Implicitly-Multithreaded Processors

Il Park, Babak Falsafi[*] and T. N. Vijaykumar

School of Electrical & Computer Engineering
Purdue University
{parki,vijay}@ecn.purdue.edu
http://min.ecn.purdue.edu/~parki
http://www.ece.purdue.edu/~vijay

[*]Computer Architecture Laboratory (CALCM)
Carnegie Mellon University
babak@cmu.edu
http://www.ece.cmu.edu/~impetus

Abstract

This paper proposes the Implicitly-MultiThreaded (IMT) *architecture to execute compiler-specified speculative threads on to a modified Simultaneous Multithreading pipeline. IMT reduces hardware complexity by relying on the compiler to select suitable thread spawning points and orchestrate inter-thread register communication. To enhance IMT's effectiveness, this paper proposes three novel microarchitectural mechanisms: (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.*

We use SPEC2K benchmarks and cycle-accurate simulation to show that an microarchitecture-optimized IMT improves performance on average by 24% and at best by 69% over an aggressive superscalar. We also compare IMT to two prior proposals, TME and DMT, for speculative threading on an SMT using hardware-extracted threads. Our best IMT design outperforms a comparable TME and DMT on average by 26% and 38% respectively.

1 Introduction

Architects are now exploring thread-level parallelism to exploit the continuing improvements in CMOS technology to deliver higher performance. Simultaneous Multithreading (SMT) [13] has been proposed to improve system throughput by overlapping multiple (either multiprogrammed or explicitly parallel) threads on a single wide-issue processor. The proposed Alpha 21464, the recently-announced IBM Power5, and the HyperThreaded Pentium 4 currently in production [6] are examples of SMT processors. Recently, researchers have also advocated using SMT's threading support to improve a single sequential program's execution time. Examples of these proposals include Threaded Multipath Execution (TME) [15] and Dynamically MultiThreaded (DMT) processors [1].

In this paper, we propose the Implicitly-Multi-Threaded (IMT) processor. 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 — i.e., compiler-specified speculative threads [11] — can be decoupled from the processor organization — i.e., distributed processing cores. Multiscalar [11] employs sophisticated specialized hardware, the register ring and address resolution buffer, which are strongly coupled to the distributed core organization. In contrast, IMT proposes to map speculative threads on to generic SMT.

IMT differs fundamentally from prior proposals, TME and DMT, for speculative threading on SMT. While TME executes multiple threads only in the uncommon case of branch mispredictions, IMT invokes threads in the common case of correct predictions, thereby enhancing execution parallelism. Unlike IMT, DMT creates threads in hardware. Because of the lack of compile-time information, DMT uses value prediction to break data dependence across threads. Unfortunately, inaccurate value prediction incurs frequent misspeculation stalls, prohibiting DMT from extracting thread-level parallelism effectively. Moreover, selective recovery from misspeculation in DMT requires fast and frequent searches through prohibitively large (e.g., ~1000 entries) custom instruction trace buffers that are difficult to implement efficiently.

In this paper, we find that a naive mapping of compiler-specified speculative threads onto SMT performs poorly. Despite using an advanced compiler [14] 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 negatively offsets the advantages of speculative threading.

We also identify three key microarchitecture optimizations necessary to alleviate the inefficiencies in N-IMT, and address them in our proposal, called *Optimized IMT (O-IMT)*. These novel optimizations are:

- **Novel fetch policy to bring suitable instructions**: Because the choice of instruction fetch policy fundamentally impacts performance, O-IMT carefully controls fetch via a *resource- and dependence-based fetch policy*. We propose a highly accurate (~97%) dynamic resource predictor to gauge resource (e.g., physical registers) availability and avoid thread misspeculation due to lack of resources midway through execution. Moreover, we propose a *inter-thread dependence heuristic* to avoid delaying earlier threads' instructions in favor of fetching from later threads that are data-dependent on earlier threads. In contrast, TME, DMT, and N-IMT use variations of ICOUNT [13] 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 [13] 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 the number of instructions discarded per misspeculation [14]. To obviate the need for larger threads, O-IMT multiplexes the hardware contexts by mapping and simultaneously executing as many in-program-order threads onto a single context as allowed by the resources.
- **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 and newly-invoked threads. Many prior proposals for speculative threading (e.g., DMT and Multiscalar) do not explicitly address the overhead due to thread start-up delay. TME and N-IMT both account for this overhead and incur extra start-up delay prior to thread invocation. In contrast, O-IMT hides the delay by overlapping rename table set-up with previous threads' execution, because the compiler-specified inter-thread register dependence information is available well before the thread starts.

Using the SPEC2K benchmarks, we show that N-IMT actually degrades performance in integer benchmarks on average by 3%, and improves performance negligibly in floating-point benchmarks relative to a 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 a comparable superscalar. Our results also indicate that TME and DMT are on average not competitive relative to a comparable superscalar.

The rest of this paper is organized as follows. Section 2, briefly describes compiler-specified threading. Section 3, describes our proposals for N-IMT and O-IMT. In Section 4, we present experimental results. We discuss related work in Section 5, and conclude in Section 6.

2 Compiler-Specified Speculative Threads

Speculatively-threaded architectures may use hardware [1,7] or compiler [11,5,12,9] to partition a sequential program into threads. Architectures extracting speculative threads in hardware have the key advantage that they offer binary compatibility with superscalar. These architectures, however, may incur high thread speculation overhead because: (1) hardware has relatively limited scope in selecting suitable threads and thread spawning points, (2) hardware typically precludes thread-level code optimization, and (3) these architectures primarily rely on value prediction (with potentially low accuracy) to implement inter-thread communication.

Instead, IMT uses Multiscalar's compiler-specified speculative threads. The Multiscalar compiler employs several heuristics to optimize thread selection [14]. The compiler maximizes thread size while limiting the number of thread exit points to a pre-specified threshold. 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 paper.

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 live registers entering 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*; and (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. Figure 1 shows an example thread. An instruction that is the last write to an architectural register in all the possible control flow paths is annotated with *forward*

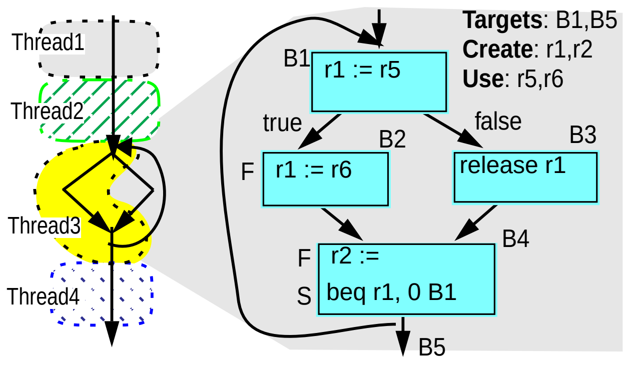

Figure 1: Compiler-specified speculative threads.

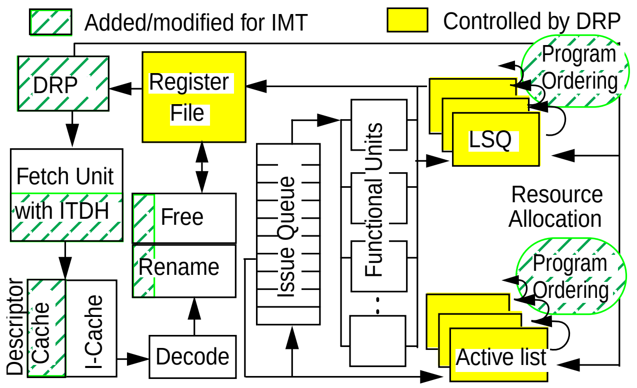

Figure 2: The anatomy of an IMT processor.

bits (labeled "F") and is referred to as a *forward* instruction. 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 it is in the path B1B3B4. To handle this case, the compiler inserts a *release* instruction in B3. In Section 3.2, we explain how the hardware uses forward and release instructions to implement inter-thread register communication. Instructions that lead to a target are annotated with *stop* bits (labeled "S"), signaling the end of the thread.

3 Implicitly-Multithreaded Processors

We propose the Implicitly-MultiThreaded (IMT) processor to utilize SMT's support for multithreading by executing speculative threads. Figure 2 depicts the anatomy of an IMT processor derived from SMT. IMT uses the rename tables for register renaming, the issue queue for out-of-order scheduling, the per-context load/store queue (LSQ) and active list for memory dependences and instruction reordering prior to commit. As in SMT, IMT shares the functional units, physical registers, issue queue, and memory hierarchy among all contexts.

IMT exploits *implicit* parallelism, as opposed to programmer-specified, *explicit* parallelism exploited by conventional SMT and multiprocessors. Like Multiscalar, IMT predicts the threads in succession and maps them to execution resources, with the earliest thread as the *non-speculative* (head) thread, followed by subsequent *speculative* threads [11]. IMT honors the inter-thread control-flow and register dependences specified by the compiler. IMT uses the LSQ to enforce inter-thread memory dependences. Upon completion, IMT commits the threads in program order.

We present two IMT variations: (1) a *Naive IMT (N-IMT)* that performs comparably to an aggressive superscalar, and (2) an *Optimized IMT (O-IMT)* that uses novel microarchitectural techniques to enhance performance.

3.1 Thread Invocation

Like Multiscalar, both IMT variants invoke threads in program order by predicting the next thread from among the targets of the previous thread (specified by the thread descriptor) using a thread predictor. A descriptor cache (Figure 2) stores recently-fetched thread descriptors. Although threads are invoked in program order, IMT may fetch later threads' instructions out of order prior to fetching all of earlier threads' instructions, thereby interleaving instructions from multiple threads. To decide which thread to fetch from, IMT consults the fetch policy.

3.1.1 Resource Allocation & Fetch Policy

Our base IMT processor, N-IMT, uses an unmodified ICOUNT policy [13], in which the thread with the least number of instructions in flight 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.

We also 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 LSQ 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 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 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.

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 exceed the predicted demand by the thread. The DRP dynamically monitors the threads activity and allows fetch to be initiated from newly invoked threads when earlier threads commit and resources become available.

Figure 3 (a) depicts an example of DRP. O-IMT indexes into a table using the start PC of a thread. Each table entry holds the numbers of active list and LSQ slots, and physical registers used by the thread’s last four execution instances. The pipeline monitors a thread’s resource needs, and upon thread commit, updates the thread’s DRP entry. DRP supplies the maximum among the four instances for each resource as the prediction for the next instance’s resource requirement. In Section 4.2, we show results indicating that overestimating resource usage using the maximum value works well in practice due to low variation in resource needs across nearby instances of a thread.

O-IMT’s 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 use variants of conventional fetch policies. TME uses biased-ICOUNT, a variant of ICOUNT that does not consider resource availability and thread-level independence. DMT’s fetch policy statically partitions two fetch ports, and allocates one port for the non-speculative thread and the other for speculative threads in a round-robin manner. However, DMT does not suffer from resource contention because the design assumes prohibitively large custom instruction trace buffers (holding thousands of instructions) allowing for threads to make forward progress without regards to resource availability and thread-level independence. Unfortunately, frequent associative searches through such large buffers are slow and impractical.

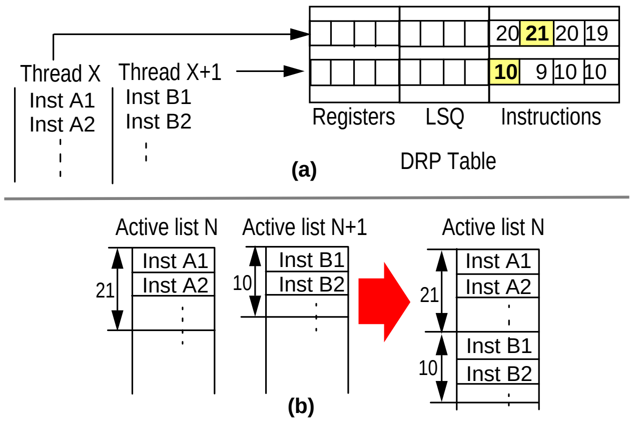

Figure 3: Using DRP (a) and context multiplexing (b).

3.1.2 Multiplexing Hardware Contexts

Much like prior proposals, N-IMT assigns a single thread to a hardware context. Because many programs have short threads [14] 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 likelihood of dependence misspeculation [14] and the number of instructions discarded per misspeculation, and cause speculative buffer overflow [5].

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 by the resources in one context (typically 3-6 threads for SPEC2K). Context multiplexing requires for each context only an additional fetch PC register and rename table pointer per thread for a given maximum number of threads per context. Context multiplexing differs from prior proposals for mapping multiple threads on to a single processing core [12,3] to alleviate load imbalance, in that multiplexing allows instructions from multiple threads within a context to execute and share resources *simultaneously.*

Two design complexities arise due to sharing resources in context multiplexing. First, conventional active list and LSQ designs assume that instructions enter these queues in (the predicted) program order. Such an 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 accessing thread, and then (conceptually) repeat the search among entries from the thread preceding the accessing thread, working towards older threads. Unfortunately, the active list and LSQ cannot afford these additional design complications because active lists are made large and therefore non-searchable by design and the LSQ's ordered, associative search is already complex and time-critical.

Second, allowing a single context to have multiple *out-of-program-order* threads complicates managing inter-thread dependence. Because two in-program-order threads may be mapped to different contexts, honoring memory dependences would require memory accesses to search through multiple contexts thereby prohibitively increasing LSQ search time and design complexity.

Using DRP, O-IMT avoids the first design complexity by placing instructions in the active list and LSQ in program order. O-IMT keeps instructions in both structures in program order while fetching instructions out of order, by using DRP's resource demand estimates for a thread and creating a gap (as in [2]) in the active list and LSQ for the thread's yet-to-be-fetched instructions. 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 thread execution 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. As such, DRP helps dynamically partition a context's active list and LSQ so that instructions from one thread do not interfere with those of other threads within the context.

O-IMT avoids the second design complexity by mapping threads to a context in program order. Inter-thread and intra-thread dependences within a single context are treated similarly. Figure 3 (b) shows how in-program-order 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.

3.2 Register Renaming

Superscalar's register rename table relies on in-order instruction fetch to link register value producers to consumers. IMT processors' out-of-order fetch raises two issues in linking producers in earlier threads to consumers in later threads. First, IMT has to ensure that the rename maps for earlier threads' source registers are not clobbered by later threads. Second, IMT must guarantee that later threads' consumer instructions obtain the correct rename maps and wait for the yet-to-be-fetched earlier threads' producer instructions. While others [1,7] employ hardware-intensive value prediction to address these issues potentially incurring frequent misspeculation and recovery overhead, IMT uses the create and use masks (Section 2) combined with conventional SMT rename tables.

Both IMT variants address these issues as follows. Upon thread start-up (and prior to instruction fetch), the processor copies the rename maps of the registers in create and use masks from a *master rename table*, to a thread's *local rename table.*[1] To allow for invoking subsequent threads, the processor pre-allocates physical registers and pre-assigns mappings for all the create-mask registers in a *pre-assign rename table.* Finally, the processor updates the master table with the pre-assigned mappings and marks them as *busy* to reflect the yet-to-be-created register values. Therefore, upon thread invocation the master table correctly reflects the register mappings that a thread should either use or wait for.

Instructions use the local table both to get their source rename maps and to put their destination rename maps. Instructions that produce and consume values (locally) within a thread allocate new mappings in the local table. Instructions that are data-dependent on earlier-threads' instructions wait until the corresponding pre-assigned physical register is *ready.* Forward and release instructions (Section 2) wake up waiting instructions in subsequent threads through the pre-assigned physical registers; forward instructions write their results in the pre-assigned physical registers, and release instructions copy values from the physical registers given by the local table to the pre-assigned physical registers. By copying the create mask maps at thread start-up, 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.

3.2.1 Hiding the Thread Start-up Delay

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. This waiting diminishes the full benefit of the fetch policy and context multiplexing. Updating the local, master and pre-assign tables must complete before a thread's instructions can be renamed. The updating rate of rename tables is limited by the table bandwidth. In conventional pipelines, this bandwidth

1. Conventional superscalar pipelines similarly checkpoint rename tables upon branch prediction to accelerate misprediction recovery.

matches the pipeline width and is sufficient for the peak demand. In contrast, IMT's requirement of updating the tables creates a burst demand that may exceed the bandwidth and may take several (e.g., 2-4) cycles to complete.

Our base IMT processor, N-IMT, incurs the thread start-up overhead immediately prior to fetching instructions. O-IMT, however, prevents the bandwidth constraint from delaying thread start-up. 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 given cycle only six instructions are renamed but the rename tables have the bandwidth to rename eight instructions, O-IMT uses the unused bandwidth to modify the tables. Thus, O-IMT overlaps a thread's start-up with previous threads's execution, hiding the thread start-up delay.

Thread start-up delay also exists in Multiscalar, TME, and 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.

3.3 Load/Store Queues

N-IMT imposes program order in the LSQs to enforce memory dependences within and across threads. A thread's memory search its context's LSQ to honor memory dependences. If there is no match in the local LSQ, accesses proceed to search other contexts' LSQs. 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 premature loads. Thus, N-IMT uses the LSQ to achieve the same functionality as ARB's [4].

Searching other contexts' LSQs takes extra cycles which may impact load hit latency. In addition, this searching makes the hit latency variable, which may complicate early scheduling of instructions dependent on the load. Fortunately, the non-speculative thread's loads, which are the most critical accesses, do not incur any extra searching, and hence, do not have variable hit latency problems. In speculative threads, IMT schedules load-dependent instructions only after loads finish searching. Thus, IMT gives up early scheduling of load-dependent instructions to avoid scheduling complications. The latency incurred by speculative threads' loads and their dependent instructions is hidden under instruction-level and thread-level parallelism. Upon a memory dependence violation, IMT squashes the offending threads. IMT uses memory dependence synchronization [8] — e.g., squash buffer [11] — to avoid frequent dependence violation.

Table 1: System configuration parameters.

Processing Units		**System**	
Issue width	8	**DRP table**	64 entries (3 x 256 bytes)
Issue queue	64 entries		
Number of contexts	8	**ITDH**	3 program counters
Branch unit	hybrid GAg & PAg 4K-entries each,	**L1 cache 2-port i-cache & 4-port d-cache**	64K 2-way, pipelined 2-cycle hit, 32-byte block
BTB	1K-entry 4-way		
Miss Penalty	7 cycles		
Functional units	8 integer, 8 pipelined 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, 4 ports	**Thread desc. cache**	16K 2-way, 2-cycle hit

4 Results

We have built a cycle-accurate simulator of an out-of-order SMT pipeline with extensions to evaluate a base superscalar processor (using a single SMT context), and the three speculatively-threaded processors, IMT, DMT, and TME. We use the Multiscalar compiler [14] 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 we assume for this study. Our base pipeline assumes an eight-wide issue out-of-order SMT with eight hardware contexts. The pipeline assumes two i-cache ports and the branch predictor allows up to two predictions per context per cycle. In addition to the base pipeline, O-IMT also uses a 64-entry DRP table and a 3-entry ITDH table to optimize fetch.

To gauge speculative threading's potential conservatively, we compare IMT's performance against an aggressive superscalar implementation that assumes the same resources available to a single context within the SMT

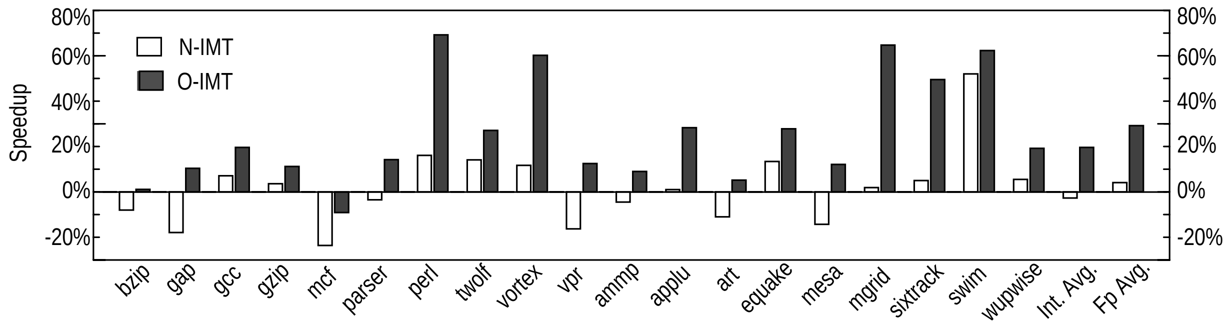

Figure 4: Performance comparison of N-IMT and O-IMT normalized to the baseline superscalar.

pipeline including the high-bandwidth branch prediction and fetch, and the large register file. We also assume a large active list of 1024 entries, because active lists are FIFO structures and are inherently scalable.

Table 2 shows the SPEC2K applications we use in this study, and the branch prediction accuracy and superscalar IPC we achieve per application. We use the reference input set in all of the benchmarks. To allow for practical simulation turnaround times, our simulator skips the first 3 billion instructions before simulating a total of 500 million instructions (plus overhead instructions, in IMT's case). We use total number of cycles as our 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. We first compare the performance of N-IMT and O-IMT to superscalar, and break down the performance bottlenecks O-IMT optimizes. Then we present results on the effectiveness of O-IMT's microarchitectural optimizations. Then we present O-IMT's ability to increase issue queue and LSQ efficiency as compared to superscalar using thread-level parallelism. Finally, we compare and contrast O-IMT with TME and DMT, two prior proposals for speculative threading using SMT hardware.

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			

4.1 Base System Results

Figure 4 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 our 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 on average by 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 on average 20% and 29% for integer and floating-point benchmarks respectively.

Figure 5 compares the key sources of execution overhead in superscalar, N-IMT and O-IMT. The breakdown includes the overhead of squashing instructions due to branch misprediction (both within and across threads) and resource pressure (in N-IMT and O-IMT), register data dependence stalls, memory waiting stalls (due to data cache misses), underutilized instruction fetch bandwidth, and runtime instruction overhead for IMT machines.

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.

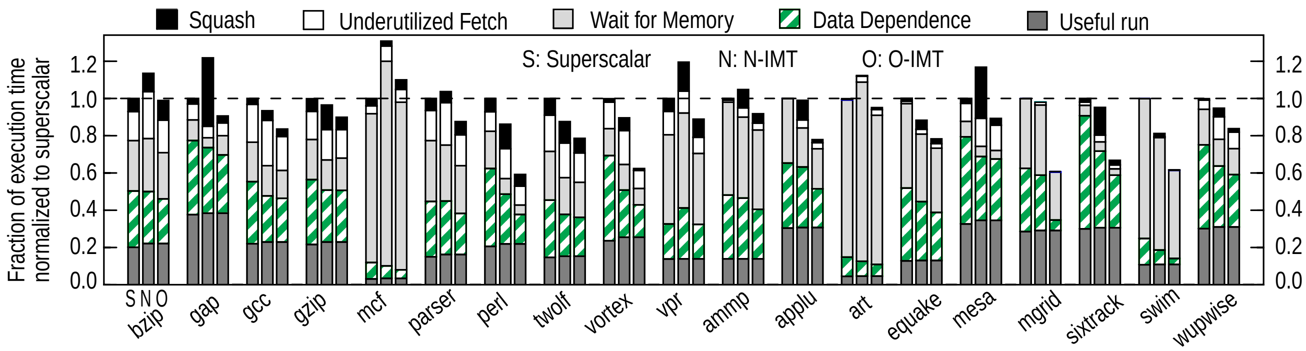

Figure 5: Breakdown of execution into instruction execution and pipeline stalls.

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 to 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 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 4.2 presents detail analysis on these techniques' contributions to O-IMT's performance.

4.2 Optimizing Thread-Level Speculation

Resource Allocation & Prediction. Figure 6 illustrates the need for dynamic resource allocation, and the impact of DRP's accurate prediction on performance in O-IMT. The figure compares performance under dynamic partitioning using DRP against static partitioning for LSQ entries (left) and the register file (right). 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. The graphs plot average performance (for integer and floating-point benchmarks) 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. In these graphs, we plot the optimal number of threads (denoted by the letter T) for every design point on the x-axis.

The graphs on the left indicate that DRP successfully eliminates all stalls related to a limited 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,

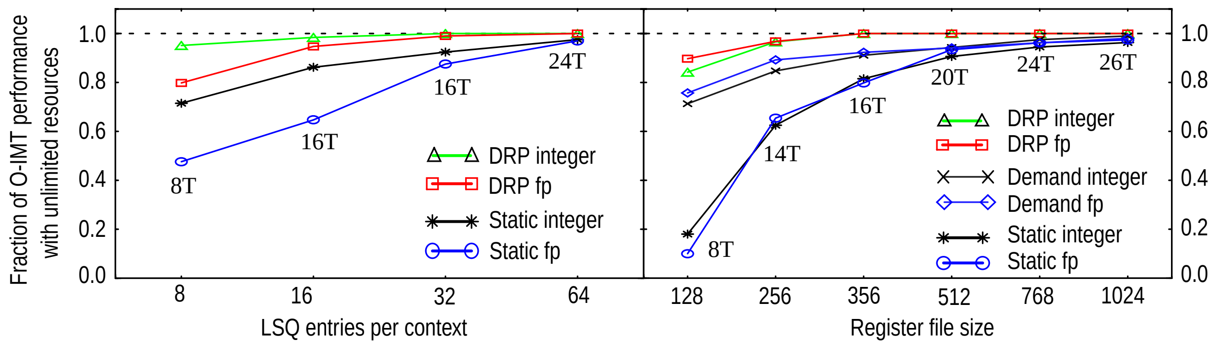

Figure 6: Dynamic vs. static resource partitioning.

reducing performance on average by 40% given 16 entries per context.

The graphs on the right indicate 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. Our 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 3.1). The table depicts the fraction of the time and the amount by which our 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.

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

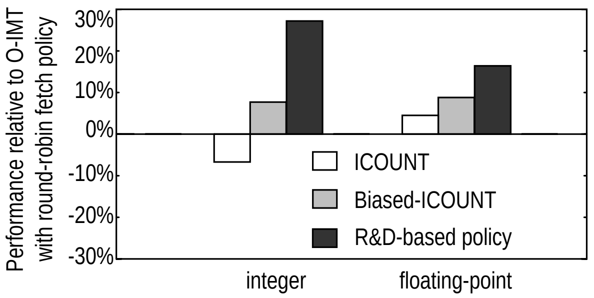

Figure 7: The impact of fetch policy.

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 [13] (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 [15] (used in TME) may improve performance only slightly when resource usage and dependence across threads drastically vary.

Figure 7 shows O-IMT's performance under four different fetch policies. The figure plots three priority-based fetch policies, ICOUNT, biased-ICOUNT, and resource- and dependence-based fetch policy. The graphs plot the average performance improvement for integer and floating-point benchmarks. The figure indicates that indeed in integer benchmarks, ICOUNT reduces performance on average over round-robin, because it allows speculative threads issuing at a high rate to inadvertently fetch, allocate resources, and subsequently squash. Biased-ICOUNT addresses this shortcoming in ICOUNT by biasing the priority towards the non-speculative thread by a constant 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.

The figure also indicates that the floating-point benchmarks actually slightly benefit from ICOUNT and biased-ICOUNT. The floating-point applications exhibit a high fraction of thread-level parallelism and independence across threads. As in SMT, ICOUNT allows for the

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

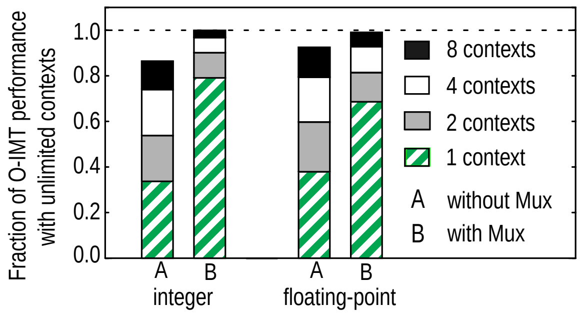

Figure 8: The impact of context multiplexing.

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.

Context Multiplexing. 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 8 illustrates the impact of multiplexing on O-IMT's performance. To accurately gauge the overall impact on performance with an increase in available resources, we also vary the register file size 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 on average reach best achievable performance even with eight hardware contexts. Moreover, performance substantially degrades (to as low as 35% in integer applications) when reducing the number of contexts.

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 [6]) to outperform eight contexts without multiplexing. Multiplexing also benefits floating-point benchmarks, reducing the required number of contexts. Floating-point benchmarks' performance, however, scales well with an increase in the number of contexts even without multiplexing due to these benchmarks' long-running threads.

Hiding the Thread Start-up Delay. Figure 9 illustrates the impact of thread start-up delay on O-IMT's performance. The graphs represent performance for start-up latency of two and four cycles as a fraction of that in an ideal system with no start-up delay. The figure indicates that a higher start-up delay of four cycles on average can reduce performance by 9% in integer benchmarks.

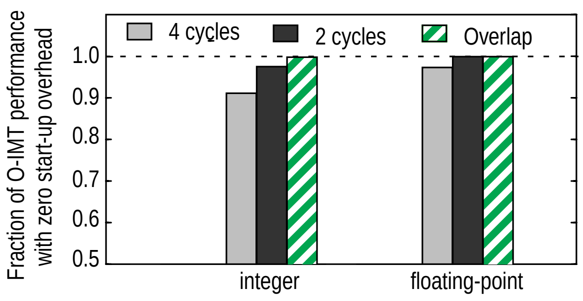

Figure 9: The impact of start-up delay.

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 overhead).

4.3 Issue Queue & LSQ Performance Sensitivity

In SMT/superscalar pipelines, the issue queue and LSQ(s) sizes are often the key impediments to performance scalability [10]. 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 10 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 issue queue/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

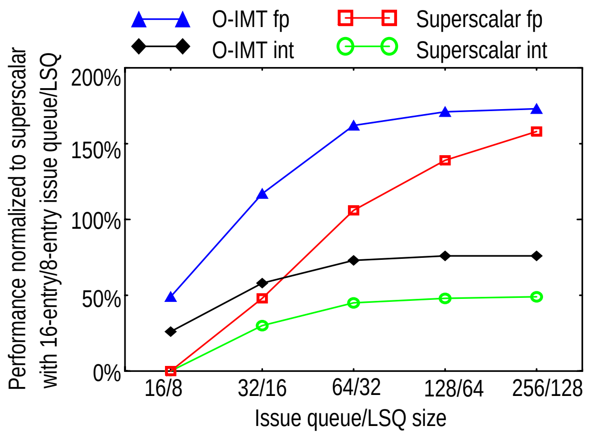

Figure 10: Issue queue/LSQ sensitivity.

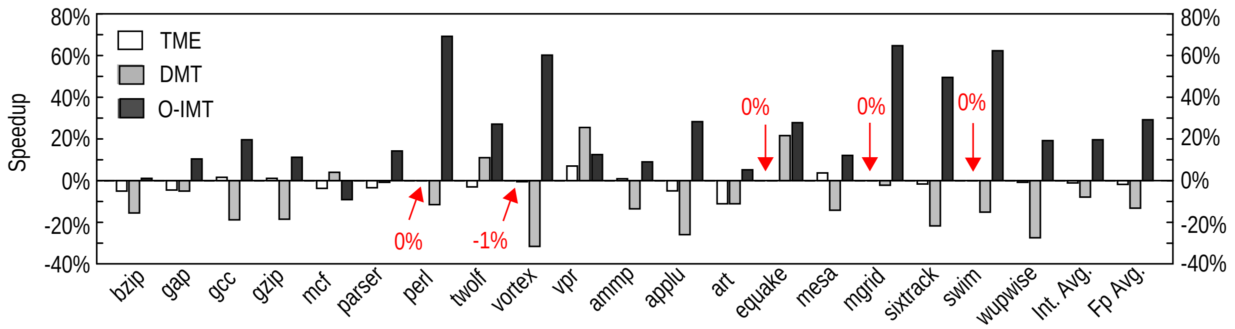

Figure 11: Performance comparison of TME, DMT, and IMT normalized to baseline superscalar.

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/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 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.

4.4 Comparison to TME & DMT

In this section, we compare O-IMT's performance against TME and DMT. Our models for TME and DMT are quite aggressive allowing for a conservative comparison against these machines. We assume no contention for TME's mapping synchronization bus [23]. To favor DMT, we assume that DMT has a 256-entry custom trace buffer per context (for a total of 2048 entries) with zero-cycle access, zero-cycle thread spawning, and selective recovery (squash) with zero-cycle penalty. 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. We 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. We compare these improved models against the original proposals.

Figure 11 compares speedups of our optimized TME and DMT machines, against O-IMT normalized to our baseline superscalar. Unlike O-IMT, TME and DMT reduce performance on average with respect to a comparable superscalar. TME [15] 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 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, our baseline superscalar machine. *Vpr* and *mesa* are benchmark examples in which the confidence predictor predicts accurately, allowing TME to improve performance over 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. Our 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 a significantly lower branch prediction accuracy as compared to our baseline superscalar, resulting in a lower average performance than superscalar.

5 Conclusions

SMT has emerged as a promising architecture to share a wide-issue processor's datapath across multiple program executions. This paper proposed the IMT processor to utilize SMT's support for multithreading to execute compiler-specified speculative threads from a single sequential program. The paper presented a case arguing that a naive mapping of even highly-optimized threads onto SMT performs only comparably to an aggressive superscalar. N-IMT incurs high thread execution overhead because it indiscriminately divides SMT's shared pipeline resources (e.g., as fetch bandwidth, issue queue, LSQs, and physical registers) across threads independently of resource availability, thread resource usage, and inter-thread dependence.

This paper also proposed O-IMT, an IMT variant employing three key mechanisms to improve speculative thread execution efficiency in an SMT pipeline. (1) a novel resource- and dependence-based fetch policy to decide which thread to fetch from every cycle. (2) context multiplexing to map as many threads to a single hardware context as allowed by hardware resources, and (3) a mechanism to virtually eliminate the thread start-up overhead of setting up rename tables (to ensure proper register value communication between earlier threads and the newly invoked thread). As SMT and speculative threading become prevalent, O-IMT's optimizations will be necessary to achieve high performance.

Using results from cycle-accurate simulation and SPEC2K benchmarks we showed that O-IMT improves performance by 24% over an aggressive superscalar. We also presented performance comparisons against two prior proposals for speculative threading on SMT, and showed that O-IMT outperforms a comparable TME by 26% and a comparable DMT by 38%.

Acknowledgements

We would like to thank Haitham Akkary, the members of CMU Impetus group, and the anonymous reviewers for feedback on earlier drafts of this paper. This work is supported in part by grants from IBM and Intel, an NSF Next-Generation Software grant, NSF CAREER awards, and an NSF Instrumentation grant.

References

[1] Haitham Akkary and Michael A. Driscoll. A dynamic multithreading processor. In *Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 31)*, pages 226–236, December 1998.

[2] Chen-Yong Cher and T. N. Vijaykumar. Skipper: A microarchitecture for exploiting control-flow independence. December 2001.

[3] Marcelo Cintra, Jose F. Martinez, and Josep Torrellas. Architectural support for scalable speculative parallelization in shared-memory systems. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 13–24, June 2000.

[4] Manoj Franklin and Gurindar S. Sohi. ARB: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers*, 45(5):552–571, May 1996.

[5] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, October 1998.

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

[7] Pedro Marcuello and Antonio Gonzalez. Thread-spawning schemes for speculative multithreading. February 2003.

[8] Andreas Moshovos, Scott E. Breach, and T. N. Vijaykumar. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[9] 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 2001 International Conference on Supercomputing*, June 2001.

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

[11] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

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

[13] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, and Jack 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.

[14] T. N. Vijaykumar and Gurindar S. Sohi. Task selection for a multiscalar processor. In *Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 31)*, December 1998.

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