Balanced Multithreading: Increasing Throughput via a Low Cost Multithreading Hierarchy

Eric Tune Rakesh Kumar Dean M. Tullsen Brad Calder
Computer Science and Engineering Department
University of California at San Diego
{etune,rakumar,tullsen,calder}@cs.ucsd.edu

Abstract

A simultaneous multithreading (SMT) processor can issue instructions from several threads every cycle, allowing it to effectively hide various instruction latencies; this effect increases with the number of simultaneous contexts supported. However, each added context on an SMT processor incurs a cost in complexity, which may lead to an increase in pipeline length or a decrease in the maximum clock rate. This paper presents new designs for multithreaded processors which combine a conservative SMT implementation with a coarse-grained multithreading capability. By presenting more virtual contexts to the operating system and user than are supported in the core pipeline, the new designs can take advantage of the memory parallelism present in workloads with many threads, while avoiding the performance penalties inherent in a many-context SMT processor design. A design with 4 virtual contexts, but which is based on a 2-context SMT processor core, gains an additional 26% throughput when 4 threads are run together.

1 Introduction

The ratio between main memory access time and core clock rates continues to grow. As a result, a processor pipeline may be idle during much of a programs execution. A multithreading processor can maintain a high throughput despite a large relative memory latencies by executing instructions from several programs. Many models of multithreading have been proposed. They can be categorized by how close together in time instructions from different threads may be executed, which affects how the state for different threads must be managed. Simultaneous Multithreading [31, 30, 12, 33] (SMT) is the least restrictive model, in that instructions from multiple threads can execute in the same cycle. This flexibility allows an SMT processor to hide stalls in one thread by executing instructions from other threads. However, the flexibility of SMT comes at a cost. The register file and rename tables must be enlarged to accommodate the architectural registers of the additional threads. This in turn can increase the clock cycle time and/or the depth of the pipeline.

Coarse-grained multithreading (CGMT) [1, 21, 26] is a more restrictive model where the processor can only execute instructions from one thread at a time, but where it can switch to a new thread after a short delay. This makes CGMT suited for hiding longer delays. Soon, general-purpose microprocessors will be experiencing delays to main memory of 500 or more cycles. This means that a context switch in response to a memory access can take tens of cycles and still provide a considerable performance benefit. Previous CGMT designs relied on a larger register file to allow fast context switches, which would likely slow down current pipeline designs and interfere with register renaming. Instead, we describe a new implementation of CGMT which does not affect the size or design of the register file or renaming table.

We find that CGMT alone, triggered only by main-memory accesses, provides unimpressive increases in performance because it cannot hide the effect of shorter stalls in a single thread. However, CGMT and SMT complement each other very well. A design which combines both types of multithreading provides a balance between support for hiding long and short stalls, and a balance between high throughput and high single-thread performance. We call this combination of techniques *Balanced Multithreading* (BMT).

This combination of multithreading models can be compared to a cache hierarchy, which results in a *multithreading hierarchy*. The lowest level of multithreading (SMT) is small (fewer contexts), fast, expensive, and closely tied to the processor cycle time. The next level of multithreading (CGMT) is slower, potentially larger (fewer limits to the number of contexts that can be supported), cheaper, and has no impact on processor cycle time or pipeline depth.

In our design, the operating system sees more *virtual contexts* than are supported in the core pipeline. These virtual contexts are controlled by a mechanism to quickly switch between threads on long latency load misses. The method we

Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37 2004)
1072-4451/04 $20.00 © 2004 IEEE

propose for adding more virtual contexts does not increase the size of the physical register file or of the renaming tables. Instead, inactive contexts reside in a separate memory dedicated to that purpose, which can be simpler and far from the core as compared to a register file, and will not be timing critical. Further, those threads that are swapped out of the processor core do not need to be renamed, which avoids an increase in the size of the renaming table. This architecture can achieve the throughput near that of a many-context SMT processor, but with the pipeline and clock rate of an SMT implementation that supports fewer threads. We find that we can increase the throughput of an SMT processor design by as much as 26% by applying these small changes to the processor core.

This paper is organized as follows: Section 2 discusses related prior work. Section 3 presents the architecture and mechanisms for combining SMT and CGMT. Section 4 discusses our evaluation methodology. Results are presented in Section 5.

2 Related Work

There has been a large body of work on the three primary multithreading execution models. Fine-grained multithreading architectures [24, 2, 10, 16] switch threads every processor cycle. Coarse-grained multithreading [1, 21, 26, 18, 5] (CGMT) architectures switch to a different thread if the current thread has a costly stall. Simultaneous multithreading [31, 30, 12, 33] (SMT) architectures can issue instructions from multiple threads simultaneously.

2.1 Coarse-Grain Multithreading

The Sparcle CPU [1] in the Alewife machine implements CGMT, performing a context switch in 14 cycles (4 cycles with aggressive optimizations). The Sparcle architects disabled the register windows present in the Sparc processor that they reused, and used the extra registers to support a second context. The Sparcle processor was in-order, with a short pipeline and did not perform register renaming. The IBM RS64 IV processor [5] supports CGMT with 2 threads, and is in-order. The RS64 designers chose to implement only two contexts, which avoided any cycle-time penalty from the additional registers. For the processors we seek to improve, which have large instruction windows backed by additional registers, the register file access time is much more likely to be on the critical timing path. Therefore, we present a different approach to context switching.

Waldspurger and Wiehl [32] avoid expanding the register file in a CGMT architecture by recompiling code so that each thread used fewer registers. Mowry and Ramkissoon [18] propose software-controlled CGMT to help tolerate the latency of shared data in a shared-memory multiprocessor. They suggest compiler-based register file partitioning to reduce context-switch overhead. Horowitz, *et al.* similarly suggest using memory references which cause cache misses to branch or trap to a user-level handler [13]. Our approach uses lightweight hardware support to make context switches faster than would be possible purely using software, and does not require recompilation.

2.2 Simultaneous Multithreading

Simultaneous multithreading can increase the utilization of the execution resources of a single processor core by executing instructions from different threads at the same time. However, each additional simultaneous thread expands structures whose speed may directly affect performance, in particular the register file. To reduce the incremental cost of additional threads in an SMT processor, Redstone, *et al.* [20] propose partitioning the architectural register file. Lo *et al.* [17] propose software-directed register deallocation to decrease dynamic register file demand for SMT processors. Both [20] and [17] require compiler support. Multi-level register file organizations reduce the average register access time [4, 8, 3].

Register file speed is a function of the number of ports, as well as the number of registers it contains. A processor with a high issue width requires a register file with many ports to avoid contention. The port requirements can be relaxed [19, 14, 27], but that requires additional hardware to arbitrate among the ports.

Tullsen and Brown [29] note that very long latency memory operations can create problems for an SMT processor. They suggest that when a thread is stalled waiting for a memory access, the instructions after the miss should be flushed from the pipeline, freeing critical shared execution resources. Our scheme inherently provides the same functionality. However, their proposal fails to free the most critical shared resource – thread contexts. We compare our processor designs against an SMT processor which implements their flushing mechanism. Our results show that freeing resources being held by a stalled thread is indeed very important; however, making those same resources available to a thread that would not otherwise have a chance to run is also important. Other researchers have suggested more sophisticated flushing policies for SMT [6], which we do not evaluate. However, improvements to policies which control when to flush an SMT processor can also be applied to controlling thread-swapping in a BMT processor.

3 Architecture

In this paper, we use the term *context* to refer to the hardware which gives a processor the ability to run a process without operating system or software intervention. We use the term *thread* to refer to a program assigned to a context by the

operating system. Because the BMT architecture we propose exposes more contexts to the operating system than can be active at once in the processor core, we distinguish between *physical contexts* and *virtual contexts*.

The number of physical contexts, denoted C_{phys}, is the number of threads which can have instructions in the pipeline simultaneously, and is limited by the register file and renaming table sizes. The number of *virtual contexts*, denoted C_{virt}, is the total number of threads which are supported at once, via CGMT. For an SMT-only processor, $C_{virt} = C_{phys}$. We refer to an SMT-only processor design as being an SMT-C processor design when it has C contexts. For example, the Pentium 4 is an SMT-2 processor. We refer to a Balanced Multithreading design with C_{phys} physical contexts and C_{virt} virtual contexts as a BMT-C_{phys}/C_{virt} processor.

Because there are more virtual contexts than physical contexts in a BMT processor, some threads will be *inactive* at any given time. An *inactive thread* can have a pending main memory request, but, unlike an *active thread*, an inactive thread does not have instructions in the pipeline nor does it have values in the primary register file.

3.1 Firmware Context Switching

We propose a context switching mechanism which (1) does not increase the size of the register file because architectural state of inactive threads is stored elsewhere, (2) does not increase the number of ports on the register file, because the save/restore instructions access the register file like ordinary instructions, (3) does not affect the design of the renaming table, because inactive threads have no instructions in the pipeline, and (4) is considerably faster than a software context switch by the operating system. This mechanism, which we call *firmware context switching*, uses:

1. an exception-like mechanism to initiate a context switch and to flush the pipeline,
2. a microcoded instruction sequence of special instructions to swap the register state of active and inactive threads.
3. a separate buffer to hold architectural registers of inactive threads,
4. a small amount of duplicated or additional hardware in areas that should not be critical to performance.

We now describe the features of firmware context switching in greater detail.

Detecting Load Misses and Flushing—When a load instruction needs to directly access main memory, a thread swap may be initiated. A firmware context switch is not fast enough to make thread switching profitable for loads which hit in a second or third level cache, given current cache latencies. We use a simple method to detect main memory accesses: if a load has an execution latency over a certain threshold, the load is assumed to be accessing main memory. When such a load is detected, it is *canceled*, but its memory request remains in the memory system. In the commit stage, the load instruction will raise an exception when it is the oldest instruction in its thread. Fetching from that thread stops, instructions from that thread are flushed, the PC of the cancelled instruction is saved, and the register map is restored to point to the proper state. However, instead of jumping to a trap handler, control is transferred to a microprogrammed instruction sequence.

One side effect of canceling an instruction, as we do with long latency loads, is that the possibility of livelock is introduced. Kubiatowicz gives a thorough treatment of these issues in [15]. To avoid livelock in our simulations, we require that a thread commit at least one instruction before it can be swapped out.

Microprogrammed Context Switch—After a thread has been flushed, instructions are fetched from a microcode control store. This microprogram consists of (1) a sequence of store-like `rsave` instructions, (2) a special thread-switch instruction, and (3) a sequence of load-like `rrestore` instructions. Each of the `rsave` and `rrestore` instructions is renamed, issued, and executed on an integer unit like a normal instruction. They are like a load or store instruction in they have one register operand, but they do not access programmer-visible memory space or undergo address translation. Instead they access a special buffer, the *Inactive Register Buffer* (IRB), which is described below. The address in the IRB is implicit given the operand and thread associated with an `rsave`/`rrestore` instruction. An unoptimized microprogram would have one `rsave` and one `rrestore` instruction for each architectural register.

We add two optimizations to this microcode sequence which reduce the number of instructions in a context switch. First, a *Dirty Register Mask* (DRM) tracks which architectural registers have been modified by committed instructions since the last thread swap. The microcode sequencer uses this bitmask to selectively generate `rsave` instructions only for registers which have been modified. The correct value of unmodified registers is still in the IRB. For the short times that threads are often swapped in, this can significantly reduce the number of `rsave` instructions. Second, for those benchmarks which never use floating point registers, the floating point registers are not restored. Operating systems already use this technique to shorten software context switches. Both techniques shorten the time to swap threads and reduce contention for functional units with other active threads.

Duplicated Hardware—While registers are saved and restored on a context switch, some small bits of hardware can simply be replicated for each virtual context. These include the branch global history register, the return stack, and processor control registers, such as the page-table base register and floating-point control register. Each of these resources, which we expect are not likely to be on a critical circuit path,

would need to be accessed through a multiplexer which would be controlled by a physical-to-virtual context mapping register. The special thread-switch instruction changes this register to correspond to the next thread to run.

Selecting the Next Thread—The next thread to swap in is known before a thread swap occurs. We use a Least-Recently-Run policy for selecting the next thread. When an active thread is swapped out of the pipeline, the least recently run thread is swapped in.

When a thread incurs a miss, but all inactive threads are also waiting for memory, we found that a good policy was to swap out the stalled thread, swap in the least recently run thread, but gate (stall) fetch for that least recently run thread until its data is returned from memory. This prevents the still-stalled thread from introducing instructions into the processor that will interfere with other active threads. Eickemeyer, *et al.*, [9], refer to this policy as *switch-when-ready* in their evaluation of a CGMT-only processor.

Inactive Register Buffer—Adding physical contexts to a processor increases the total number of registers in the register file, which is likely to affect the clock rate or pipeline length. The access requirements for active and inactive registers are quite different. As a result of these differences, the design constraints on the IRB are considerably relaxed, compared to the register file. (We will use the term *primary register file* to emphasize that we are not referring to the IRB.) For a 4-wide processor design, the IRB has at most 4 ports (read/write), compared to 12 ports (8 read and 4 write) for the primary register file. It does not require bypassing, because the same locations are never written and then read close together in time. Also, it can tolerate being placed far from the core pipeline, and thus has fewer layout constraints. In regard to the last item, we model a 10 cycle (pipelined) access time for the IRB, implying its distance from the core is similar to the L2 cache, certainly further than the L1.

In addition, firmware context switching is well-suited to a processor with a *unified register file* for both architectural registers and for uncommitted results, as in [34, 11]. In that type of architecture, including those with separate floating-point and integer register files, an architectural register is not mapped to a fixed location in the register file, so saving or restoring it involves first consulting the renaming table. The alternative architecture, with a separate reorder buffer and commit register file, may allow for greater hardware support of context switching, but it requires a higher read bandwidth on the reorder buffer for a given level of instruction throughput, and is poorly suited to SMT.

Our firmware approach to context switching does not add additional ports to the register file, since the thread switching operations use the ordinary instruction path. In summary, the inactive register buffer adds no complexity to the core of the processor.

3.2 Time Required to Swap Threads

In our simulations, with the baseline BMT configuration, a majority of firmware context switches take 60 cycles or less. However, there is considerable room for variation. This section describes the range of times required for each step of the context switch.

25 cycles to detect main memory access—If a load instruction does not complete execution in 25 cycles, then it is considered to be a main-memory access. This includes a 3 cycle load instruction latency, a 14 cycle L2 latency, and several extra cycles to account for contention when accessing the L2 cache. This is for the baseline memory architecture. For the other memory designs investigated in Section 5.4, this threshold is adjusted. In principle, this time could be reduced by an early reply from the L2 tag array, or by consulting a load-hit predictor. However, as we show in Section 5.6, switching prematurely can decrease memory parallelism by missing the opportunity to issue independent load misses in parallel with the first miss encountered. The 60 cycle figure above does not include these 25 cycles.

3–30 cycles to trigger flush—There is a 3-cycle minimum delay to trigger a flush in our model. However, older uncommitted instructions from the same thread may further delay the flush. In our simulations, the flush occurs after 3 cycles 64% of the time, within 15 cycles 94% of the time, and very rarely after more than 30 cycles. A flush could be triggered before the canceled load becomes the oldest instruction in its thread, but we found that the cost of unnecessary flushes caused by wrong path instructions outweighed the advantage of flushing sooner.

15 cycles for microcode to reach execute—Instructions can be fetched from the microcode control store immediately after the flush has been triggered. In the pipeline we model, there are 15 stages between fetch and execute.

$\sim$10 cycles to issue `rsave` instructions—The microprogram will contain 1–62 `rsave` instructions, depending on the number of dirty registers. There is considerable variation between benchmarks. Overall, though, on 50% of thread swaps, 20 or fewer registers had been modified, and on 90% of thread swaps, 40 or fewer had been modified. The `rsave` instructions compete to use the integer units with instructions from other active threads, but in the best case, 40 `rsave`s take 10 cycles to execute, 4 at a time.

$\sim$16 cycles to issue `rrestore` instructions—The microprogram concludes with 62 `rrestore` instructions to restore the registers of the new thread. These take at least 16 cycles to execute. For those 4 of the 16 benchmarks which do not use floating-point registers, there are only 31 `rrestore`s.

$\leq$10 cycles restore-use latency—After the microprogram is fetched, but concurrently with the execution of the `rrestore`s, the processor fetches from the new thread. We

Fetch $\leq$ 3 instructions per thread from $\leq$ 2 threads each cycle
Branch prediction 64Kbit 2bcGskew **Deep Pipeline** 22 stages, 16 cycle misp. penalty
Out-of-order execution with 48/32/20 entry integer/fp/memory instruction queues, which may issue 4 integer/mem instructions ($\leq$ 2 mem) and 2 fp instructions each cycle
Instruction Window supports 128 in-flight instructions†
Memory system
32k 4-way 3 cycle L1 Instruction and Data caches (2 acc/cyc)
64 byte linesize for L1 caches
64 entry DTLB / 48 entry ITLB, fully associative
256 entry second level Data and Instruction TLBs
128 byte linesize for higher-level caches
2MB 8-way 14 cycle L2 cache (1 acc/cyc)†
500 cycle memory access time†
†— Baseline parameter, different where noted.

Table 1. Simulated Processor Specifications.

Name	Code	Input	Fast Forward Instructions ($\times 10^6$)
ammp	0		2000
art	1	-startx 110	7500
crafty	2		700
eon	3	rushmeier	100
galgel	4		5000
gap	5		185330
gcc	6	166	2100
gzip	7	graphic	39300
mcf	8		12600
mesa	9		1300
mgrid	A		2100
parser	B		400
perl	C	makerand	10000
twolf	D		900
vortex	E	2	6000
vpr	F	route	36100

Table 2. Benchmarks.

2A 01	Workload	4A 0123	8A 01234567
2B 12	↓ Name	4B 4567	8B 89ABCDEF
2C 23		4C 89AB	8C 02468ACE
2D 34	3A 012	4D CDEF	8D 13579BDF
2E 45	3B 345	4E 0246	
2F 56	3C 678	4F 8ACE	10A 0123456789
2G 67	3D 9AB	4G 1357	10B 456789ABCD
2H 78	3E CDE	4H 9BDF	10C 89ABCDEF01
2I 89	3F 024		10D CDEF012345
2J 9A	3G 68A	6A 012345	
2K AB	3H CEF	6B 6789AB	12A 456789ABCDEF
2L BC	3I 135	6C ABCDEF	12B 012389ABCDEF
2M CD	3J 79B	6D 0369EF	12C 01234567CDEF
2N DE	3K DF6	6E 147C28	12D 0123456789AB
2O EF			
2P F0	Bench.↑ Codes	(see Tbl 2)	16A 01234567 89ABCDEF

Table 3. Workloads.

model a 10 cycle latency for the `rrestore` instructions, and the execution of the `rrestore` instructions is fully pipelined. Depending on what registers are used first by the new thread, there will be a 0–10 cycle delay. This could be reduced by strategically reordering the `rrestore` instructions to match the order of their use by the new thread, based on the instructions previously flushed. Of course, the new thread may also incur an instruction cache miss.

3.3 Common Architecture

The parameters common to all processor designs are shown in Table 1. We intend that these parameters represent a reasonable processor design one or two process generations from now, except that the cache sizes are somewhat smaller than might be projected. We chose relatively smaller cache sizes to match the memory footprint of the benchmarks we use.

The baseline SMT processors we evaluate implements the flush-on-cache-miss policy from [29], which makes more room in the instruction window for instructions from non-stalled threads. Thus, the miss detection and flushing capability required by BMT should not be viewed as an extra cost of our design.

We model a software TLB miss handler mechanism close to that used in the Alpha architecture [7] for all processor designs. For some workloads, page-table walks due to TLB misses represent a significant fraction of all main memory accesses, and a fraction which increases as more threads are run together. Therefore, we allow thread swaps to occur on the loads in the TLB miss trap handler routine. A system with a hardware TLB handler should be able to accommodate thread-swapping as well.

4 Methodology

We evaluate each design alternative by simulation. For each design, we simulate workloads of different sizes. For each workload size, we present the average of several different workloads. Each of the workloads are comprised of a subset of the SPEC2000 benchmarks.

We perform all simulations using a detailed, execution-driven simulator, based on SMTSIM [28]. The simulator executes Alpha binaries which are compiled with the DEC C (`-O4`) or Fortran (`-O5`) compiler. We added a software TLB miss handler that closely models the Alpha architecture PALCode TLB trap handler.

The speedup results we present are meant to be an estimate of the overall improvement in throughput for a system which continuously runs the 16 benchmarks shown in Table 2, as compared to a single-threaded system. We simulate a portion of each benchmark. With the assistance of SimPoint [22], we select a starting point for simulation within each benchmark. Using the multiple simulation point algorithm, we select a phase in each benchmark that represents the largest amount of execution.

We simulate several different workloads for each workload size, which represent a sampling of the space of possible workloads. The exact combinations used are shown in 3, where each workload is described as a string of characters. Each character represents a benchmark, as shown in the column labeled *Codes* in Table 2. For example, workload `2B` consists of 2 threads, `art` and `crafty`. The workloads are selected so that each benchmark is included in more than one workload at each workload size, and to reduce commonality between workloads without unduly increasing the number of simulations. Beyond that, the combinations are selected without any design.

In all simulations, after advancing each thread to the simulation starting point indicated in Table 2 using a checkpoint, we performed a detailed simulation until $10^8 \times n$ instructions had been executed (where n is the number of threads in the workload). When simulating multiple threads, each benchmark in a single workload will run for a different number of instructions under different processor parameters. If there is a large variation in performance of the running threads, this will complicate interpretation of the results. Thus, we present all performance results as *weighted speedup* [25, 29]. The weighted speedup of a multithreaded workload is defined as the sum of the speedups of each individual thread within the workload over a baseline run (in this case single-thread execution). The speedup of a thread within a workload is defined as its performance, in instructions per cycle (IPC), when part of a multithreaded run, divided by its IPC when run by itself over the same range of instructions. Thus weighted speedup represents average relative progress on the workload. By contrast, other metrics, like total instructions per cycle, can artificially create the appearance of increases in performance when more instructions are executed from a higher-IPC thread.

Each speedup we present for 2, 3, 4, 6, 8, 10 or 12 threads at a time represents the average of 16, 11, 8, 6, 5, or 4 different simulations, respectively, as shown in Table 3.

We use CACTI 3.2 [23] for modeling register access times. Since CACTI 3.2 is designed to evaluate the access time of caches, we discarded the tag path in the measures presented here. Access times assume a 70nm process.

5 Analysis and Results

The number of physical contexts supported by a processor, C_{phys}, affects the size of the physical register file and the renaming table, both of which are likely to affect the maximum clock speed and pipeline length. This can degrade performance, especially when one or few threads are run at a time. In this paper, we use the number of physical registers, R_{phys}, as a proxy for these other effects. The following subsection examines the relationship between throughput and R_{phys} for different multithreading schemes. Also in this section, we examine how BMT performs over a range of workload sizes

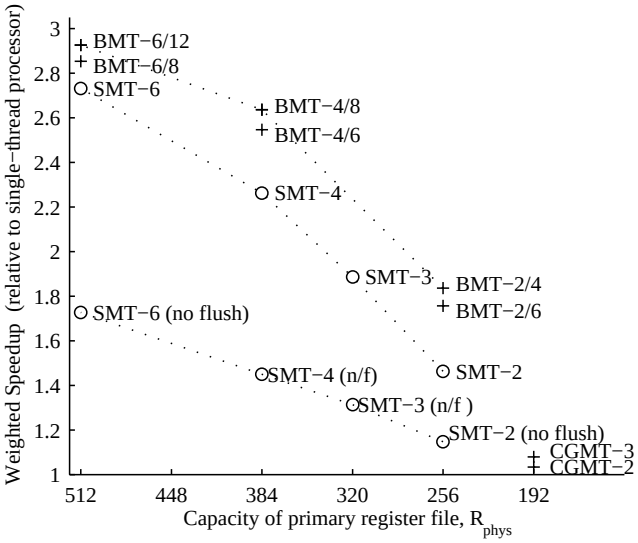

Figure 1. Speedup vs register file size.

and memory parameters. We also examine the importance of firmware support for thread switching, and of store retirement policies. We consider the effect of changing the delay to trigger a thread swap after a miss. Finally, we quantify the effect of a larger register file on overall performance.

5.1 Increasing Throughput Simply

Figure 1 illustrates the tradeoff between throughput and physical register file size. The figure shows that, for a given register file size, a BMT processor gets greater throughput than an SMT processor.

The x-axis shows the number of registers in the physical register file, R_{phys}. We define

$$R_{phys} = C_{phys} \times R_{arch} + R_{ren}$$

where $R_{arch} = 62$ because the Alpha ISA defines 62 non-zero registers, and, again, C_{phys} is the number of physical contexts. All designs assume a single unified physical register file. The IRB is not included in R_{phys}, because it should not be part of a critical circuit timing path. For all the results, except where noted in Section 5.7, $R_{ren} = 128$, which permits 128 in-flight instructions across all threads. The y-axis shows speedup relative to an otherwise equivalent single-thread processor.

There are 4 groups of points to consider. The points labeled *SMT-n*, in the middle curve, show the speedup of SMT processors running a workload of size n. For an SMT processor, $n \leq C_{phys}$, and $C_{phys} = C_{virt}$, so we use $C_{phys} = n$ to compute R_{phys}. The points labeled *SMT-n (no flush)*, in the lower curve, show the performance of a series of SMT processor designs without a mechanism to flush a thread with

a long-latency load [29]. We present this to emphasize the importance of having such a mechanism in any multithreading processor with a shared instruction window. The points labeled *BMT-m/n* represent BMT designs with $C_{phys} = m$ and $C_{virt} \geq n$, running workloads of n threads.

The *BMT-2/4* processor gets 26% more throughput than an *SMT-2* processor, while running at the same clock speed with the same pipeline depth. A 4-context SMT processor gets 17% more throughput when enhanced with BMT, assuming 8 jobs are ready to run.

We model the same pipeline depth and cycle time for all SMT and BMT configurations. As additional hardware contexts are added, keeping the pipeline depth or cycle time constant is unlikely, but the focus of our comparisons are between SMT and BMT configurations with the same number of physical contexts. Because the speedup is not adjusted for these effects, care should be taken when comparing points with different values of R_{phys}. For example, while the 4-context SMT processor shows 54% higher throughput than a 2-context SMT processor when 4 threads are available, differences in the pipeline and/or clock rate between those two designs mean that the relative throughput of the 4-context SMT processor will be lower than that number.

Even ignoring complexity differences, however, the additional benefit of our approach is significant. Regardless of whether a 2, 4 or 6 context SMT processor design is the best choice for particular technology and performance goals, BMT can be added to boost throughput without affecting pipeline complexity. Additionally, these results assume all physical contexts are filled. When there are fewer threads than contexts, the advantage of the BMT designs over SMT are even greater.

This figure also shows the performance of CGMT alone. It provides only marginal gains over a single-threaded processor. Because of the high cost of moving state in and out of the processor core, CGMT alone is of less value. But when CGMT is added to SMT, the additional physical contexts can do useful work while a context switch is underway, hiding the cost of the switch.

5.2 Scalability of Balanced Multithreading

Adding more threads to a processor can increase performance by increasing memory parallelism. However, with too many threads, the benefits can be outweighed by the cost of contention between threads. In this section, we investigate how well different BMT designs perform, compared to SMT designs, as the virtual-to-physical context ratio, C_{virt}/C_{phys}, increases.

The firmware mechanism to swap threads in and out of the processor core has two costs. First, the time required to complete the context switch delays the start of execution of the incoming thread. Second, the firmware save/restore

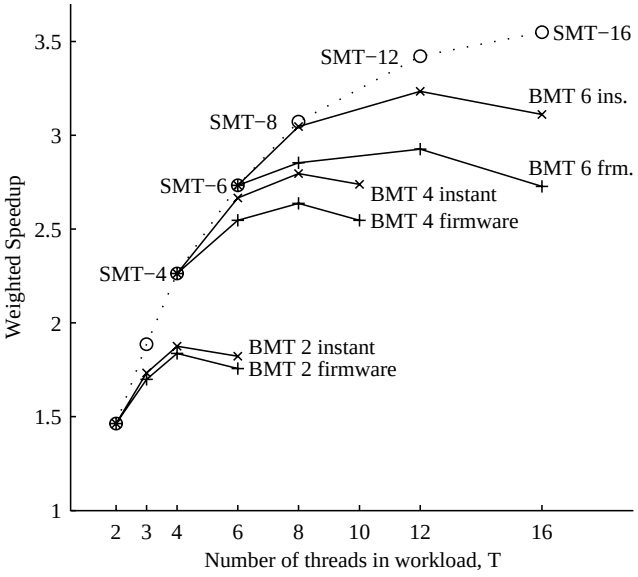

Figure 2. Speedup versus workload size.

instructions contend with other active threads for execution resources. To understand the cost of the firmware context switching mechanism, we compare the performance of the firmware mechanism with a hypothetical *instant* save/restore mechanism.

Figure 2 shows the weighted speedup of several different SMT and BMT designs. The x-axis shows the number of threads in a workload, n, which is assumed to be equal to C_{virt} for this study. The y-axis shows weighted speedup of each design compared to a single-thread processor. On the curve where the points are labeled *SMT-n*, the points represent SMT processors capable of running workloads of n threads together. There are three sets of curves for BMT designs with 2, 4, or 6 physical contexts. Within each set, there is a curve labeled *firmware*, for a processor using the firmware thread swapping mechanism, and a curve labeled *instant* which represents a processor with an idealized, nearly instantaneous thread-swapping mechanism. The instant mechanism requires only 1 cycle to save and restore the architectural registers of the outgoing and incoming threads, once the miss-to-memory is detected and a thread is flushed.

Figure 2 illustrates two effects. First, for each value of C_{phys}, there is an optimal value of C_{virt}/C_{phys}. Second, as C_{phys} increases, the relative cost of the firmware thread swapping mechanism increases too. The figure shows that the gain from BMT peaks when $C_{virt}/C_{phys} = 2$. When the ratio is larger than 2, the costs of running multiple threads begin to outweigh the benefits. For a BMT processor, that cost has two components: the cost of thread swapping and the cost of interference between threads. The curves labeled

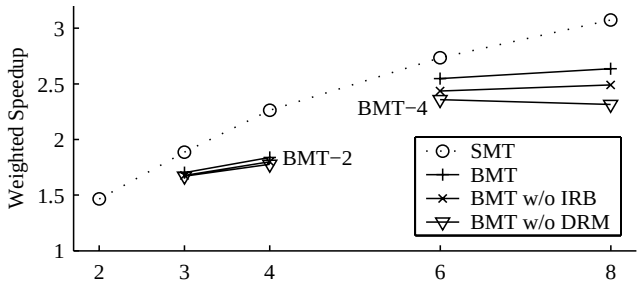

Figure 3. Performance of BMT with different levels of hardware support.

instant, while being perhaps impractical, show the relative contribution of these two effects. When n is small, the cost of swapping is low. The cost of thread swapping comes from contention for instruction queue space and load/store ports from the thread-swapping instructions. Thus, at the *BMT-2* design point, there is little reason to try to further optimize the thread swapping mechanism, but for *BMT-6*, there is an incentive to improve it.

For larger values of C_{virt}/C_{phys} and larger n, the benefit from increased memory parallelism is outweighed by a loss of locality in the higher level caches. The loss of locality is caused by having many threads in the workload. The optimal C_{virt}/C_{phys} ratios suggested by this graph are for an average over many workloads, but will vary with the particular threads running. This represents an opportunity to further improve performance by adaptively sizing the number of threads in a workload based on the behavior of the constituent threads.

5.3 Hardware Support for Thread Swapping

The previous section compared the performance of our baseline thread swapping mechanism with a hypothetical one-cycle latency thread swapping mechanism. Our baseline mechanism already includes some optimizations to reduce swapping latency. This section evaluates two of those optimizations: the Dirty Register Mask (DRM) and the Inactive Register Buffer (IRB).

The DRM, discussed in Section 3.1, allows the thread swap to only save registers values that have been touched. The IRB may be considered an optimization compared to a purely software thread swap, where a context's state is stored using conventional loads and stores. Figure 3 shows the performance of BMT processors with 2 or 4 physical contexts, with varying levels of hardware support for thread swapping, and of SMT processors with 2–6 contexts.

The two BMT features are not important for the *BMT-2* processor, but are important for the *BMT-4* processor. Of the two, the DRM is more important. The benefit of the dirty-register mask increases as more threads are run because of the greater contention for functional units. A lesser effect may be that programs are swapped in for less time when more threads are present, and thus have time to dirty fewer registers.

Without an IRB, inactive registers could be stored directly into memory (where they would typically be caught by the cache). Thus, for the no-IRB configuration, the save-restore instructions use the load/store units, which halves the rate at which they may issue. In the no-IRB configuration, if a miss occurs in the thread-swap microcode, the thread waits instead of performing a second swap. Because such misses are uncommon in the *BMT-2* configurations, there is little performance impact. With a larger workload size, the IRB is important for good performance.

5.4 Sensitivity to Memory Hierarchy

The speedup provided by balanced multithreading is sensitive to three parameters of the memory hierarchy: The size of the caches, the latency to access the lowest level of cache, and the latency to main memory. Figure 4 shows the performance of SMT and BMT with different memory configurations. Each group of bars shows the performance of different processor designs with the same memory hierarchy. All configurations have the L1 caches described in Table 1, but the lower levels of the hierarchy are varied. The configurations were chosen to study the sensitivity to individual memory-system parameters. The y-axis represents weighted speedup. For each group of bars, the speedup is computed relative to a single-threaded processor with the same memory hierarchy. As a result, the speedup for a design with a larger cache hierarchy may be less than that for a design with a smaller cache.

The best C_{virt}/C_{phys} ratio for a BMT system depends on the memory system, so we show two BMT configurations next to each corresponding SMT processor design. Above each group of bars is shown the speedup of the better of the two BMT bars over the adjacent SMT bar. All three of those bars the same C_{phys}. For example, the first group of bars, labeled *Base*, represents the memory configuration used for all previous results in this paper: a 500 cycle memory latency and a 14-cycle 2MB L2 Cache. As noted in the plot, a *BMT-2/4* design gets 26% speedup over an *SMT-2* processor, and a *BMT-4/8* gets 16% speedup over an *SMT-4* design.

Running more threads at the same time has a cost and a benefit. Part of the cost is from increased contention in the caches, predictors and other structures. The benefit is an increase in the number of parallel memory accesses. Changes to the memory parameters shift these costs and benefits.

A larger cache, as in *Big$*, reduces the number of opportunities to use coarse-grained thread switching. Also, a slower cache increases the cost of misses caused by cache contention, and increases the latency before the processor can detect a main memory access. This is illustrated by the lower

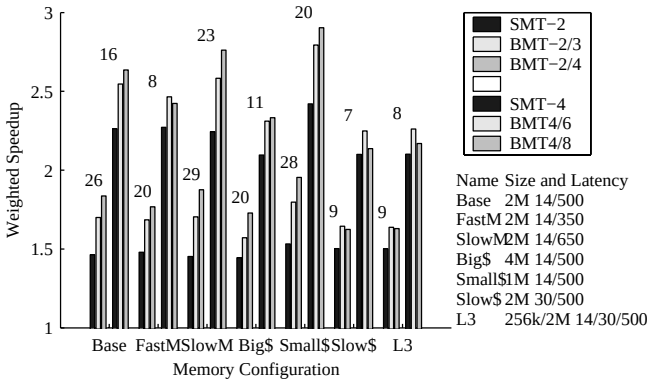

Figure 4. Speedup vs memory hierarchy size and speed

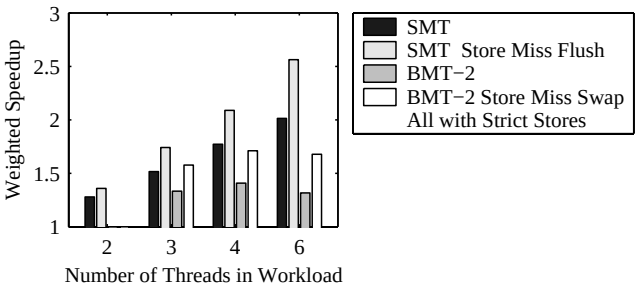

Figure 5. Performance of SMT and BMT processors with strict store retirement policy.

additional speedup from BMT for the *Slow$* group.

The *L3* configuration has a third level of cache, which has both the detrimental effects just mentioned. In this configuration, context switching only occurs on an L3 miss, because the firmware context switch mechanism is too slow to hide an L3 hit. With a faster memory (*FastM*), the fraction of time spent on context switches relative to total execution time increases.

In the case of the larger cache for the *Big$* configuration, there are simply not enough misses to main memory to offset the increase in contention. For example, the *BMT-4/6* processor with the Base memory configuration had, over all workloads, 2.5 main memory accesses per 1000 committed instructions. The same processor with the *Big$* memory configuration only had 0.8 main memory accesses per 1000 instructions. The larger cache significantly reduces the opportunity to benefit from thread swapping. At the same time, the number of Data Cache misses which do not go to memory increases. For *BMT-4/6*, with the *Base* memory, there are 16 L1 misses that do not go to memory per 1000 instructions. For the *Big $* configuration, there are 22 L1 misses per 1000 instructions that are filled without going to memory.

It should be noted that the lessened need for BMT with large caches is primarily a function of the workload, rather than the architecture. Even today, many commercial applications will exercise caches of this size much more heavily. Thus, while cache sizes will increase, which reduces the number of main memory accesses, which in turn reduces the effectiveness of our technique, we expect this effect to be largely mitigated by increases in application working set sizes. Thus, when evaluating our technique, we feel it is fair to focus on the results for the baseline memory configuration.

The bar-groups labeled *FastM* and *SlowM* show results for processors with 350 and 650 cycle main memory latencies, respectively. We use 500 cycles as the baseline main memory latency, and we expect that real systems will reach that level soon. As memory latency increases, the advantage of adding more virtual contexts increases: With *SlowM*, the BMT2/4 and BMT4/8 configurations significantly outperform BMT2/3 and BMT4/6, respectively.

5.5 Store Retirement Policies

All the architectures presented in this chapter allow store instructions which miss in cache to *partially complete*. That is, younger non-store instructions may commit, freeing up space in the instruction window, even when a store's result has not yet been written to the L1 data cache. We believe that this fairly reflects some modern processor designs. Nevertheless, we also evaluated an architecture with a *strict* store retirement policy; younger instructions wait for a store to write to the L1 cache. A strict store retirement policy might be necessary in some systems to insure timely handling of interrupts. With a strict retirement policy, a long-latency store may cause a thread to fill up the instruction window, stalling progress for all threads. To counteract this, we found that swapping on long latency stores as well as loads produces good results. Figure 5 evaluates SMT and BMT architectures with a strict store retirement policy. This shows that, under a strict store retirement policy, SMT and BMT architectures both benefit significantly from flushing or swapping on stores.

5.6 Delayed Detection of Load Misses

In the baseline BMT configuration, a thread swap is triggered when a load instruction takes longer than 25 cycles to complete. The minimum latency for an L2 hit in the baseline architecture is 17 cycles, but some loads take longer due to contention at the L2 cache. Waiting an additional 8 cycles avoids premature swapping. The simple wait-25-cycles approach only requires a small counter for each active load instruction. A alternative mechanism might include a signal from the L2 cache after the tags has been checked. Detect-

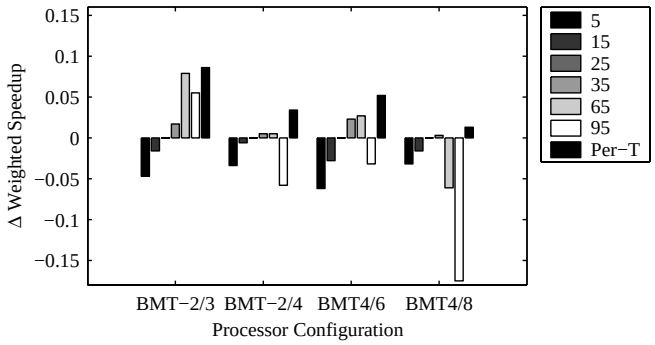

Figure 6. Performance of BMT processors with different delays to initiate swapping on a miss.

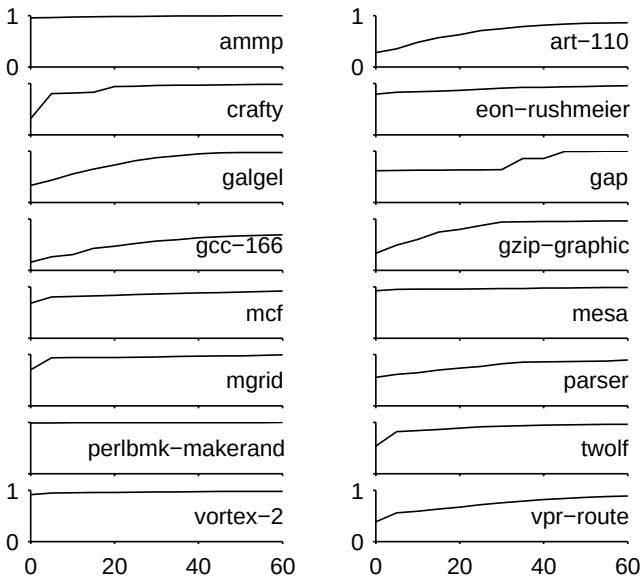

Figure 7. Probability, y, at time x after executing a load instruction which misses in L2, that no further main-memory accesses will be initiated.

ing a load miss and switching sooner may improve performance, since the next thread begins executing sooner. However, flushing a thread too soon can prevent the execution of a second load instruction, which would otherwise initiate a second, parallel, main-memory access.

We evaluated the performance of 4 BMT designs with different values for the load-execution to miss-detection latency, l. Those results are shown in Figure 6. The value of l is indicated in the legend. The y-axis shows the change in weighted speedup for a given design when l is changed from its baseline value of 25. Note that detecting an L2 miss after only 5 cycles would require either checking the L2 tags very quickly, or a load hit predictor. Fortunately, detecting a miss sooner actually decreases throughput. For example, if L2 misses could be detected 5 cycles after a load first executed, the weighted speedup of *BMT-2/3* would drop by 0.05 (from 1.70, as indicated in Figure 1 or Table 4, to 1.65).

In all cases, increasing l to 35 increases the throughput of BMT. However, with larger workloads, higher values of l may reduce throughput. With a larger workload, it is more likely that there is a ready-to-run thread waiting to be swapped in. The best single value of l depends on the number of virtual and physical contexts, and the particular set of benchmarks. However, an even better policy would be one which sets a different value of l for each thread.

It is profitable to delay swapping out a thread if it is likely that additional main memory accesses can be initiated by waiting. As illustrated in Figure 7, benchmarks differ considerably in the number of main-memory accesses that may occur in parallel. There is one subgraph for each benchmark we use. The y-axis shows the probability that no additional main-memory accesses will be initiated following a load which misses in the L2 cache. The x-axis shows time in cycles after the first miss. Note that only subsequent accesses to different cache lines are counted. For `perl` amd `ammp`, when a load misses in the L2, it is highly unlikely that subsequent loads will initiate additional memory activity, so those threads should be swapped out as soon as possible. For `gcc`, even 60 cycles following a L2 miss, it is quite likely that additional misses will occur before the first miss completes, so `gcc` should be swapped out after a longer delay. We evaluated a static, per-thread swap-delay policy. This is shown as the bar labeled *Per–T* in Figure 6. For this policy, all threads are swapped out on a load which takes more than 20 cycles, except `art`, `galgel`, `gap`, `gcc`, and `vpr`, for which $l = 80$. In all 4 cases, the *Per–T* policy performs better than any single value of l. With this policy, BMT2/4 gets an additional 3% speedup over single-thread execution.

We present the Per-T policy to show that there is benefit from a dynamic policy which detects which threads have high memory level parallelism. To implement such a policy, l could be held in a counter which is periodically set to a high value, and which is decremented each time no concurrent misses occur.

5.7 Quantifying the Cost of Additional Registers

The weighted speedup results presented in this paper do not reflect any cycle time or pipeline length penalties that may arise from adding physical contexts to a processor. In this section, we attempt to quantify the cost of adding additional physical contexts to an SMT processor, as opposed to adding virtual contexts.

Table 4 lists the different architectures studied in previous

Type	C_p	n	WSU	R_{ren}	R_{phys}	t_{acc}	n_{stg}
Uni	1	1	1.00	128	190	0.46	5
SMT	2	2	1.46	128	252	0.58	6
	3	3	1.89	”	314	0.60	7
	4	4	2.26	”	376	0.62	7
	6	6	2.73	”	500	0.65	7
	8	8	3.07	”	628	0.72	8
BMT-2	2	3	1.70	128	252	0.58	6
	2	4	1.84	”	”	”	
	2	6	1.76	”	”	”	
BMT-4	4	6	2.57	128	376	0.62	7
	4	8	2.64	”	”	”	
BMT-6	6	12	2.93	128	500	0.65	7

C_p	the number of physical contexts
n	the number of threads in workload
WSU	the weighted speedup
R_{ren}	the number of additional registers for renaming
R_{phys}	the number of physical registers
t_{acc}	the register file access time (ns)
n_{stg}	the estimated number of stages at 10 Ghz

Table 4. Performance and register file speed.

sections. The speedups shown are for the base memory configuration (see Table 1). The last two columns show estimates of the register file access times for different architectures and an estimate of the number of clock cycles that it would require if pipelined at 10 GHz. By this estimate, 3 additional pipeline stages would be needed for an 8 context SMT processor, compared to an otherwise similar 1-context processor. Our access time estimates do not quantify several additional costs of additional contexts. A slower register file read time can add stages between issue and execute, which complicates scheduling. A slower register file write time requires additional hardware to hold bypassed results longer. And a larger register file in turn increases the size of the renaming table. Also, the additional pipeline stages required to tolerate a larger register file fall in a particularly inopportune place in the pipeline. Lengthening the pipeline at this point increases load hit misspeculation penalties [4].

In previous sections, we simulate processors with a large instruction window. The instruction window requires 128 registers beyond those required to hold programmer-visible state (which is 62 per physical context). An alternate way to reduce the size of the register file is to provide fewer of these additional registers. Doing this does not negate the benefit of BMT. If reducing the size of the instruction window increases the performance of the processor, or makes room for additional physical contexts, then BMT can still be used. Figure 8 shows that a *BMT-2/4* processor configuration beats an SMT-2 processor configuration, with fewer additional registers for renaming (a smaller instruction window).

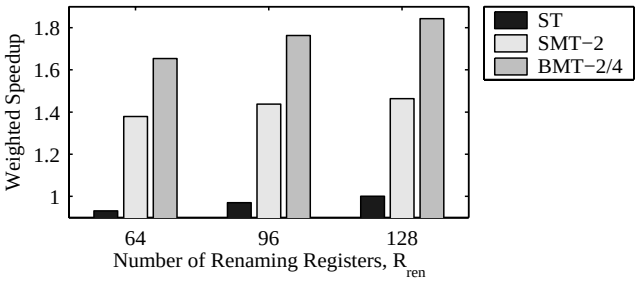

Figure 8. Speedup vs instruction window size. Weighted speedup is relative to single-thread execution with 128 renaming registers.

6 Conclusions

This paper explores the benefits of adding coarse-grained threading support to an SMT processor, creating an architecture we call *Balanced Multithreading*. SMT allows the processor to tolerate even the smallest latencies. CGMT is sufficient to tolerate long memory latencies. We present a form of CGMT which requires no changes to timing-critical processor resources such as the register file and the renaming table. The combination of the two results in a processor that provides high single thread performance via a high clock rate, shorter pipeline and high instruction-level parallelism; and high memory parallelism and thread-level parallelism when more threads are available.

We evaluate the combination of CGMT and SMT, over a range of workload sizes, memory configurations, and several context-switching optimizations, including a method for reducing register saves.

We find that in the face of long memory latencies, balanced multithreading can provide instruction throughput similar to a wide SMT processor, but without many of the hardware costs. In particular, we show that by adding support for balanced multithreading, the throughput of an SMT processor can be improved by 26%, with no significant changes to the core of the processor, the cycle time, or the pipeline.

7 Acknowledgements

We would like to thank the anonymous reviewers for their comments. Eric Tune was supported by an Intel Foundation Fellowship. Other support for this research came from NSF grants CCR-0311683 and CCR-0105743, and a grant from Intel.

References

[1] A. Agarwal, J. Kubiatowicz, D. Kranz, B.-H. Lim, D. Yeung, G. D’Souza, and M. Parkin. Sparcle: An evolutionary proces-

sor design for large-scale multiprocessors. *IEEE Micro*, June 1993.

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

[3] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *In Proceedings of the 34th Annual International Symposium on Microarchitecture*, Dec. 2001.

[4] E. Borch, E. Tune, B. Manne, and J. Emer. Loose loops sink chips. In *Eigth International Symposium on High Performance Computer Architecture*, Feb. 2002.

[5] J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kallaa, and S. R. Kunkel. A multithreaded powerPC processor for commercial servers. *IBM J. Res. Dev.*, 44(6):885–898, 2000.

[6] F. J. Cazorla, E. Fernandez, A. Ramírez, and M. Valero. Improving memory latency aware fetch policies for smt processors. In *Proceedings of the 5th International Symposium on High Performance Computing*, pages 70–85. IEEE Computer Society, October 2003.

[7] Compaq Computer Corp., Shrewsbury, MA. *Alpha 21264 Microprocessor Hardware Reference Manual*, Feb. 2000.

[8] J. Cruz, A. Gonzalez, M. Valero, and N. P. Topham. Multiple-banked register file architectures. In *International Symposium on Computer Architecture(ISCA-27)*, 2000.

[9] R. J. Eickemeyer, R. E. Johnson, S. R. Kunkel, B.-H. Lim, M. S. Squillante, and C. E. Wu. Evaluation of multithreaded processors and thread-switch policies. *International Symposium on High Performance Computing*, pages 75–90, 1997.

[10] R. Halstead and T. Fujita. MASA: a multithreaded processor architecture for parallel symbolic computing. In *25th Annual International Symposium on Computer Architecture*, pages 443–451, 1998.

[11] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the pentium 4 processor. *Intel Technology Journal Q1*, 2001.

[12] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *19th Annual International Symposium on Computer Architecture*, pages 136–145, May 1992.

[13] M. Horowitz, M. Martonosi, T. C. Mowry, and M. D. Smith. Informing memory operations: Providing memory performance feedback in modern processors. In *23rd Annual International Symposium on Computer Architecture*, pages 260–270, 1996.

[14] N. S. Kim and T. Mudge. Reducing register ports using delayed write-back queues and operand pre-fetch. In *17th International Conference on Supercomputing*, June 2003.

[15] J. D. Kubiatowicz. Closing the window of vulnerability in multiphase memory transactions: The alewife transaction store. Master's thesis, Massachusetts Institute of Technology, Feb. 1993.

[16] J. Laudon, A. Gupta, and M. Horowitz. Interleaving: A multithreading technique targeting multiprocessors and workstations. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 308–318, Oct. 1994.

[17] J. Lo, S. Parekh, S. Eggers, H. Levy, and D. Tullsen. Software-directed register deallocation for simultaneous multithreading processors. In *IEEE Transactions on Parallel and Distributed Systems, 10(9)*, Sept. 1999.

[18] T. Mowry and S. Ramkissoon. Software-controlled multithreading using informing memory operations. In *Seventh International Symposium on High Performance Computer Architecture*, 2000.

[19] I. Park, M. Powell, and T. Vijaykumar. Reducing register ports for higher speed and lower energy. In *35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)*, Nov. 2002.

[20] J. Redstone, S. Eggers, and H. Levy. Mini-threads: Increasing TLP on small-scale SMT processors. In *Ninth International Symposium on High Performance Computer Architecture*, Feb. 2003.

[21] R. H. Saavedra-Barrera, D. E. Culler, and T. von Eicken. Analysis of multithreaded architectures for parallel computing. In *Second Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, July 1990.

[22] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Tenth International Comference on Architectural Support for Programming Languages and Operating Systems(ASPLOS 2002)*, Oct. 2002.

[23] P. Shivakumar and N. Jouppi. CACTI 3.0: An integrated cache timing, power and area model. In *Technical Report 2001/2, Compaq Computer Corporation*, Aug. 2001.

[24] B. Smith. The architecture of HEP. In *On Parallel MIMD computation: HEP supercomputer and its applications*, pages 41–55, 1985.

[25] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading architecture. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

[26] R. Thekkath and S. Eggers. The effectiveness of multiple hardware contexts. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1994.

[27] J. Tseng and K. Asanovic. Banked multiported register files for high-frequency superscalar microprocessors. In *In Proceedings of ISCA-30*, June 2003.

[28] D. M. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *22nd Annual Computer Measurement Group Conference*, Dec. 1996.

[29] D. M. Tullsen and J. A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *34th International Symposium on Microarchitecture*, Dec. 2001.

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

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

[32] C. Waldspurger and W. Weihl. Register relocation: Flexible contexts for multithreading. In *20th Annual International Symposium on Computer Architecture*, 1993.

[33] W. Yamamoto and M. Nemirovsky. Increasing superscalar performance through multistreaming. In *Conference on Parallel Architectures and Compilation Techniques*, pages 49–58, June 1995.

[34] K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, 1996.