Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor

Dean M. Tullsen*, Susan J. Eggers*, Joel S. Emer†, Henry M. Levy*,
Jack L. Lo*, and Rebecca L. Stamm†

*Dept of Computer Science and Engineering
University of Washington
Box 352350
Seattle, WA 98195-2350

†Digital Equipment Corporation
HLO2-3/J3
77 Reed Road
Hudson, MA 01749

Abstract

Simultaneous multithreading is a technique that permits multiple independent threads to issue multiple instructions each cycle. In previous work we demonstrated the performance potential of simultaneous multithreading, based on a somewhat idealized model. In this paper we show that the throughput gains from simultaneous multithreading can be achieved *without* extensive changes to a conventional wide-issue superscalar, either in hardware structures or sizes. We present an architecture for simultaneous multithreading that achieves three goals: (1) it minimizes the architectural impact on the conventional superscalar design, (2) it has minimal performance impact on a single thread executing alone, and (3) it achieves significant throughput gains when running multiple threads. Our simultaneous multithreading architecture achieves a throughput of 5.4 instructions per cycle, a 2.5-fold improvement over an unmodified superscalar with similar hardware resources. This speedup is enhanced by an advantage of multithreading previously unexploited in other architectures: the ability to favor for fetch and issue those threads most efficiently using the processor each cycle, thereby providing the "best" instructions to the processor.

1 Introduction

Simultaneous multithreading (SMT) is a technique that permits multiple independent threads to issue multiple instructions each cycle to a superscalar processor's functional units. SMT combines the multiple-instruction-issue features of modern superscalars with the latency-hiding ability of multithreaded architectures. Unlike conventional multithreaded architectures [1, 2, 15, 23], which depend on fast context switching to share processor execution resources, all hardware contexts in an SMT processor are active simultaneously, competing each cycle for all available resources. This dynamic sharing of the functional units allows simultaneous multithreading to substantially increase throughput, attacking the two major impediments to processor utilization — long latencies and limited per-thread parallelism. Tullsen, *et al.*, [27] showed the potential of an SMT processor to achieve significantly higher throughput than either a wide superscalar or a multithreaded processor. That paper also demonstrated the advantages of simultaneous multithreading over multiple processors on a single chip, due to SMT's ability to dynamically assign execution resources where needed each cycle.

Those results showed SMT's potential based on a somewhat idealized model. This paper extends that work in four significant ways. First, we demonstrate that the throughput gains of simultaneous multithreading are possible *without* extensive changes to a conventional, wide-issue superscalar processor. We propose an architecture that is more comprehensive, realistic, and heavily leveraged off existing superscalar technology. Our simulations show that a minimal implementation of simultaneous multithreading achieves throughput 1.8 times that of the unmodified superscalar; small tuning of this architecture increases that gain to 2.5 (reaching throughput as high as 5.4 instructions per cycle). Second, we show that SMT need not compromise single-thread performance. Third, we use our more detailed architectural model to analyze and relieve bottlenecks that did not exist in the more idealized model. Fourth, we show how simultaneous multithreading creates an advantage previously unexploitable in other architectures: namely, the ability to choose the "best" instructions, from all threads, for both fetch and issue each cycle. By favoring the threads most efficiently using the processor, we can boost the throughput of our limited resources. We present several simple heuristics for this selection process, and demonstrate how such heuristics, when applied to the fetch mechanism, can increase throughput by as much as 37%.

This paper is organized as follows. Section 2 presents our baseline simultaneous multithreading architecture, comparing it with existing superscalar technology. Section 3 describes our simulator and our workload, and Section 4 shows the performance of the baseline architecture. In Section 5, we examine the instruction fetch process, present several heuristics for improving it based on intelligent instruction selection, and give performance results to differentiate those heuristics. Section 6 examines the instruction issue process in a similar way. We then use the best designs chosen from our fetch and issue studies in Section 7 as a basis to discover bottlenecks for further performance improvement. We discuss related work in Section 8 and summarize our results in Section 9.

This research was supported by ONR grants N00014-92-J-1395 and N00014-94-1-1136, NSF grants CCR-9200832 and CDA-9123308, NSF PYI Award MIP-9058439, the Washington Technology Center, Digital Equipment Corporation, and fellowships from Intel and the Computer Measurement Group.

Proceedings of the 23rd Annual International Symposium on Computer Architecture, Philadelphia, PA, May, 1996

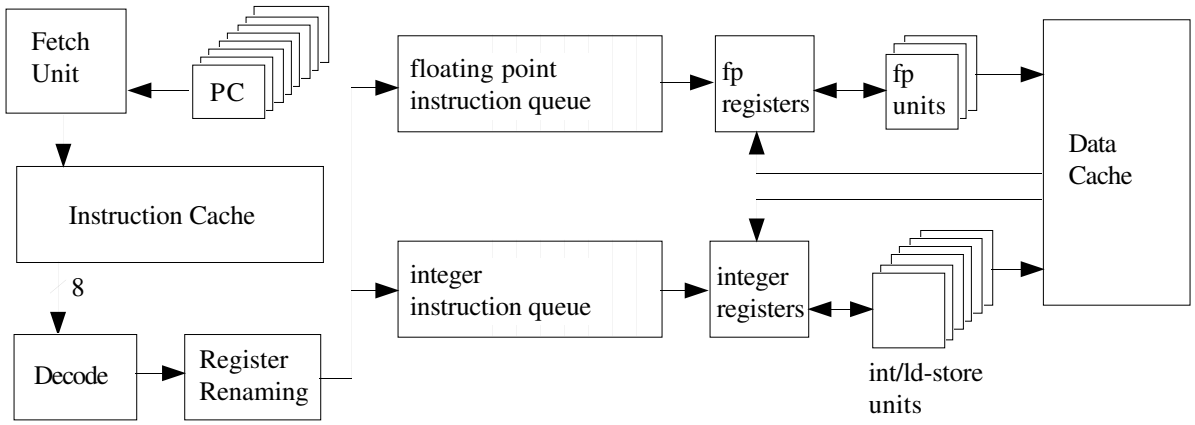

Figure 1: **Our base simultaneous multithreading hardware architecture.**

2 A Simultaneous Multithreading Processor Architecture

In this section we present the architecture of our simultaneous multithreading processor. We show that the throughput gains provided by simultaneous multithreading are possible without adding undue complexity to a conventional superscalar processor design.

Our SMT architecture is derived from a high-performance, out-of-order, superscalar architecture (Figure 1, without the extra program counters) which represents a projection of current superscalar design trends 3-5 years into the future. This superscalar processor fetches up to eight instructions per cycle; fetching is controlled by a conventional system of branch target buffer, branch prediction, and subroutine return stacks. Fetched instructions are then decoded and passed to the register renaming logic, which maps logical registers onto a pool of physical registers, removing false dependences. Instructions are then placed in one of two instruction queues. Those instruction queues are similar to the ones used by the MIPS R10000 [20] and the HP PA-8000 [21], in this case holding instructions until they are issued. Instructions are issued to the functional units out-of-order when their operands are available. After completing execution, instructions are retired in-order, freeing physical registers that are no longer needed.

Our SMT architecture is a straightforward extension to this conventional superscalar design. We made changes only when necessary to enable simultaneous multithreading, and in general, structures were not replicated or resized to support SMT or a multithreaded workload. Thus, nearly all hardware resources remain completely available even when there is only a single thread in the system. The changes necessary to support simultaneous multithreading on that architecture are:

- multiple program counters and some mechanism by which the fetch unit selects one each cycle,
- a separate return stack for each thread for predicting subroutine return destinations,
- per-thread instruction retirement, instruction queue flush, and trap mechanisms,
- a thread id with each branch target buffer entry to avoid predicting phantom branches, and
- a larger register file, to support logical registers for all threads plus additional registers for register renaming. The size of the register file affects the pipeline (we add two extra stages) and the scheduling of load-dependent instructions, which we discuss later in this section.

Noticeably absent from this list is a mechanism to enable simultaneous multithreaded scheduling of instructions onto the functional units. Because any apparent dependences between instructions from different threads are removed by the register renaming phase, a conventional instruction queue (IQ) designed for dynamic scheduling contains all of the functionality necessary for simultaneous multithreading. The instruction queue is shared by all threads and an instruction from any thread in the queue can issue when its operands are available.

We fetch from one program counter (PC) each cycle. The PC is chosen, in round-robin order, from among those threads not already experiencing an I cache miss. This scheme provides simultaneous multithreading at the point of issue, but only fine-grain multithreading of the fetch unit. We will look in Section 5 at ways to extend simultaneous multithreading to the fetch unit. We also investigate alternative thread priority mechanisms for fetching.

A primary impact of multithreading on our architecture is on the size of the register file. We have a single register file, as thread-specific logical registers are mapped onto a completely shared physical register file by the register renaming. To support eight threads, we need a minimum of 8*32 = 256 physical integer registers (for a 32-register instruction set architecture), plus more to enable register renaming. Access to such a large register file will be slow, almost certainly affecting the cycle time of the machine.

To account for the size of the register file, we take two cycles to read registers instead of one. In the first cycle values are read into a buffer closer to the functional units. The instruction is sent to a similar buffer at the same time. The next cycle the data is sent to a functional unit for execution. Writes to the register file are treated similarly, requiring an extra *register write* stage. Figure 2 shows the pipeline modified for two-phase register access, compared to the pipeline of the original superscalar.

The two-stage register access has several ramifications on our architecture. First, it increases the pipeline distance between *fetch* and *exec*, increasing the branch misprediction penalty by 1 cycle. Second, it takes an extra cycle to write back results, requiring an extra level of bypass logic. Third, increasing the distance between

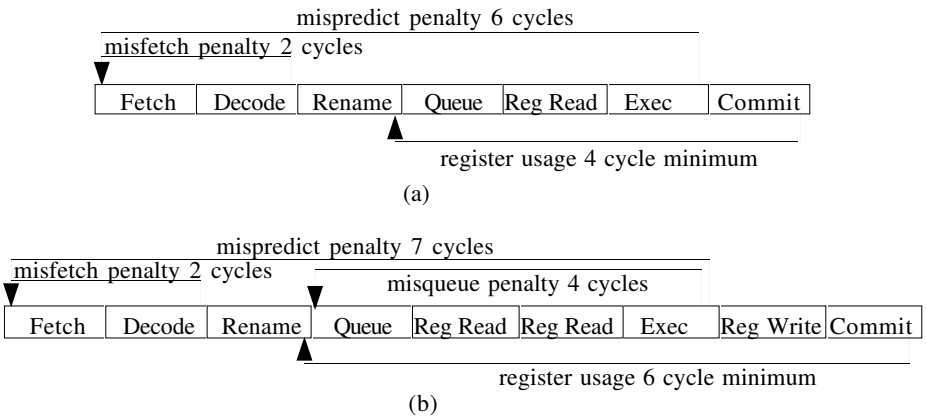

Figure 2: **The pipeline of (a) a conventional superscalar processor and (b) that pipeline modified for an SMT processor, along with some implications of those pipelines.**

queue and *exec* increases the period during which wrong-path instructions remain in the pipeline after a misprediction is discovered (the misqueue penalty in Figure 2). Wrong-path instructions are those instructions brought into the processor as a result of a branch misprediction. Those instructions consume instruction queue slots, renaming registers and possibly issue slots, all of which, on an SMT processor, could be used by other threads.

This pipeline does not increase the inter-instruction latency between most instructions. Dependent (single-cycle latency) instructions can still be issued on consecutive cycles, for example, as long as inter-instruction latencies are predetermined. That is the case for all instructions but loads. Since we are scheduling instructions a cycle earlier (relative to the *exec* cycle), load-hit latency increases by one cycle (to two cycles). Rather than suffer this penalty, we schedule load-dependent instructions assuming a 1-cycle data latency, but squash those instructions in the case of an L1 cache miss or a bank conflict. There are two performance costs to this solution, which we call *optimistic issue*. Optimistically issued instructions that get squashed waste issue slots, and optimistic instructions must still be held in the IQ an extra cycle after they are issued, until it is known that they won't be squashed.

The last implication of the two-phase register access is that there are two more stages between *rename* and *commit*, thus increasing the minimum time that a physical register is held by an in-flight instruction. This increases the pressure on the renaming register pool.

We assume, for each machine size, enough physical registers to support all active threads, plus 100 more registers to enable renaming, both for the integer file and the floating point file; i.e., for the single-thread results, we model 132 physical integer registers, and for an 8-thread machine, 356. We expect that in the 3-5 year time-frame, the scheme we have described will remove register file access from the critical path for a 4-thread machine, but 8 threads will still be a significant challenge. Nonetheless, extending our results to an 8-thread machine allows us to see trends beyond the 4-thread numbers and anticipates other solutions to this problem. The number of registers available for renaming determines the number of instructions that can be in the processor between the *rename* stage and the *commit* stage.

This architecture allows us to address several concerns about simultaneous multithreaded processor design. In particular, this paper shows that:

- Instruction scheduling is no more complex than on a dynamically scheduled superscalar.
- Register file data paths are no more complex than in the superscalar, and the performance implications of the register file and its extended pipeline are small.
- The required instruction fetch throughput is attainable, even without any increase in fetch bandwidth.
- Unmodified (for an SMT workload) cache and branch prediction structures do not thrash on that workload.
- Even aggressive superscalar technologies, such as dynamic scheduling and speculative execution, are not sufficient to take full advantage of a wide-issue processor without simultaneous multithreading.

We have only presented an outline of the hardware architecture to this point; the next section provides more detail.

2.1 Hardware Details

The processor contains 3 floating point functional units and 6 integer units; four of the six integer units also execute loads and stores. The peak issue bandwidth out of the two instruction queues is therefore nine; however, the throughput of the machine is bounded by the peak fetch and decode bandwidths, which are eight instructions per cycle. We assume that all functional units are completely pipelined. Table 1 shows the instruction latencies, which are derived from the Alpha 21164 [8].

We assume a 32-entry integer instruction queue (which handles integer instructions and all load/store operations) and a 32-entry floating point queue, not significantly larger than the HP PA-8000 [21], which has two 28-entry queues.

The caches (Table 2) are multi-ported by interleaving them into banks, similar to the design of Sohi and Franklin [26]. We model lockup-free caches and TLBs. TLB misses require two full memory accesses and no execution resources. We model the memory subsystem in great detail, simulating bandwidth limitations and access conflicts at multiple levels of the hierarchy, to address the concern

Instruction Class	Latency
integer multiply	8,16
conditional move	2
compare	0
all other integer	1
FP divide	17,30
all other FP	4
load (cache hit)	1

Table 1: **Simulated instruction latencies**

	ICache	DCache	L2	L3
Size	32 KB	32 KB	256 KB	2 MB
Associativity	DM	DM	4-way	DM
Line Size	64	64	64	64
Banks	8	8	8	1
Transfer time	1 cycle	1	1	4
Accesses/cycle	var (1-4)	4	1	1/4
Cache fill time	2 cycles	2	2	8
Latency to next level	6	6	12	62

Table 2: **Details of the cache hierarchy**

that memory throughput could be a limiting condition for simultaneous multithreading.

Each cycle, one thread is given control of the fetch unit, chosen from among those not stalled for an instruction cache (I cache) miss. If we fetch from multiple threads, we never attempt to fetch from threads that conflict (on an I cache bank) with each other, although they may conflict with other I cache activity (cache fills).

Branch prediction is provided by a decoupled branch target buffer (BTB) and pattern history table (PHT) scheme [4]. We use a 256-entry BTB, organized as four-way set associative. The 2K x 2-bit PHT is accessed by the XOR of the lower bits of the address and the global history register [18, 30]. Return destinations are predicted with a 12-entry return stack (per context).

We assume an efficient, but not perfect, implementation of dynamic memory disambiguation. This is emulated by using only part of the address (10 bits) to disambiguate memory references, so that it is occasionally over-conservative.

3 Methodology

The methodology in this paper closely follows the simulation and measurement methodology of [27]. Our simulator uses emulation-based, instruction-level simulation, and borrows significantly from MIPSI [22], a MIPS-based simulator. The simulator executes unmodified Alpha object code and models the execution pipelines, memory hierarchy, TLBs, and the branch prediction logic of the processor described in Section 2.

In an SMT processor a branch misprediction introduces wrong-path instructions that interact with instructions from other threads. To model this behavior, we fetch down wrong paths, introduce those instructions into the instruction queues, track their dependences, and issue them. We eventually squash all wrong-path instructions a cycle after a branch misprediction is discovered in the *exec* stage. Our throughput results only count useful instructions.

Our workload comes primarily from the SPEC92 benchmark suite [7]. We use five floating point programs (alvinn, doduc, fpppp, ora, and tomcatv) and two integer programs (espresso and xlisp) from that suite, and the document typesetting program TeX. We assign a distinct program to each thread in the processor: the multiprogrammed workload stresses our architecture more than a parallel program by presenting threads with widely varying program characteristics and with no overlap of cache, TLB or branch prediction usage. To eliminate the effects of benchmark differences, a single data point is composed of 8 runs, each T * 300 million instructions in length, where T is the number of threads. Each of the 8 runs uses a different combination of the benchmarks.

We compile each program with the Multiflow trace scheduling compiler [17], modified to produce Alpha code. In contrast to [27], we turn off trace scheduling in the compiler for this study, for two reasons. In our measurements, we want to differentiate between useful and useless speculative instructions, which is easy with hardware speculation, but not possible for software speculation with our system. Also, software speculation is not as beneficial on an architecture which features hardware speculation, and in some cases is harmful. However, the Multiflow compiler is still a good choice for our compilation engine, because of the high quality of the loop unrolling, instruction scheduling and alignment, and other optimizations, as well as the ease with which the machine model can be changed. The benchmarks are compiled to optimize single-thread performance on the base hardware.

4 Performance of the Base Hardware Design

In this section we examine the performance of the base architecture and identify opportunities for improvement. Figure 3 shows that with only a single thread running on our SMT architecture, the throughput is less than 2% below a superscalar without SMT support. The drop in throughput is due to the longer pipeline (described in Section 2) used by the SMT processor. Its peak throughput is 84% higher than the superscalar. This gain is achieved with virtually no tuning of the base architecture for simultaneous multithreading. This design combines low single-thread impact with high speedup for even a few threads, enabling simultaneous multithreading to reap benefits even in an environment where multiple processes are running only a small fraction of the time. We also note, however, that the throughput peaks before 8 threads, and the processor utilization, at less than 50% of the 8-issue processor, is well short of the potential shown in [27].

We make several conclusions about the potential bottlenecks of this system as we approach 8 threads, aided by Figure 3 and Table 3. Issue bandwidth is clearly not a bottleneck, as the throughput represents a fraction of available issue bandwidth, and our data shows that no functional unit type is being overloaded. We appear to have enough physical registers. The caches and branch prediction logic are being stressed more heavily at 8 threads, but we expect the latency-hiding potential of the additional threads to make up for those drops. The culprit appears to be one or more of the following three problems: (1) IQ size — IQ-full conditions are common, 12 to 21% of cycles total for the two queues; (2) fetch throughput — even in those cycles where we don't experience an IQ-full condition, our data shows that we are sustaining only 4.2 useful instructions fetched per cycle (4.5 including wrong-path); and (3) lack of parallelism — although the queues are reasonably full, we find fewer

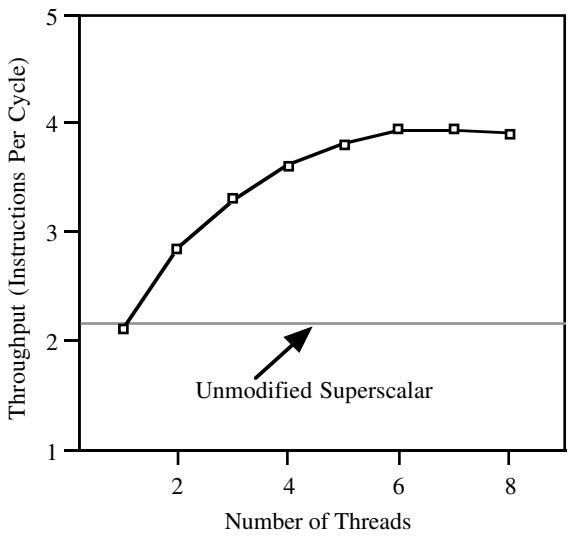

Figure 3: **Instruction throughput for the base hardware architecture.**

Metric	Number of Threads		
	1	4	8
out-of-registers (% of cycles)	3%	7%	3%
I cache miss rate	2.5%	7.8%	14.1%
-misses per thousand instructions	6	17	29
D cache miss rate	3.1%	6.5%	11.3%
-misses per thousand instructions	12	25	43
L2 cache miss rate	17.6%	15.0%	12.5%
-misses per thousand instructions	3	5	9
L3 cache miss rate	55.1%	33.6%	45.4%
-misses per thousand instructions	1	3	4
branch misprediction rate	5.0%	7.4%	9.1%
jump misprediction rate	2.2%	6.4%	12.9%
integer IQ-full (% of cycles)	7%	10%	9%
fp IQ-full (% of cycles)	14%	9%	3%
avg (combined) queue population	25	25	27
wrong-path instructions fetched	24%	7%	7%
wrong-path instructions issued	9%	4%	3%

Table 3: **The result of increased multithreading on some low-level metrics for the base architecture.**

than four out of, on average, 27 instructions per cycle to issue. We expect eight threads to provide more parallelism, so perhaps we have the wrong instructions in the instruction queues.

The rest of this paper focuses on improving this base architecture. The next section addresses each of the problems identified here with different fetch policies and IQ configurations. Section 6 examines ways to prevent issue waste, and Section 7 re-examines the improved architecture for new bottlenecks, identifying directions for further improvement.

5 The Fetch Unit — In Search of Useful Instructions

In this section we examine ways to improve fetch throughput without increasing the fetch bandwidth. Our SMT architecture shares a single fetch unit among eight threads. We can exploit the high level of competition for the fetch unit in two ways not possible with single-threaded processors: (1) the fetch unit can fetch from multiple threads at once, increasing our utilization of the fetch bandwidth, and (2) it can be selective about which thread or threads to fetch from. Because not all paths provide equally useful instructions in a particular cycle, an SMT processor can benefit by fetching from the thread(s) that will provide the best instructions.

We examine a variety of fetch architectures and fetch policies that exploit those advantages. Specifically, they attempt to improve fetch throughput by addressing three factors: fetch efficiency, by partitioning the fetch unit among threads (Section 5.1); fetch effectiveness, by improving the quality of the instructions fetched (Section 5.2); and fetch availability, by eliminating conditions that block the fetch unit (Section 5.3).

5.1 Partitioning the Fetch Unit

Recall that our baseline architecture fetches up to eight instructions from one thread each cycle. The frequency of branches in typical instruction streams and the misalignment of branch destinations make it difficult to fill the entire fetch bandwidth from one thread, even for smaller block sizes [5, 24]. In this processor, we can spread the burden of filling the fetch bandwidth among multiple threads. For example, the probability of finding four instructions from each of two threads should be greater than that of finding eight from one thread.

In this section, we attempt to reduce *fetch block fragmentation* (our term for the various factors that prevent us from fetching the maximum number of instructions) by fetching from multiple threads each cycle, while keeping the maximum fetch bandwidth (but not necessarily the I cache bandwidth) constant. We evaluate several fetching schemes, which are labeled *alg.num1.num2*, where *alg* is the fetch selection method (in this section threads are always selected using a round-robin priority scheme), *num1* is the number of threads that can fetch in 1 cycle, and *num2* is the maximum number of instructions fetched per thread in 1 cycle. The maximum number of total instructions fetched is always limited to eight. For each of the fetch partitioning policies, the cache is always 32 kilobytes organized into 8 data banks; a given bank can do just one access per cycle.

RR.1.8 — This is the baseline scheme from Section 4. Each cycle one thread fetches as many as eight instructions. The thread is determined by a round-robin priority scheme from among those not currently suffering an I cache miss. In this scheme the I cache is indistinguishable from that on a single-threaded superscalar. Each cache bank has its own address decoder and output drivers; each cycle, only one of the banks drives the cache output bus, which is 8 instructions (32 bytes) wide.

RR.2.4, RR.4.2 — These schemes fetch fewer instructions per thread from more threads (four each from two threads, or two each from four threads). If we try to partition the fetch bandwidth too finely, however, we may suffer *thread shortage*, where fewer threads are available than are required to fill the fetch bandwidth.

For these schemes, multiple cache addresses are driven to each cache data bank, each of which now has a multiplexer before its address decoder, to select one cache index per cycle. Since the cache banks are single-ported, bank-conflict logic is needed to ensure that each address targets a separate bank. RR.2.4 has two cache output

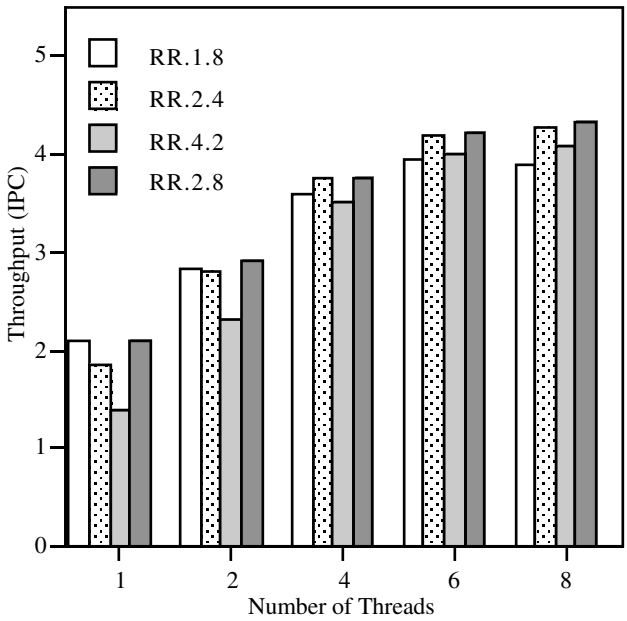

Figure 4: **Instruction throughput for the different instruction cache interfaces with round-robin instruction scheduling.**

buses, each four instructions wide, while RR.4.2 has four output buses, each two instructions wide. For both schemes, the total width of the output buses is 8 instructions (identical to that in RR.1.8), but additional circuitry is needed so each bank is capable of driving any of the multiple (now smaller) output buses, and is able to select one or none to drive in a given cycle. Also, the cache tag store logic must be replicated or multiple-ported in order to calculate hit/miss for each address looked up per cycle.

Thus, the hardware additions are: the address mux; multiple address buses; selection logic on the output drivers; the bank conflict logic; and multiple hit/miss calculations. The changes required for RR.2.4 would have a negligible impact on area and cache access time. The changes for RR.4.2 are more extensive, and would be more difficult to do without affecting area or access time. These schemes actually reduce the latency in the decode and rename stages, as the maximum length of dependency chains among fetched instructions is reduced by a factor of 2 and 4, respectively.

RR.2.8 — This scheme attacks fetch block fragmentation without suffering from thread shortage by fetching eight instructions more flexibly from two threads. This can be implemented by reading an eight-instruction block for each thread (16 instructions total), then combining them. We take as many instructions as possible from the first thread, then fill in with instructions from the second, up to eight total. Like RR.2.4, two addresses must be routed to each cache bank, then multiplexed before the decoder; bank-conflict logic and two hit/miss calculations per cycle are necessary; and each bank drives one of the two output buses. Now, however, each output bus is eight instructions wide, which doubles the bandwidth out of the cache compared to any of the previous schemes. This could be done without greatly affecting area or cycle time, as the additional bussing could probably be done without expanding the cache layout. In addition, logic to select and combine the instructions is necessary, which might or might not require an additional pipe stage. Our simulations assume it does not.

Figure 4 shows that we can get higher maximum throughput by splitting the fetch over multiple threads. For example, the RR.2.4 scheme outperforms RR.1.8 at 8 threads by 9%. However, better maximum throughput comes at the cost of a 12% single-thread penalty; in fact, RR.2.4 does not surpass RR.1.8 until 4 threads. The RR.4.2 scheme needs 6 threads to surpass RR.1.8 and never catches the 2-thread schemes, suffering from thread shortage.

The RR.2.8 scheme provides the best of both worlds: few-threads performance like RR.1.8 and many-threads performance like RR.2.4. However, the higher throughput of this scheme puts more pressure on the instruction queues, causing IQ-full conditions at a rate of 18% (integer) and 8% (fp) with 8 threads.

With the RR.2.8 scheme we have improved the maximum throughput by 10% without compromising single-thread performance. This was achieved by a combination of (1) partitioning the fetch bandwidth over multiple threads, and (2) making that partition flexible. This is the same approach (although in a more limited fashion here) that simultaneous multithreading uses to improve the throughput of the functional units [27].

5.2 Exploiting Thread Choice in the Fetch Unit

The efficiency of the entire processor is affected by the quality of instructions fetched. A multithreaded processor has a unique ability to control that factor. In this section, we examine fetching policies aimed at identifying the "best" thread or threads available to fetch each cycle. Two factors make one thread less desirable than another. The first is the probability that a thread is following a wrong path as a result of an earlier branch misprediction. Wrong-path instructions consume not only fetch bandwidth, but also registers, IQ space, and possibly issue bandwidth. The second factor is the length of time the fetched instructions will be in the queue before becoming issuable. We maximize the throughput of a queue of bounded size by feeding it instructions that will spend the least time in the queue. If we fetch too many instructions that block for a long time, we eventually fill the IQ with unissuable instructions, a condition we call *IQ clog*. This restricts both fetch and issue throughput, causing the fetch unit to go idle and preventing issuable instructions from getting into the IQ. Both of these factors (wrong-path probability and expected queue time) improve over time, so a thread becomes more desirable as we delay fetching it.

We define several fetch policies, each of which attempts to improve on the round-robin priority policy using feedback from other parts of the processor. The first attacks wrong-path fetching, the others attack IQ clog. They are:

BRCOUNT — Here we attempt to give highest priority to those threads that are least likely to be on a wrong path. We do this by counting branch instructions that are in the decode stage, the rename stage, and the instruction queues, favoring those with the fewest unresolved branches.

MISSCOUNT — This policy detects an important cause of IQ clog. A long memory latency can cause dependent instructions to back up in the IQ waiting for the load to complete, eventually filling the queue with blocked instructions from one thread. This policy prevents that by giving priority to those threads that have the fewest outstanding D cache misses.

ICOUNT — This is a more general solution to IQ clog. Here priority is given to threads with the fewest instructions in decode, rename, and the instruction queues. This achieves three purposes: (1) it prevents any one thread from filling the IQ, (2) it gives highest

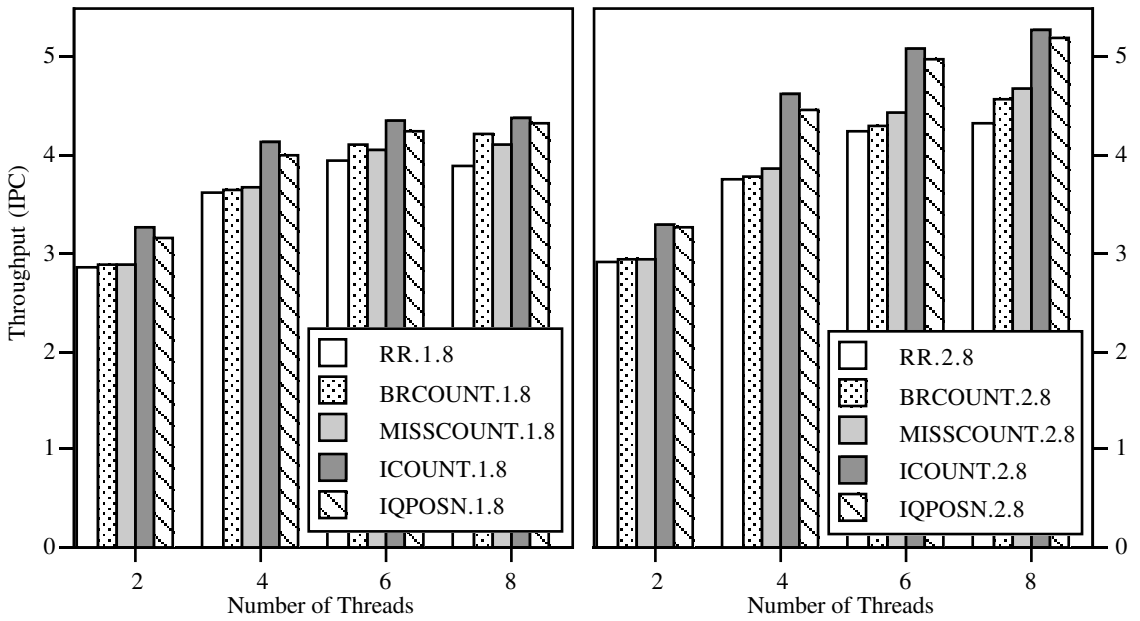

Figure 5: **Instruction throughput for fetching based on several priority heuristics, all compared to the baseline round-robin scheme. The results for 1 thread are the same for all schemes, and thus not shown.**

priority to threads that are moving instructions through the IQ most efficiently, and (3) it provides a more even mix of instructions from the available threads, maximizing the parallelism in the queues. If cache misses are the dominant cause of IQ clog, MISSCOUNT may perform better, since it gets cache miss feedback to the fetch unit more quickly. If the causes are more varied, ICOUNT should perform better.

IQPOSN — Like ICOUNT, IQPOSN strives to minimize IQ clog and bias toward efficient threads. It gives lowest priority to those threads with instructions closest to the head of either the integer or floating point instruction queues (the oldest instruction is at the head of the queue). Threads with the oldest instructions will be most prone to IQ clog, and those making the best progress will have instructions farthest from the head of the queue. This policy does not require a counter for each thread, as do the previous three policies.

Like any control system, the efficiency of these mechanisms is limited by the feedback latency resulting, in this case, from feeding data from later pipeline stages back to the fetch stage. For example, by the time instructions enter the *queue* stage or the *exec* stage, the information used to fetch them is three or (at least) six cycles old, respectively.

Both the branch-counting and the miss-counting policies tend to produce frequent ties. In those cases, the tie-breaker is round-robin priority.

Figure 5 shows that all of the fetch heuristics provide speedup over round-robin. Branch counting and cache-miss counting provide moderate speedups, but only when the processor is saturated with many threads. Instruction counting, in contrast, produces more significant improvements regardless of the number of threads. IQPOSN provides similar results to ICOUNT, being within 4% at all times, but never exceeding it.

The branch-counting heuristic does everything we ask of it. It reduces wrong-path instructions, from 8.2% of fetched instructions to 3.6%, and from 3.6% of issued instructions to 0.8% (RR.1.8 vs. BRCOUNT.1.8 with eight threads). And it improves throughput by as much as 8%. Its weakness is that the wrong-path problem it solves is not large on this processor, which has already attacked the problem with simultaneous multithreading. Even with the RR scheme, simultaneous multithreading reduces fetched wrong-path instructions from 16% with one thread to 8% with 8 threads.

Cache miss counting also achieves throughput gains as high as 8% over RR, but in general the gains are much lower. It is not particularly effective at reducing IQ clog, as we get IQ-full conditions 12% of the time on the integer queue and 14% on the floating point queue (for MISSCOUNT.2.8 with 8 threads). These results indicate that IQ clog is more than simply the result of long memory latencies.

Metric	1 Thread	8 Threads	
		RR	ICOUNT
integer IQ-full (% of cycles)	7%	18%	6%
fp IQ-full (% of cycles)	14%	8%	1%
avg queue population	25	38	30
out-of-registers (% of cycles)	3%	8%	5%

Table 4: **Some low-level metrics for the round-robin and instruction-counting priority policies (and the 2.8 fetch partitioning scheme).**

The instruction-counting heuristic provides instruction throughput as high as 5.3 instructions per cycle, a throughput gain over the unmodified superscalar of 2.5. It outperforms the best round-robin result by 23%. Instruction counting is as effective at 2 and 4 threads (in benefit over round-robin) as it is at 8 threads. It nearly eliminates IQ clog (see IQ-full results in Table 4) and greatly improves the mix of instructions in the queues (we are finding more issuable instructions despite having fewer instructions in the two queues). Intelligent fetching with this heuristic is of greater benefit than partitioning the fetch unit, as the ICOUNT.1.8 scheme consistently

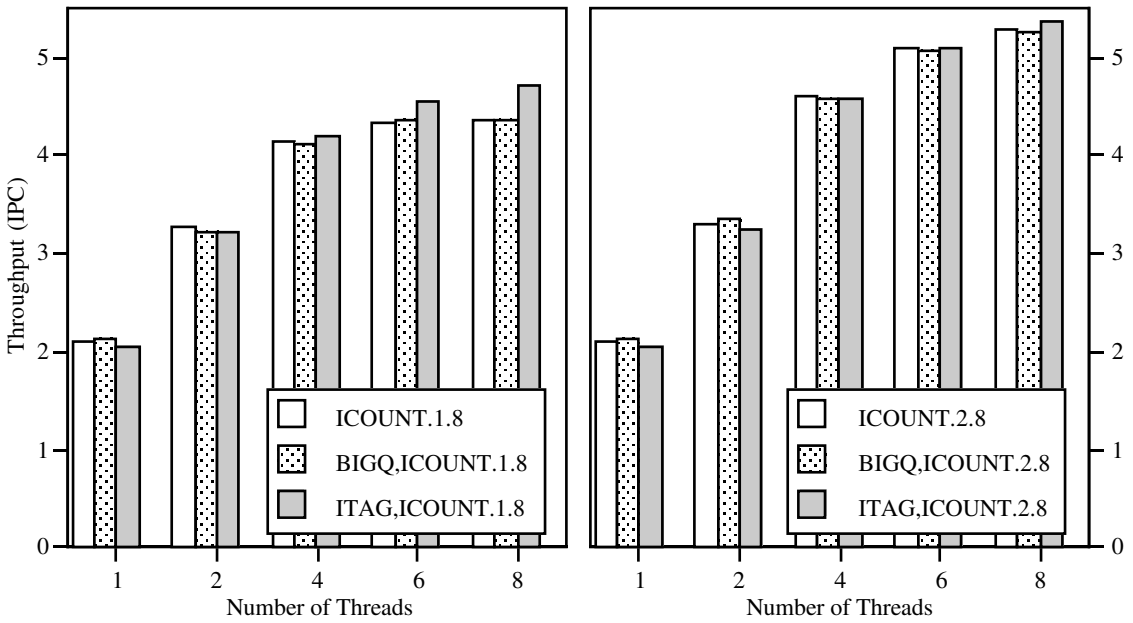

Figure 6: **Instruction throughput for the 64-entry queue and early I cache tag lookup, when coupled with the ICOUNT fetch policy.**

outperforms RR.2.8.

Table 4 points to a surprising result. As a result of simultaneous multithreaded instruction issue and the ICOUNT fetch heuristics, we actually put *less* pressure on the same instruction queue with eight threads than with one, having sharply reduced IQ-full conditions. It also reduces pressure on the register file (vs. RR) by keeping fewer instructions in the queue.

BRCOUNT and ICOUNT each solve different problems, and perhaps the best performance could be achieved from a weighted combination of them; however, the complexity of the feedback mechanism increases as a result. By itself, instruction counting clearly provides the best gains.

Given our measurement methodology, it is possible that the throughput increases could be overstated if a fetch policy simply favors those threads with the most inherent instruction-level parallelism or the best cache behavior, thus achieving improvements that would not be seen in practice. However, with the ICOUNT.2.8 policy, the opposite happens. Our results show that this scheme favors threads with lower single-thread ILP, thus its results include a higher sample of instructions from the slow threads than either the superscalar results or the RR results. If anything, then, the ICOUNT.2.8 improvements are understated.

In summary, we have identified a simple heuristic that is very successful at identifying the best threads to fetch. Instruction counting dynamically biases toward threads that will use processor resources most efficiently, thereby improving processor throughput as well as relieving pressure on scarce processor resources: the instruction queues and the registers.

5.3 Unblocking the Fetch Unit

By fetching from multiple threads and using intelligent fetch heuristics, we have significantly increased fetch throughput and efficiency. The more efficiently we are using the fetch unit, the more we stand to lose when it becomes blocked. In this section we examine schemes that prevent two conditions that cause the fetch unit to miss fetch opportunities, specifically IQ-full conditions and I cache misses. The two schemes are:

BIGQ — The primary restriction on IQ size is not the chip area, but the time to search it; therefore we can increase its size as long as we don't increase the search space. In this scheme, we double the sizes of the instruction queues, but only search the first 32 entries for issue. This scheme allows the queues to buffer instructions from the fetch unit when the IQ overflows.

ITAG — When a thread is selected for fetching but experiences a cache miss, we lose the opportunity to fetch that cycle. If we do the I cache tag lookups a cycle early, we can fetch around cache misses: cache miss accesses are still started immediately, but only non-missing threads are chosen for fetch. Because we need to have the fetch address a cycle early, we essentially add a stage to the front of the pipeline, increasing the misfetch and mispredict penalties. This scheme requires one or more additional ports on the I cache tags, so that potential replacement threads can be looked up at the same time.

Although the BIGQ scheme improves the performance of the round-robin scheme (not shown), 1.5-2% across the board, Figure 6 shows that the bigger queues add no significant improvement to the ICOUNT policy. In fact, it is actually detrimental for several thread configurations. This is because the buffering effect of the big queue scheme brings instructions into the issuable part of the instruction queue that may have been fetched many cycles earlier, using priority information that is now out-of-date. The results indicate that using up-to-date priority information is more important than buffering.

These results show that intelligent fetch heuristics have made the extra instruction queue hardware unnecessary. The bigger queue by itself is actually *less* effective at reducing IQ clog than the ICOUNT scheme. With 8 threads, the bigger queues alone (BIGQ,RR.2.8) reduce IQ-full conditions to 11% (integer) and 0% (fp), while instruction counting alone (ICOUNT.2.8) reduces them to 6% and 1%. Combining BIGQ and ICOUNT drops them to 3% and 0%.

Early I cache tag lookup boosts throughput as much as 8%

Issue Method	Number of Threads					Useless Instructions	
	1	2	4	6	8	wrong-path	optimistic
OLDEST	2.10	3.30	4.62	5.09	5.29	4%	3%
OPT_LAST	2.07	3.30	4.59	5.09	5.29	4%	2%
SPEC_LAST	2.10	3.31	4.59	5.09	5.29	4%	3%
BRANCH_FIRST	2.07	3.29	4.58	5.08	5.28	4%	6%

Table 5: **Instruction throughput (instructions per cycle) for the issue priority schemes, and the percentage of useless instructions issued when running with 8 threads.**

over ICOUNT. It is most effective when fetching one thread (ICOUNT.1.8, where the cost of a lost fetch slot is greater). However, it improves the ICOUNT.2.8 results no more than 2%, as the flexibility of the 2.8 scheme already hides some of the lost fetch bandwidth. In addition, ITAG lowers throughput with few threads, where competition for the fetch slots is low and the cost of the longer misprediction penalty is highest.

Using a combination of partitioning the fetch unit, intelligent fetching, and early I cache tag lookups, we have raised the peak performance of the base SMT architecture by 37% (5.4 instructions per cycle vs. 3.9). Our maximum speedup relative to a conventional superscalar has gone up proportionately, from 1.8 to 2.5 times the throughput. That gain comes from exploiting characteristics of a simultaneous multithreading processor not available to a single-threaded machine.

High fetch throughput makes issue bandwidth a more critical resource. We focus on this factor in the next section.

6 Choosing Instructions For Issue

Much as the fetch unit in a simultaneous multithreading processor can take advantage of the ability to choose which threads to fetch, the issue logic has the ability to choose instructions for issue. A dynamically scheduled single-threaded processor may have enough ready instructions to be able to choose between them, but with an SMT processor the options are more diverse. Also, because we have higher throughput than a single-threaded superscalar processor, the issue bandwidth is potentially a more critical resource, so avoiding issue slot waste may be more beneficial.

In this section, we examine issue priority policies aimed at preventing issue waste. Issue slot waste comes from two sources, wrong-path instructions (resulting from mispredicted branches) and optimistically issued instructions. Recall (from Section 2) that we optimistically issue load-dependent instructions a cycle before we have D cache hit information. In the case of a cache miss or bank conflict, we have to squash the optimistically issued instruction, wasting that issue slot.

In a single-threaded processor, choosing instructions least likely to be on a wrong path is always achieved by selecting the oldest instructions (those deepest into the instruction queue). In a simultaneous multithreading processor, the position of an instruction in the queue is no longer the best indicator of the level of speculation of that instruction, as right-path instructions are intermingled in the queues with wrong-path.

The policies we examine are OLDEST_FIRST, our default issue algorithm up to this point, OPT_LAST and SPEC_LAST, which only issue optimistic and speculative instructions (more specifically, any instruction behind a branch from the same thread in the instruction queue), respectively, after all others have been issued, and BRANCH_FIRST, which issues branches as early as possible in order to identify mispredicted branches quickly. The default fetch algorithm for each of these schemes is ICOUNT.2.8.

The strong message of Table 5 is that issue bandwidth is not yet a bottleneck. Even when it does become a critical resource, the amount of improvement we get from not wasting it is likely to be bounded by the percentage of our issue bandwidth given to useless instructions, which currently stands at 7% (4% wrong-path instructions, 3% squashed optimistic instructions). Because we don't often have more issuable instructions than functional units, we aren't able to and don't need to reduce that significantly. The SPEC_LAST scheme is unable to reduce the number of useless instructions at all, while OPT_LAST brings it down to 6%. BRANCH_FIRST actually increases it to 10%, as branch instructions are often load-dependent; therefore, issuing them as early as possible often means issuing them optimistically. A combined scheme (OPT_LAST and BRANCH_FIRST) might reduce that side effect, but is unlikely to have much effect on throughput.

Since each of the alternate schemes potentially introduces multiple passes to the IQ search, it is convenient that the simplest mechanism still works well.

7 Where Are the Bottlenecks Now?

We have shown that proposed changes to the instruction queues and the issue logic are unnecessary to achieve the best performance with this architecture, but that significant gains can be produced by moderate changes to the instruction fetch mechanisms. Here we examine that architecture more closely (using ICOUNT.2.8 as our new baseline), identifying likely directions for further improvements.

In this section we present results of experiments intended to identify bottlenecks in the new design. For components that are potential bottlenecks, we quantify the size of the bottleneck by measuring the impact of relieving it. For some of the components that are not bottlenecks, we examine whether it is possible to simplify those components without creating a bottleneck. Because we are identifying bottlenecks rather than proposing architectures, we are no longer bound by implementation practicalities in these experiments.

The Issue Bandwidth — The experiments in Section 6 indicate that issue bandwidth is not a bottleneck. In fact, we found that even an infinite number of functional units increases throughput by only 0.5% at 8 threads.

Instruction Queue Size — Results in Section 5 would, similarly, seem to imply that the size of the instruction queues was not a bottleneck, particularly with instruction counting; however, the schemes we examined are not the same as larger, searchable queues, which would also increase available parallelism. Nonetheless, the exper-

iment with larger (64-entry) queues increased throughput by less than 1%, despite reducing IQ-full conditions to 0%.

Fetch Bandwidth — Although we have significantly improved fetch throughput, it is still a prime candidate for bottleneck status. Branch frequency and PC alignment problems still prevent us from fully utilizing the fetch bandwidth. A scheme that allows us to fetch as many as 16 instructions (up to eight each from two threads), increases throughput 8% to 5.7 instructions per cycle. At that point, however, the IQ size and the number of physical registers each become more of a restriction. Increasing the instruction queues to 64 entries and the excess registers to 140 increases performance another 7% to 6.1 IPC. These results indicate that we have not yet completely removed fetch throughput as a performance bottleneck.

Branch Prediction — Simultaneous multithreading has a dual effect on branch prediction, much as it has on caches. While it puts more pressure on the branch prediction hardware (see Table 3), it is more tolerant of branch mispredictions. This tolerance arises because SMT is less dependent on techniques that expose single-thread parallelism (e.g., speculative fetching and speculative execution based on branch prediction) due to its ability to exploit inter-thread parallelism. With one thread running, on average 16% of the instructions we fetch and 10% of the instructions we execute are down a wrong path. With eight threads running and the ICOUNT fetch scheme, only 9% of the instructions we fetch and 4% of the instructions we execute are wrong-path.

Perfect branch prediction boosts throughput by 25% at 1 thread, 15% at 4 threads, and 9% at 8 threads. So despite the significantly decreased efficiency of the branch prediction hardware, simultaneous multithreading is much less sensitive to the quality of the branch prediction than a single-threaded processor. Still, better branch prediction is beneficial for both architectures. Significant improvements come at a cost, however; a better scheme than our baseline (doubling the size of both the BTB and PHT) yields only a 2% gain at 8 threads.

Speculative Execution — The ability to do speculative execution on this machine is not a bottleneck, but we would like to know whether eliminating it would create one. The cost of speculative execution (in performance) is not particularly high (again, 4% of issued instructions are wrong-path), but the benefits may not be either.

Speculative execution can mean two different things in an SMT processor, (1) the ability to issue wrong-path instructions that can interfere with others, and (2) the ability to allow instructions to issue before preceding branches from the same thread. In order to guarantee that no wrong-path instructions are issued, we need to delay instructions 4 cycles after the preceding branch is issued. Doing this reduces throughput by 7% at 8 threads, and 38% at 1 thread. Simply preventing instructions from passing branches only lowers throughput 1.5% (vs. 12% for 1 thread). Simultaneous multithreading (with many threads) benefits much less from speculative execution than a single-threaded processor; it benefits more from the ability to issue wrong-path instructions than from allowing instructions to pass branches.

Memory Throughput — While simultaneous multithreading hides memory latencies effectively, it is less effective if the problem is memory throughput, since it does not address that problem. For that reason, our simulator models memory throughput limitations at multiple levels of the cache hierarchy, and the buses between them. With our workload, we never saturate any single cache or bus, but in some cases there are significant queueing delays for certain levels of the cache. If we had infinite bandwidth caches (i.e., the same cache latencies, but no cache bank or bus conflicts), the throughput would only increase by 3%.

Register File Size — The number of registers required by this machine is a very significant issue. While we have modeled the effects of register renaming, we have not set the number of physical registers low enough that it is a significant bottleneck. In fact, setting the number of excess registers to infinite instead of 100 only improves 8-thread performance by 2%. Lowering it to 90 reduces performance by 1%, and to 80 by 3%, and 70 by 6%, so there is no sharp drop-off point. The ICOUNT fetch scheme is probably a factor in this, as we've shown that it creates more parallelism with fewer instructions in the machine. With four threads and fewer excess registers, the reductions were nearly identical.

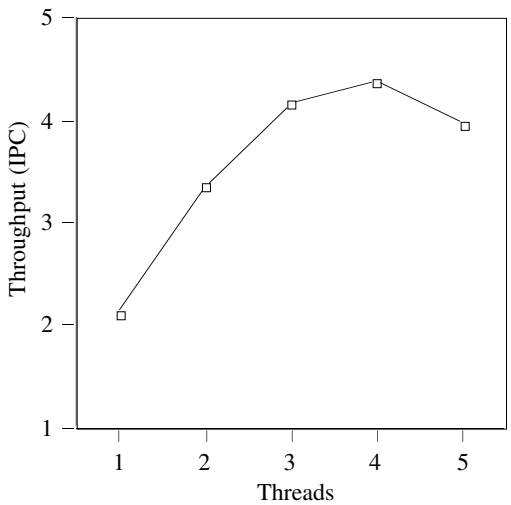

Figure 7: **Instruction throughput for machines with 200 physical registers and from 1 to 5 hardware contexts.**

However, this does not completely address the total size of the register file, particularly when comparing different numbers of threads. An alternate approach is to examine the maximize performance achieved with a given set of physical registers. For example, if we identify the largest register file that could support the scheme outlined in Section 2, then we can investigate how many threads to support for the best performance. The tradeoff arises because supporting more hardware contexts leaves fewer (excess) registers available for renaming. The number of renaming registers, however, determines the total number of instructions the processor can have in-flight. It is difficult to predict the right register file size that far into the future, but in Figure 7 we illustrate this type of analysis by finding the performance achieved with 200 physical registers. That equates to a 1-thread machine with 168 excess registers or a 4-thread machine with 72 excess registers, for example. In this case there is a clear maximum point at 4 threads.

In summary, fetch throughput is still a bottleneck in our proposed architecture. It may no longer be appropriate to keep fetch and issue bandwidth in balance, given the much greater difficulty of filling the fetch bandwidth. Also, register file access time will likely be a limiting factor in the number of threads an architecture can support.

8 Related Work

A number of other architectures have been proposed that exhibit simultaneous multithreading in some form. Tullsen, *et al.*, [27] demonstrated the potential for simultaneous multithreading, but did not simulate a complete architecture, nor did that paper present a specific solution to register file access or instruction scheduling. This paper presents an architecture that realizes much of the potential demonstrated by that work, simulating it in detail.

Hirata, *et al.*, [13] present an architecture for a multithreaded superscalar processor and simulate its performance on a parallel ray-tracing application. They do not simulate caches or TLBs and their architecture has no branch prediction mechanism. Yamamoto, *et al.*, [29] present an analytical model of multithreaded superscalar performance, backed up by simulation. Their study models perfect branching, perfect caches and a homogeneous workload (all threads running the same trace). Yamamoto and Nemirovsky [28] simulate an SMT architecture with separate instruction queues and up to four threads. Gulati and Bagherzadeh [11] model a 4-issue machine with four hardware contexts and a single compiler-partitioned register file.

Keckler and Dally [14] and Prasadh and Wu [19] describe architectures that dynamically interleave operations from VLIW instructions onto individual functional units.

Daddis and Torng [6] plot increases in instruction throughput as a function of the fetch bandwidth and the size of the dispatch stack, a structure similar to our instruction queue. Their system has two threads, unlimited functional units, and unlimited issue bandwidth.

In addition to these, Beckmann and Polychronopoulus [3], Gunther [12], Li and Chu [16], and Govindarajan, *et al.*, [10] all discuss architectures that feature simultaneous multithreading, none of which can issue more than one instruction per cycle per thread.

Our work is distinguished from most of these studies in our dual goals of maintaining high single-thread performance and minimizing the architectural impact on a conventional processor. For example, two implications of those goals in our architecture are limited fetch bandwidth and a centralized instruction scheduling mechanism based on a conventional instruction queue.

Most of these studies either model infinite fetch bandwidth (with perfect caches) or high-bandwidth instruction fetch, each context fetching from a private cache. However, Hirata, *et al.*, and Daddis and Torng both model limited fetch bandwidth (with zero-latency memory), using round-robin priority, our baseline mechanism; neither model the instruction cache, however. Gulati and Bagherzadeh fetch from a single thread each cycle, and even look at thread selection policies, but find no policy with improvement better than intelligent round robin.

Also, only a few of these studies use any kind of centralized scheduling mechanism: Yamamoto, *et al.*, model a global instruction queue that only holds ready instructions; Govindarajan, *et al.*, and Beckmann and Polychronopoulus have central queues, but threads are very restricted in the number of instructions they can have active at once; Daddis and Torng model an instruction queue similar to ours, but they do not couple that with a realistic model of functional units, instruction latencies, or memory latencies. Gulati and Bagherzadeh model an instruction window composed of four-instruction blocks, each block holding instructions from a single thread.

The M-Machine [9] and the Multiscalar project [25] combine multiple-issue with multithreading, but assign work onto processors at a coarser level than individual instructions. Tera [2] combines LIW with fine-grain multithreading.

9 Summary

This paper presents a simultaneous multithreading architecture that:

- borrows heavily from conventional superscalar design, requiring little additional hardware support,
- minimizes the impact on single-thread performance, running only 2% slower in that scenario, and
- achieves significant throughput improvements over the superscalar when many threads are running: a 2.5 throughput gain at 8 threads, achieving 5.4 IPC.

The fetch improvements result from two advantages of simultaneous multithreading unavailable to conventional processors: the ability to partition the fetch bandwidth over multiple threads, and the ability to dynamically select for fetch those threads that are using processor resources most efficiently.

Simultaneous multithreading achieves multiprocessor-type speedups without multiprocessor-type hardware explosion. This architecture achieves significant throughput gains over a superscalar using the same cache sizes, fetch bandwidth, branch prediction hardware, functional units, instruction queues, and TLBs. The SMT processor is actually less sensitive to instruction queue and branch prediction table sizes than the single-thread superscalar, even with a multiprogrammed workload.

Acknowledgments

We would like to thank Tryggve Fossum for his support of this work and for numerous suggestions. And we would like to thank Bert Halstead and Rishiyur Nikhil for several valuable discussions, and the referees for their helpful comments. We would also like to thank John O'Donnell from Equator Technologies, Inc. for access to the source for the Multiflow compiler.

References

[1] A. Agarwal, B.H. Lim, D. Kranz, and J. Kubiatowicz. APRIL: a processor architecture for multiprocessing. In *17th Annual International Symposium on Computer Architecture*, pages 104–114, May 1990.

[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] C.J. Beckmann and C.D. Polychronopoulos. Microarchitecture support for dynamic scheduling of acyclic task graphs. In *25th Annual International Symposium on Microarchitecture*, pages 140–148, December 1992.

[4] B. Calder and D. Grunwald. Fast and accurate instruction fetch and branch prediction. In *21st Annual International Symposium on Computer Architecture*, pages 2–11, April 1994.

[5] T.M. Conte, K.N. Menezes, P.M. Mills, and B.A. Patel. Optimization of instruction fetch mechanisms for high issue rates. In *22nd Annual International Symposium on Computer Architecture*, pages 333–344, June 1995.

[6] G.E. Daddis, Jr. and H.C. Torng. The concurrent execution of multiple instruction streams on superscalar processors. In *International Conference on Parallel Processing*, pages I:76–83, August 1991.

[7] K.M. Dixit. New CPU benchmark suites from SPEC. In *COMPCON, Spring 1992*, pages 305–310, 1992.

[8] J. Edmondson and P. Rubinfield. An overview of the 21164 AXP microprocessor. In *Hot Chips VI*, pages 1–8, August 1994.

[9] M. Fillo, S.W. Keckler, W.J. Dally, N.P. Carter, A. Chang, Y. Gurevich, and W.S. Lee. The M-Machine multicomputer. In *28th Annual International Symposium on Microarchitecture*, November 1995.

[10] R. Govindarajan, S.S. Nemawarkar, and P. LeNir. Design and peformance evaluation of a multithreaded architecture. In *First IEEE Symposium on High-Performance Computer Architecture*, pages 298–307, January 1995.

[11] M. Gulati and N. Bagherzadeh. Performance study of a multithreaded superscalar microprocessor. In *Second International Symposium on High-Performance Computer Architecture*, pages 291–301, February 1996.

[12] B.K. Gunther. *Superscalar performance in a multithreaded microprocessor*. PhD thesis, University of Tasmania, December 1993.

[13] 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.

[14] S.W. Keckler and W.J. Dally. Processor coupling: Integrating compile time and runtime scheduling for parallelism. In *19th Annual International Symposium on Computer Architecture*, pages 202–213, May 1992.

[15] 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, October 1994.

[16] Y. Li and W. Chu. The effects of STEF in finely parallel multithreaded processors. In *First IEEE Symposium on High-Performance Computer Architecture*, pages 318–325, January 1995.

[17] P.G. Lowney, S.M. Freudenberger, T.J. Karzes, W.D. Lichtenstein, R.P. Nix, J.S. ODonnell, and J.C. Ruttenberg. The multiflow trace scheduling compiler. *Journal of Supercomputing*, 7(1-2):51–142, May 1993.

[18] S. McFarling. Combining branch predictors. Technical Report TN-36, DEC-WRL, June 1993.

[19] R.G. Prasadh and C.-L. Wu. A benchmark evaluation of a multi-threaded RISC processor architecture. In *International Conference on Parallel Processing*, pages I:84–91, August 1991.

[20] Microprocessor Report, October 24 1994.

[21] Microprocessor Report, November 14 1994.

[22] E.G. Sirer. Measuring limits of fine-grained parallelism. Senior Independent Work, Princeton University, June 1993.

[23] B.J. Smith. Architecture and applications of the HEP multiprocessor computer system. In *SPIE Real Time Signal Processing IV*, pages 241–248, 1981.

[24] M.D. Smith, M. Johnson, and M.A. Horowitz. Limits on multiple instruction issue. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 290–302, 1989.

[25] G.S. Sohi, S.E. Breach, and T.N. Vijaykumar. Multiscalar processors. In *22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[26] G.S. Sohi and M. Franklin. High-bandwidth data memory systems for superscalar processors. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 53–62, April 1991.

[27] 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.

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

[29] W. Yamamoto, M.J. Serrano, A.R. Talcott, R.C. Wood, and M. Nemirosky. Performance estimation of multistreamed, superscalar processors. In *Twenty-Seventh Hawaii International Conference on System Sciences*, pages I:195–204, January 1994.

[30] T.-Y. Yeh and Y. Patt. Alternative implementations of two-level adaptive branch prediction. In *19th Annual International Symposium on Computer Architecture*, pages 124–134, May 1992.