call or backward branch. Therefore, DMT precludes exploiting the potentially high degree of parallelism that exists across inner loop iterations. Moreover, DMT’s threads are typically inordinately long, increasing the probability of data dependence misspeculation despite using “dataflow” dependence prediction. Second, DMT achieves low conditional branch and return address prediction accuracies because DMT spawns threads out of program order while global branch history and return address stack require in-program-order information to result in high prediction accuracy. The results indicate that DMT results in lower branch and return address prediction accuracies whether the branch history register and return address stack contents are cleared or copied upon spawning new threads.

Due to the low accuracy of DMT’s branch and data-dependence prediction, DMT fetches, executes, and subsequently squashes twice as many instructions as it commits (i.e., DMT’s commit rate is one third of its fetch/execute rate). With the exception of *mcf, twolf, vpr,* and *equake*, in which branch prediction accuracies remain high, all benchmarks exhibit significantly lower branch prediction accuracy as compared to our baseline superscalar, resulting in a lower average performance than superscalar.

Figure 25 corroborates the results that the optimized models for TME and DMT actu-

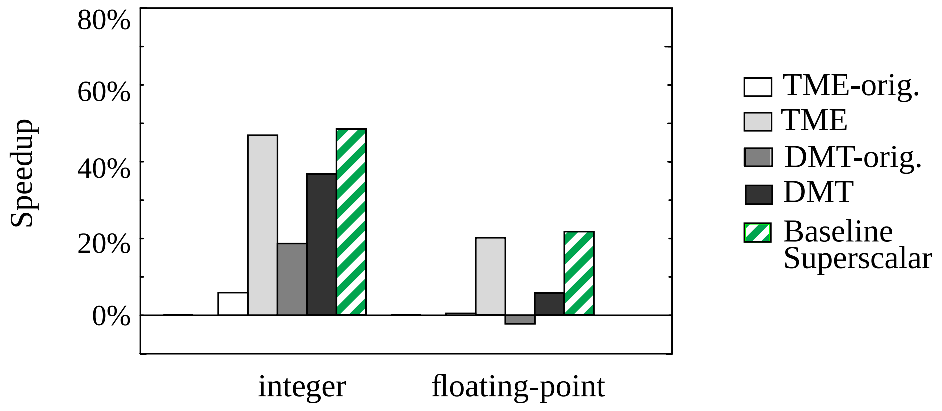


Fig. 25. Performance comparison of TME, DMT, and IMT normalized to prior-work's superscalar [31].

ally improve performance over the original proposals. The figure compares the speedups from the original proposals for TME and DMT against the optimized proposals for them normalized to a less aggressive superscalar processor with a single i-cache fetch port and a single branch prediction per cycle used in the prior studies [31,1]. The figure also plots the speedups for the baseline superscalar assumed in this dissertation with two i-cache ports and two branch predictions per cycle. Not surprisingly, TME substantially benefits from the second i-cache port in the common case when there are no speculative threads but fails to improve performance over the baseline superscalar. DMT's performance also improves slightly due to a better utilization of the second i-cache port when the speculative threads do not actively fetch. However, DMT's poor threading severely limits the performance improvement, resulting inferior performance compared to TME or the aggressive (Baseline) superscalar.

6.6 Comparison to single thread run on single SMT context

This section shows how much performance gain IMT can actually get compared to a genuine SMT when there is enough contexts available on a SMT. Figure 26 shows O-IMT performance compared to a base case, which is different from the base case that I used through the Result chapter. The base case used in this figure is a superscalar that has only hardware resources as much as a single SMT context.

Perl achieves the biggest performance improvement with 199% speed-up and the bar goes beyond the range of Y axis. This figure shows that IMT improves the single program performance by an average of 50% for Integer benchmarks over genuine SMT execution

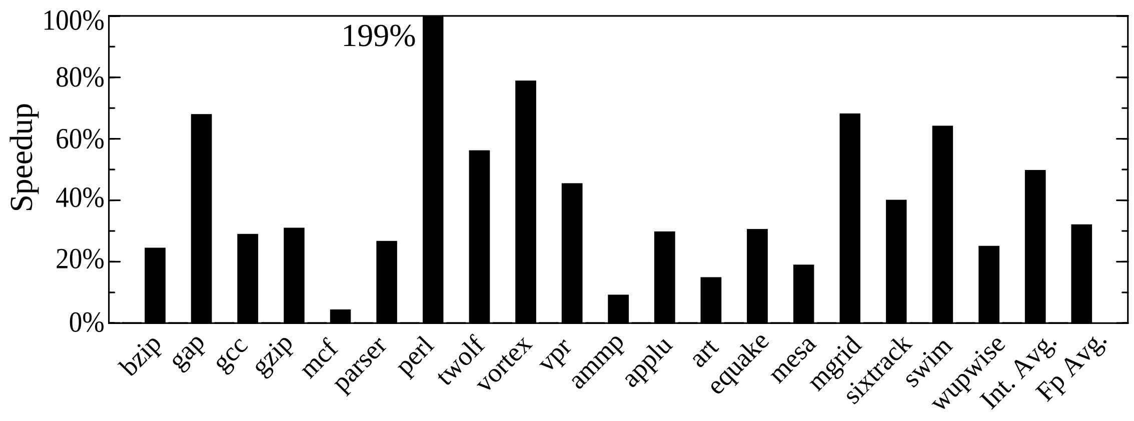


Fig. 26. Performance comparison of O-IMT normalized to the superscalar with hardware resources as a single SMT context.

when the number of contexts is not a limitation. Floating-points benchmarks achieve less impressive speed-up than Integers, but they still achieve an average of 32% speed-up over SMT. This result shows that the programs with characteristics of *gap, perl, twolf, vortex, mgrid,* and *swim* can greatly benefit from IMT architectures compared to genuine SMT.

7 CONCLUSIONS

SMT has emerged as a promising architecture to share a wide-issue processor's data-path across multiple program executions. This dissertation proposed the Implicitly-Multi-Threaded (IMT) processor to utilize SMT's support for multithreading to execute compiler-specified speculative threads from a single sequential program. The dissertation presented a case arguing that a naive mapping of even highly-optimized threads onto SMT performs only comparably to an aggressive superscalar. I proposed a naive IMT (N-IMT) that incurs high thread execution overhead because it indiscriminately divides SMT's shared pipeline resources (e.g., as fetch bandwidth, issue queue, load/store queues, and physical registers) across threads without regard to resource availability, thread resource usage, or inter-thread dependence.

I also proposed an optimized IMT (O-IMT) that employs three primary mechanisms and two secondary optimizations to improve speculative thread execution efficiency in an SMT pipeline. The three primary optimizations are as follows: (1) O-IMT employs a novel resource- and dependence-based fetch policy to decide from which thread to fetch every cycle. (2) O-IMT multiplexes contexts by mapping as many threads as allowed by the hardware resources, increasing instruction overlap. (3) Speculatively-threaded architectures incur rename table set-up overhead at thread start-up to ensure proper register value communication between earlier threads and the newly invoked thread. O-IMT virtually eliminates the rename table set-up overhead incurred in speculatively-threaded architectures by overlapping the start-up delay with previous threads' execution. The secondary optimizations are as follows: (1) O-IMT speculatively releases register values to avoid the implementation and performance issues of N-IMT's thread-level squashing, and (2) O-IMT employs two-phase commit to reduce register pressure by freeing some registers at instruction commit, before the thread commits. As SMT and speculative threading become prevalent, O-IMT's optimizations will be necessary to achieve high performance.

Using results from execution-driven simulation and SPEC2K benchmarks with reference input sets, I showed that O-IMT improves performance by an average of 20% and 29% with a maximum of 69% and 65% for integer and floating-point benchmarks, respectively, over an aggressive superscalar. I also presented performance comparisons against two prior proposals, TME and DMT, which execute speculative threading on SMT-based architectures. I showed that O-IMT outperforms a comparable TME by 26% and a comparable DMT by 38%.

In the future, the circuit technology will continue to scale the transistor size but not the wire delay. This trend means that there will be billions of transistors in a chip, but the wire delay will severely limit the design options for utilizing transistors. The CMP could be an answer for that situation. However, one open question is what the design options for each core within a CMP will be. Will it be an SMT? Or will it be a regular wide-issue superscalar? How can IMT be applied to those design options?

If each core will be an SMT, all of IMT's architectural optimizations will be directly applicable to the future designs as long as single-program performance remains an important issue. Even though there is no reason not to believe that each core within a CMP will be an SMT, let us assume that each core will be a regular superscalar. Under that assumption, before I can decide whether IMT will be effectively applicable there, I have to answer the following question. Can I apply context multiplexing to CMP? One of the biggest overheads of CMP-based speculative threading is the load imbalance. Context multiplexing will effectively remove this overhead with no doubt. However, unlike on SMT, context multiplexing on a CMP will complicate the cache design. Even without context multiplexing, the cache design for CMP-based speculative threading is already too complicated. Context multiplexing on a CMP requires that each cache block should keep a thread identifier as a tag. It means not only increasing storage overhead in caches but also increasing the number of tag comparisons for cache accesses due to identifying thread ID.

If context multiplexing will be taken as a design point to reduce the load imbalance in spite of those issues, all IMT techniques will be effective for future designs. If not, only some of IMT's optimizations will be applicable, but others will not. Hiding thread start-up delay and speculative releasing will still benefit future designs. However, the R&D-based fetch policy will not benefit the CMP with superscalar cores, because there is no resource sharing across threads and so there is no need to steer the fetching unit to prioritize the best instructions to bring ahead. The two-phase commit will not benefit for the same reason, because reducing register pressure will not affect other threads' running in separate cores.

8 FUTURE DIRECTION

This dissertation proposed key microarchitectural optimizations to remove the inefficiencies of speculative threading on Simultaneous Multithreading (SMT), which is a centralized architecture. It will be interesting to see how these optimizations react with different architectures such as the Chip MultiProcessor (CMP), which is a distributed architecture. Multiscalar was the first proposal for speculative threading, and it uses the CMP as a hardware platform. Although I expect that O-IMT's optimizations will still be effective on CMP-based speculative machines, including the previously-proposed Multiscalar, different optimizations may be found to be more effective on CMP-based speculative machines. The comparison of the previously proposed Multiscalar with and without optimizations against O-IMT will be also interesting to see. However, the wire delay factor should be considered carefully for such a comparison because these two architectures use different hardware platforms and have different implications on wire delay.

Recently, there have been commercial products that support only in-order execution but with multithreading, such as Intel Itanium [14] and Sun Niagara [26]. By not supporting out-of-order execution, these architectures have the advantage of simplifying pipeline components such as the issue queue. With faster CPU clocks and wider pipelines, all relevant microarchitectural components should scale accordingly. Otherwise, the processor performance will show little improvement despite the faster clock and wider pipeline. Unfortunately, such scaling becomes extremely difficult with the faster clock and wider pipeline. Therefore, the advantage of reducing the design complexity of the pipeline is appealing.

However, when supporting in-order execution only, these architectures have the disadvantage of losing instruction-level parallelism (ILP) and consequently performance. These in-order-execution cores maintain the performance by increasing the pipeline throughput through thread-level parallelism (TLP). Meanwhile, single-thread performance suffers significantly, reducing the attractiveness of the in-order execution with multithreading. The performance of speculative threading when applied to the multithreading in-order execution cores with and without O-IMT's optimizations will be an interesting topic. The O-IMT’s optimizations could improve single-thread performance in multi-threading in-order cores enough to overcome the performance deficit due to the in-order execution.

O-IMT’s context multiplexing enables superscalar to run speculative threading by dynamically allocating the active list entries and load/store queue entries for multiple threads. In this dissertation, I showed that O-IMT’s context multiplexing improves performance even when only one context is available. The optimizations proposed in this dissertation can be changed or extended to accommodate speculative-threading execution even on conventional superscalar pipelines.

REFERENCES

[1] H. Akkary and M. A. Driscoll. A dynamic multithreading processor. In *Proceedings of the 31st International Symposium on Microarchitecture*, pages 226–236, Dec. 1998.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The tera computer system. In *In International Conference on Supercomputing*, pages 1–6, June 1990.

[3] S. E. Breach. Design and evaluation of a multiscalar processor. Technical Report 1393, Computer Sciences Department, University of Wisconsin–Madison, Feb. 1999.

[4] S. E. Breach, T. N. Vijaykuamar, and G. S. Sohi. The anatomy of the register file in a multiscalar processor. In *Proceedings of the 27th International Symposium on Microarchitecture,* pages 181–190, Nov. 1994.

[5] P.-Y. Chang, E. Hao, T.-Y. Yeh, , and Y. Patt. Branch classification: A new mechanism for improving branch predictor performance. In *Proceedings of the 27th International Symposium on Microarchitecture*, pages 22–31, Nov. 1994.

[6] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt. Simultaneous subordinate microthreading (ssmt). In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 186–195, Oct. 1999.

[7] C.-Y. Cher and T. N. Vijaykumar. Skipper: A microarchitecture for exploiting control-flow independence. In *Proceedings of the 34th International Symposium on Microarchitecture,* pages 4–15, Dec. 2001.

[8] M. Cintra, J. F. MartÌnez, and J. Torrellas. Architectural support for scalable speculative parallelization in shared-memory systems. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 13–24, June 2000.

[9] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17(5):12–19, Sept. 1997.

[10] M. Franklin and G. S. Sohi. The expandable split window paradigm for exploiting fine-grain parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 58–67, May 1992.

[11] M. Franklin and G. S. Sohi. ARB: A hardware mechnism for dynamic reordering of

memory references. *IEEE Transactions on Computers*, 45(5):552–571, May 1996.

[12] S. Gopal, T. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative versioning cache. In *4th Annual Symposium on High Performance Computer Architecture*, pages 195–205, Feb. 1998.

[13] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 58–69, Oct. 1998.

[14] Intel Corporation. *Intel Itanium Processor: Hardware Developer's Manual*, Aug. 2001.

[15] Intel Corporation. *Intel Pentium 4 and Intel Xeon Processr Optimization: Reference Manual*, Oct. 2002.

[16] J. Kahle. A dual-cpu processor chip. In *In Microprocessor Forum*, Oct. 1999.

[17] P. Marcuello and A. Gonzalez. Thread-spawning schemes for speculative multithreading. In *9th Annual Symposium on High Performance Computer Architecture*, pages 55–64, Feb. 2003.

[18] A. I. Moshovos, S. E. Breach, T. Vijaykumar, and G. S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 181–193, June 1997.

[19] C.-L. Ooi, S. W. Kim, I. Park, R. Eigenmann, B. Falsafi, and T. N. Vijaykumar. Multiplex: Unifying conventional and speculative thread-level parallelism on a chip multiprocessor. In *Proceedings of the 2001 International Conference on Supercomputing*, pages 368–380, June 2001.

[20] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 206–218, June 1997.

[21] E. Rotenberg, Q. Jacobson, , and J. E. Smith. A study of control independence in superscalar processors. In *5th Annual Symposium on High Performance Computer Architecture*, pages 115–124, Feb. 1999.

[22] A. Roth and G. S. Sohi. Speculative data-driven multithreading. In *7th Annual Symposium on High Performance Computer Architecture*, pages 20–24, Jan. 2001.

[23] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22th International Symposium on Computer Architecture*, pages 414–425,

June 1995.

[24] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 1–12, July 2000.

[25] J. G. Steffan and T. C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *4th Annual Symposium on High Performance Computer Architecture*, pages 2–13, February 1998.

[26] Stephen Shankland. *New chips put spark in Sun*. CNET News.com, http://zdnet.com.com/2100-1103-986048.html, Feb. 2003.

[27] M. Tremblay. An architecture for the new millennium. In *In Proceedings of the 1999 Hot Chips Symposium*, Aug. 1999.

[28] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, and J. L. Lo. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 13–24, May 1996.

[29] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22th International Symposium on Computer Architecture*, pages 392–403, June 1995.

[30] T. N. Vijaykumar and G. S. Sohi. Task selection for a multiscalar processor. In *Proceedings of the 31st International Symposium on Microarchitecture*, pages 81–92, Dec. 1998.

[31] S. Wallace, B. Calder, and D. M. Tullsen. Threaded multiple path execution. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 238–249, June 1998.

[32] C. Zilles and G. S. Sohi. Execution-based prediction using speculative slices. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 2–13, June 2001.

VITA

Name : Il Park
Phone: (O) 765-494-0617, (H) 765-409-4259
E-mail : parki@ecn.purdue.edu,
Home page : http://min.ecn.purdue.edu/~parki/
Address : 3118 Salem Courthouse Dr #2B. West Lafayette. IN. 47906

Education

1. PhD (1998 Fall ~ 2003 August) Dept of Electrical and Computer Engineering, Purdue University. West Lafayette. IN
   Primary Area: Computer Architecture
   Dissertation title : Implicitly Multithreaded (IMT) Processors
2. MS (1993 ~ 1994) Dept of Electrical Engineering, Pohang Institute Science and Technology (POSTECH). Pohang, KOREA.
   Primary Area: Automatic Control and Robotics
   Dissertation title : Hybrid algorithm for a mobile robot
3. BS (1989 ~ 1992) Dept of Electrical Engineering, POSTECH. Pohang, KOREA.
   Primary Area: Automatic Control
   Dissertation title: Micro-mouse robot with DC Motor-Control by 16-bit embedded controller

Research Experience

At Purdue University

1. Implicitly MultiThreaded (IMT) Processors (June 2000 ~)
   :Developing the new speculative architecture based on SMT architecture.

- SMT is designed to improve system's throughput. However, SMT has little benefit for running a single application that cannot be partitioned into independent multiple threads. IMT, which based on SMT architecture, speculatively runs multiple threads generated from a single program by a compiler. More Info: http://min.ecn.purdue.edu/~parki/Private/imt-final.pdf

2. Power Management in Register File (Jun 2001 ~ Jul 2002)
   :Reducing energy dissipation of register files in high-performance processors.

- The key issues for register file design in high-performance processors are access time and energy. While previous work has focused on reducing the number of registers, we propose to reduce the number of register ports through two proposals, one for reads and the other for writes. More Info: http://min.ecn.purdue.edu/~parki/Private/regfile-final.pdf

3. Purdue MULTIPLEX (Dec 1999 ~ May 2000)
   :Developed the new radical speculative architecture based on Chip Multi-Processor.
   More Info: http://dynamo.ecn.purdue.edu/~mux/
   http://min.ecn.purdue.edu/~parki/Private/ics01.pdf
4. Branch Prediction using Dynamic Decision Tree (DDT) (June 1999 ~ Nov 1999)
   :Developed a new branch prediction mechanism using Dynamic Decision Tree(DDT) that is branched from AI techniques.

At POSTECH

5. Steering algorithm for unmanned tank (Spring 1993 ~ Fall 1994)
   :Supported by Korea Agency for Defense &Development (ADD) and Hyundai Motor
   ; Was responsible for the implementation of the low-level controller.
6. Indoor mobile robot (Spring 1993 ~ Fall 1994)
   :Supported by Communication Research Center of Posco (Pohang Iron and Steel company); Was responsible for the implementation of the sensor interface and low-level controller.
7. Hybrid algorithm for a mobile robot (Spring 1993 ~ Winter 1994)
   ; Made a hardware platform (mobile robot), algorithms for steering and low-level controllers; Main OS was VrTx -- real-time OS. (Bus:VME, CPU: One 68030, Five 80C196KC)

Work Experience

1. Engineering Consultant (Assistant Manager). Taehan Telecom, Seoul, KOREA (Nov. 1996 ~ Jul. 1998); Work for the project "FSN(Full Service Network) Plan for a next communication network."; Proposed the network development plan for the 21C communication market in KOREA.
2. Research Engineer LG Electronics Company, Seoul, KOREA (Feb. 1995 ~ Oct. 1996)
   ; A member of the PDA (Personal Digital Assistance) development team; Managed the power management of the PDA; Designed ASIC for PMU (Power Management Unit); developed source code of micro controller and main CPU (SH3-Hitachi’s LISC chip) interacting with PMU.

Teaching Experience

1. Purdue : Teaching Assistant. West Lafayette, IN; Computer architecture course (Spring 2000 ~ Spring 2001);
2. POSTECH : Teaching Assistant. Pohang, KOREA; Automatic control course (Fall 1993);

Publications

1. Il Park, Babak Falsafi, and T. N. Vijaykumar, Implicitly Multithreaded Processors. In Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA 30), June 2003. http://min.ecn.purdue.edu/~parki/Private/imt-final.pdf
2. Il Park, Mike Powell, and T. N. Vijaykumar, Reducing Register Ports for Higher Speed and Lower Energy. In Proceedings of the 35th Annual International Symposium on Microarchitecture (MICRO 35), November 2002. http://min.ecn.purdue.edu/~parki/Private/regfile-final.pdf
3. Chong-Liang Ooi, Seon Wook Kim, Il Park, Rudolf Eigenmann, Babak Falsafi, and T. N. Vijaykumar, Multiplex: Unifying Conventional and Speculative Thread-Level Parallelism on a Chip Multiprocessor. In Proceedings of the International Conference on Supercomputing (ICS), June 2001. Also available as Technical Report 00-13, School of Electrical and Computer Engineering, Purdue University, October 2000. http://min.ecn.purdue.edu/~parki/Private/ics01.pdf
4. Il Park, Young D. Kwon, and Jin-Soo Lee, Hybrid algorithm for a mobile robot. Journal of Korea Automatic Control Assoc. Vol.33-B, No.7, Korea. July 1996.

Reference:

1. T. N. Vijaykumar: Assistant Professor. School of Electrical & Computer Eng. Purdue University. Email: vijay@ecn.purdue.edu; Tel: (765) 494-0592; Fax: (765) 494-6440
2. Babak Falsafi: Associate Professor. Electrical & Computer Eng. Computer Science. Carnegie Mellon University. Email: babak@cmu.edu; Tel: (412) 268-7047; FAX: (412) 268-6353
3. Rudolf Eigenmann: Associate Professor. School of Electrical & Computer Eng. Purdue University. Email: eigenman@ecn.purdue.edu; Tel: (765) 494-1741; Fax: (765) 494-6440