The Impact of Resource Partitioning on SMT Processors

Steven E. Raasch and Steven K. Reinhardt
Advanced Computer Architecture Laboratory
EECS Department, University of Michigan
{sraasch,stever}@eecs.umich.edu

Abstract

Simultaneous multithreading (SMT) increases processor throughput by multiplexing resources among several threads. Despite the commercial availability of SMT processors, several aspects of this resource sharing are not well understood. For example, academic SMT studies typically assume that resources are shared dynamically, while industrial designs tend to divide resources statically among threads.

This study seeks to quantify the performance impact of resource partitioning policies in SMT machines, focusing on the execution portion of the pipeline. We find that for storage resources, such as the instruction queue and reorder buffer, statically allocating an equal portion to each thread provides good performance, in part by avoiding starvation. The enforced fairness provided by this partitioning obviates sophisticated fetch policies to a large extent. SMT's potential ability to allocate storage resources dynamically across threads does not appear to be of significant benefit.

In contrast, static division of issue bandwidth has a negative impact on throughput. SMT's ability to multiplex bursty execution streams dynamically onto shared function units contributes to its overall throughput.

Finally, we apply these insights to SMT support in clustered architectures. Assigning threads to separate clusters eliminates inter-cluster communication; however, in some circumstances, the resulting partitioning of issue bandwidth cancels out the performance benefit of eliminating communication.

1. Introduction

Multithreading increases processor throughput by multiplexing several execution threads onto a common set of pipeline resources. Because individual threads rarely saturate the peak execution capacity of a modern CPU, each of *n* threads sharing a pipeline typically achieves more than 1/*n*th of its stand-alone performance, resulting in a net throughput gain.

A key aspect of multithreaded processor design is the division of shared pipeline resources among threads. Early multithreaded processors [1, 2, 28, 31] were built on single-issue, in-order pipelines. In these machines, at most one instruction can occupy a pipe stage at one time, so inter-thread resource sharing is limited to an interleaving of instructions from different threads. With the advent of simultaneous multithreading (SMT) [7, 21, 34, 35] on superscalar, out-of-order machines, the possible dimensions for inter-thread resource sharing increase significantly. The set of instructions processed in a single cycle by a particular pipe stage need not all be from the same thread. (That is, the pipeline may be shared "horizontally" as well as "vertically" [34].) Buffers such as the instruction queue, reorder buffer, and store queue generally contain instructions from multiple threads simultaneously.

Meanwhile, wire delay constraints are forcing designers of aggressive single-threaded processors to divide pipeline resources into clusters to improve physical communication locality [11, 17, 18, 23]. These designs trade increased inter-cluster communication latency for a higher clock rate. When adding SMT to such a design, it seems natural to map these physical divisions to the conceptual division of resources among threads. When multiple independent threads are active, assigning them to separate physical partitions can simplify the design and mitigate inter-cluster communication penalties. However, SMT's efficiency comes from the processor's ability to share execution resources dynamically across threads. Intuitively, the flexibility of dynamic resource allocation provides the potential for higher efficiency than static partitioning.

This paper explores the impact of resource partitioning on SMT processor efficiency. The goal of this paper is to determine the extent to which dynamic sharing of particular processor resources among threads is important to SMT efficiency. Resources for which sharing is less critical can be partitioned statically among active threads without a significant performance penalty. (As in prior SMT designs, we assume that all resources can be dedicated to a single thread to maximize its performance in the absence of other threads; we are concerned in this paper only with allocation of resources when multiple threads are active.)

In this paper, we focus our attention on the execution and retirement portions of the processor pipeline, and examine the impact of partitioning the instruction queue, the reorder buffer, execution bandwidth, and commit bandwidth. We find that static thread-based partitioning of *storage* resources (the instruction queue and the reorder buffer) has a surprisingly modest—and often positive—impact on performance, while partitioning of *bandwidth* resources (at execution and at commit) can degrade SMT efficiency.

This difference arises in part because storage resources are more susceptible to *starvation*, where one or a few threads acquire and hold on to a majority of a structure's elements, leaving insufficient resources for other threads. For storage resources, the benefits due to static partitioning's elimination of starvation generally outweigh any losses due to suboptimal division of resources among threads. In contrast, bandwidth resources are reallocated typically every cycle, making it simple to avoid starving a thread over the course of multiple cycles.

We further find that the elimination of starvation through partitioning significantly reduces the benefit of the well-known instruction count (IC) fetch policy [32] relative to a round robin (RR) scheme. IC fetches from the thread with the fewest instructions in the execution pipeline, maintaining a roughly equal allocation of storage resources among threads. While IC provides a decided performance advantage over RR for fully shared pipelines [32], we show that this advantage stems primarily from IC's ability to avoid starvation. Using partitioning to eliminate starvation achieves nearly the same benefit, allowing the simpler RR scheme to perform within 1% of the IC policy on a fully shared pipeline for two-thread workloads. Although IC maintains a more significant advantage for four-thread workloads on small IQs, RR pulls within 5% for IQs of 64 or more entries. Thus designers may choose the simple RR policy on a simple statically partitioned IQ without foregoing any significant performance opportunity.

We also apply our insights to the design of SMT processors with clustered execution resources. In our simulated architecture, assigning independent threads to distinct clusters eliminates inter-cluster communication, but divides execution bandwidth statically among threads. Allowing threads to share clusters—and thus execution bandwidth—dynamically can provide higher throughput, in spite of the communication delays induced within each thread.

The remainder of the paper begins with a brief discussion of related work. Section 3 describes our experimental methodology. Section 4 analyzes the impact of partitioning of various pipeline structures in isolation. Section 5 applies these results to the design of multithreaded clustered architectures. We conclude in Section 6.

2. Related work

Resource allocation among threads has always been an issue in multithreaded systems, but its manifestation depends strongly on the style of multithreading. As mentioned above, early multithreaded processor designs [1, 2, 19, 22, 28, 31] employed single-issue, in-order pipelines. Once a fetch slot is given to a particular thread, the resulting instruction occupies the corresponding slot in each pipe stage through the rest of the pipeline. Resource partitioning for these machines thus corresponds to the interleaving of threads at fetch. In fact, the granularity of thread interleaving was one of the primary differentiators among these early designs. (The Tera [2] has a multi-operation (VLIW) instruction format, but the principle is the same.)

Some later designs [15, 12, 34] added per-thread instruction issue buffers, decoupling the execution-stage thread interleaving from the fetch-stage interleaving. However, instructions within a single thread still execute with little or no deviation from program order. In these designs, threads do not benefit from buffering more instructions than can be issued in one cycle; small, statically partitioned instruction buffers are thus adequate. Tullsen et al. [34] and the M-Machine [12] explored execution-unit clustering in this context.

Most recent designs feature full out-of-order execution both within and across threads. Among these, academic designs [7, 32, 35] share the instruction queue and reorder buffer dynamically, while commercial designs (the Intel Xeon [21] and Compaq EV8 [24]) partition these structures statically among active threads.[1] This paper is, to our knowledge, the first to compare the performance of these two approaches in this context.

Although Tullsen et al.'s later work [32] assumes dynamic sharing in the instruction queue and reorder buffer, they do experiment with the effects of partitioning the fetch stage. We believe that modern designs, as exemplified by the Intel Pentium 4 [14] and Compaq EV8 [24], demonstrate that wide, pipelined fetch engines can deliver significant instruction bandwidth, and are limited primarily by branch prediction accuracy rather than wire delay, making them unsuitable for physical partitioning. Our focus is on the impact of resource partitions that are likely to arise in the pursuit of single-thread performance.

Gonçalves et al. [13] compared the performance of partitioned and shared instruction queues. Our results corroborate theirs: a partitioned IQ provides higher throughput by avoiding starvation. We extend their work by studying intermediate allocation policies and the influence of fetch policy, by examining partitioning of the reorder buffer and of issue and commit bandwidths, and by analyzing the implications of these results on multithreading clustered processors.

1. The EV8 design supports four thread contexts, and guarantees each thread one fourth of the IQ. When four threads are active, this policy is equivalent to a static partition. With fewer threads, the unallocated quarters are shared dynamically among the active threads [10].

Lo et al. [20] compare the performance of an SMT processor with a chip multiprocessor (CMP)—the apex of resource partitioning—for explicitly parallel programs. Our study investigates the impact of partitioning only selected resources on multiprogrammed workloads. Nevertheless, their results indicate that issue bandwidth partitioning is a greater source of inefficiency for the CMP than IQ partitioning (see Fig. 5 of [20]).

Tullsen and Brown [33] propose flushing a thread that is suffering a long-latency cache miss from the pipeline. This policy not only limits starvation, but frees up additional resources for unblocked threads. They compare their approach with a "pseudo static" scheme that limits the usage of each thread to at most 63% of the IQ slots, and find (as we do) that such a cap provides a speedup over full dynamic sharing. Their flushing scheme provides even higher speedups on most (but not all) of their workloads, albeit on a relatively short 8-stage pipeline. Further study is required to determine whether the benefit of their approach over a partitioned IQ justifies the additional complexity of both dynamic IQ sharing and the thread flushing mechanism even on a longer pipeline with its higher restart latency.

Thread-based cache partitioning is another potentially important topic [30], but represents a sizable and largely orthogonal area of work beyond the scope of this paper.

3. Methodology

We perform our evaluations using a detailed, execution-driven simultaneous multithreading processor simulator modelling a 20-stage pipeline with separate instruction queue, reorder buffer, physical register resources, and a detailed event-driven memory hierarchy [4]. As in SimpleScalar [5], memory instructions are split into an effective-address calculation, which is routed to the IQ, and a memory access, which is stored in a separate load/store queue (LSQ). The simulator executes Compaq Alpha binaries. Our base processor parameters are listed in Table 1.

We use two policies to select the thread that will fetch during each cycle: Round Robin (RR) and Instruction Count (IC) [32]. The RR policy attempts to equalize the fetch opportunities of all threads by giving each thread an opportunity to fetch in turn. The IC policy always gives the thread with the fewest in-flight instructions the first opportunity to fetch.

Our default commit stage aggressively commits the eight oldest committable instructions from the reorder buffer (ROB) across all threads, regardless of their position in the structure.

Benchmark selection is a challenging problem for multithreaded machine studies. The 26 SPEC CPU2000 benchmarks alone generate 351 possible two-thread workloads (including symmetric pairs). We started by compiling all the CPU2000 benchmarks using Compaq's GEM compiler with full optimizations, and simulated each benchmark as a single thread for 100 million instructions using the SimPoints specified by Sherwood et al. [27].[1] We then ran each of the 351 two-thread workloads using the same starting points. We run all our multithreaded workloads until all threads complete at least 100 million instructions each, or until any one thread completes 300 million instructions.

Table 1: Processor parameters

Parameter	Value
Front-end pipeline	10 cycles fetch-to-decode, 5 cycles decode-to-dispatch
Fetch bandwidth	Up to 8 instructions per cycle; max 3 branches per cycle
Branch predictor	Hybrid local/global (a la 21264); global: 13-bit history, 8K-entry PHT local: 2K 11-bit history regs, 2K-entry PHT choice: 13-bit global history, 8K-entry PHT
BTB	4K entries, 4-way set associative
Instr. Queue	Unified int/FP; varied from 32 to 256 entries
Reorder Buffer	3X IQ size
Execution BW	Up to 8 insts / cycle (dispatch, issue, commit)
Function units	8 integer ALU, 4 integer mul, 4 FP add/sub, 4 FP mul/div/sqrt, 4 data-cache rd/wr port
Latencies	integer: mul 3, div 20, all others 1 FP: add/sub 2, mul 4, div 12, sqrt 24 all operations fully pipelined exc. divide & sqrt
L1 split I/D caches	Both: 64 KB, 2-way set assoc., 64-byte lines Instr: 1-cycle latency (to simplify fetch unit) Data: 3-cycle latency, up to 32 simul. misses
L2 unified cache	1 MB, 4-way set associative, 64-byte lines, 10-cycle latency, up to 40 simul. misses, 64 bytes/cycle bandwidth to/from L1 caches
Main memory	100-cycle latency, 8 bytes/cycle bandwidth

Given these results, we selected a representative subset of two-thread workloads by combining the quantitative workload characterization described by Eeckhout et al. [9] with a clustering methodology borrowed from Sherwood et al. [27].

First, we chose a set of 13 statistics that capture interesting aspects of each thread's execution. For this work we used: per-thread committed IPC, wrong-path IPC, IQ occupancy, ROB occupancy, fetch rate, issue rate, load-issue rate, ready-rate, L1 data cache miss rate, L2 miss rate, function unit occupancy, branch predictor accuracy, and the number of branches fetched per cycle. Because we are primarily interested in selecting workloads based on their impact on the behavior of other threads, we normalized each statistic to the single-thread values for the same thread. We then scaled each of these statistics to have a mean of zero and variance of one.

To eliminate the effects of correlation between statistics, we employed principal components analysis [9] using

1. We used the "early single" SimPoints available at www.cs.ucsd.edu/~calder/simpoint/points/early/early-single-sp.html.

Table 2: Workloads

2 Threads	4 Threads
ammp / equake	ammp / eon / equake / equake
ammp / facerec	ammp / equake / gap / mcf
applu / parser	ammp / equake / mesa / sixtrack
art / equake	ammp / equake / parser / mcf
bzip2 / vpr	ammp / facerec / equake / wupwise
crafty / perlbmk	applu / facerec / mcf / parser
eon / equake	art / bzip2 / equake / vpr
equake / equake	art / equake / fma3d / gap
facerec / mcf	bzip2 / equake / equake / vpr
facerec / wupwise	bzip2 / fma3d / mgrid / vpr
fma3d / gap	bzip2 / mesa / sixtrack / vpr
fma3d / mgrid	eon / equake / equake / equake
galgel / gcc	equake / equake / facerec / mcf
mesa /sixtrack	facerec / fma3d / gap / wupwise
parser / mcf	facerec / fma3d / mcf / mgrid
	fma3d / gap / mesa / sixtrack

SAS/STAT [25]. We selected the first 10 principal components for each benchmark. Each workload then maps to two points in 20-dimensional space, one for each ordering of the benchmarks within the workload. To provide a consistent set of points for the clustering algorithm, we choose the point for each workload that sorts the first coordinate of each benchmark in increasing order.

Given the resulting set of points characterizing the workloads, we use linkage clustering to group similar workloads into clusters. As in the work done by Sherwood [27], we calculate the Bayesian Information Criterion (BIC) score for each possible assignment scheme. The cluster assignment with the largest BIC value is the best.

We found that simply minimizing the Euclidean distance between points in a cluster resulted in heavily skewed cluster sizes. Choosing a single representative from each cluster would then result in a set of workloads of widely varying importance. To address this issue, we biased the linkage clustering algorithm to prefer more consistent cluster sizes. This modification resulted in slightly higher BIC scores while decreasing the maximum cluster size by 30%. From the set of resulting cluster assignments, we then chose the smallest workload set that has a BIC score within 12% of the best score.

Finally, we selected a representative workload for each cluster by choosing the point with the smallest Euclidean distance to the centroid of the cluster. These two-thread workloads are given in the left column of Table 2.

To generate a representative set of four-thread workloads, we repeated this procedure using all 120 order-insensitive pairs of these 15 two-thread workloads as input. This process resulted in the 16 four-thread workloads listed in the right column of Table 2.

Results are reported using *weighted speedup* [26, 29]. Raw IPC values can be misleading for multithreaded workloads, as they can be inflated by biasing execution toward high-IPC threads, thus avoiding execution of more challenging low-IPC

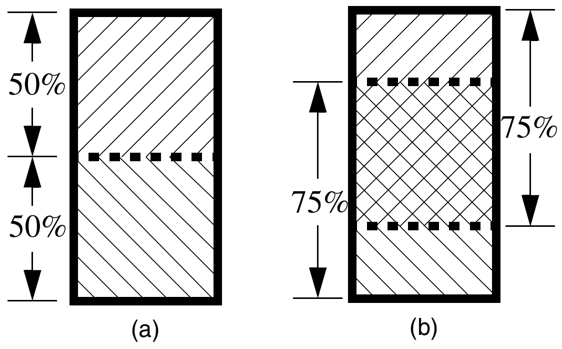


Figure 1. Partitioning Example.
In (a), 50% caps assign half the resource to each of two threads, with no sharing. In (b), 75% caps guarantee 25% to each of two threads, with 50% shared dynamically.

threads. The weighted speedup metric normalizes each thread's performance to the performance of the same program running alone on the same hardware, resulting in that individual thread's "speedup" relative to an equivalent single-threaded machine. (Of course, due to contention from other threads, these values are almost always less than one.) The speedup for a multi-threaded workload is calculated by summing the individual thread speedup values, i.e., the contributions of the individual threads are weighted by their single-thread performance. The resulting value indicates the effective throughput of the workload relative to running each thread alone. For example, if each thread of a two-thread workload ran at 60% of its single-thread performance, the weighted speedup would be 1.2.

4. Analysis of resource partitioning

In this section, we examine the impact of partitioning individual processor resources on the throughput of our representative multithreaded workloads. We focus on the execution portion of the pipeline, examining both storage resources (the instruction queue and reorder buffer) and bandwidth resources (issue and commit).

We explore a range of partitioning options. In a fully partitioned resource, threads are assigned equal and non-overlapping portions. Each of *n* threads is guaranteed one *n*th of the resource, but cannot use more. At the other extreme, a fully shared resource requires threads to compete dynamically for their portion. There is no upper or lower bound on the fraction of the total resource an individual thread can occupy. Between these extremes is a spectrum where each thread is guaranteed some minimum fraction of the resource, and all threads share the remaining fraction.

We parameterize this space by expressing the partitioning in terms of *per-thread caps*, the maximum percentage of a resource available to a single thread. Once a thread reaches its cap for any structure, no further instructions will be fetched for that thread until its usage drops below the cap. For a two-

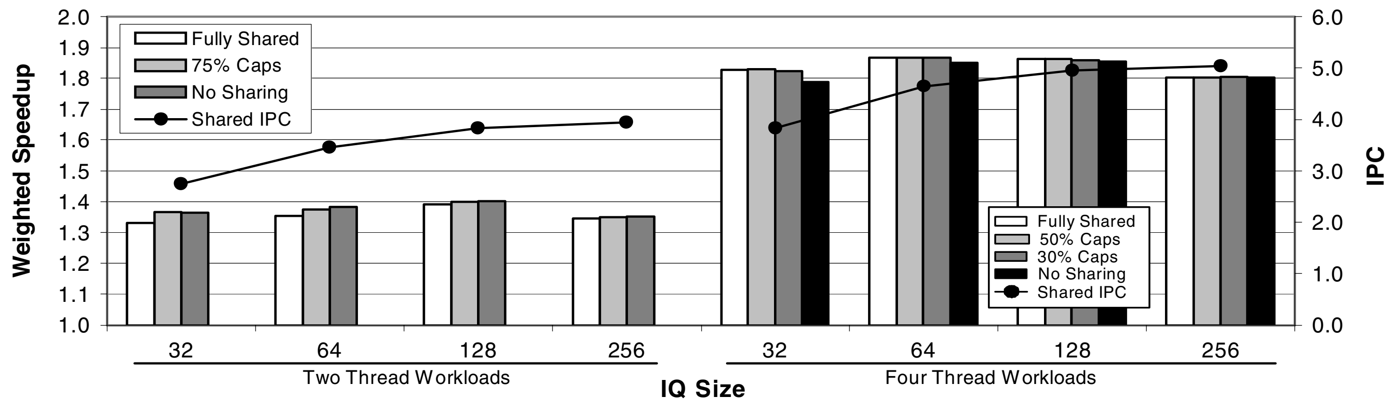


Figure 2. Instruction queue partitioning using the Instruction Count (IC) fetch policy
The vertical bars report weighted speedup for various partitioning schemes, using the scale on the left. The line graphs report the total IPC for the four IQ sizes when the queue is fully shared, using the scale on the right.

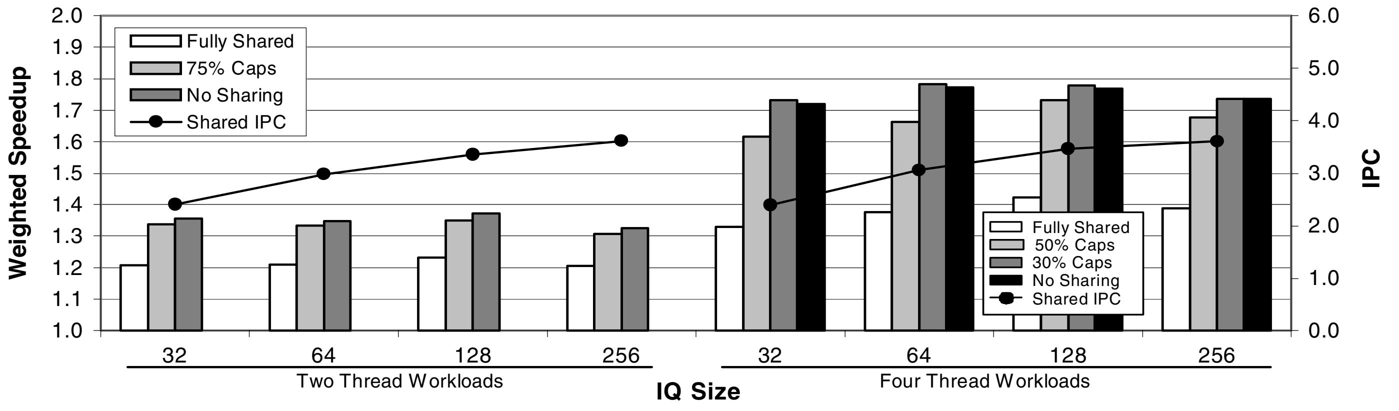


Figure 3. Instruction queue partitioning using the Round Robin (RR) fetch policy
The vertical bars report weighted speedup for various partitioning schemes, using the scale on the left. The line graphs report the total IPC for the four IQ sizes when the queue is fully shared, using the scale on the right.

thread workload, 50% caps implement a fully partitioned resource, i.e., no sharing. A fully shared resource corresponds to 100% caps. We generate intermediate points by varying the cap values between $1/n$ and 100%. For example, as illustrated in Figure 1, 75% caps provide a partitioning scheme where each thread is guaranteed 25% of the resource and the remaining 50% of the resource is shared. For four threads, 30% caps guarantee each thread a minimum 10% share, as any three threads together could occupy at most 90% of the resource.

4.1. Partitioning the instruction queue

The bars in Figure 2 plot the average weighted speedup as the instruction queue's size and partitioning scheme are varied. The left four sets of bars plot two-thread workloads, while the right four sets plot four-thread workloads. These initial results use the IC fetch policy.

One somewhat surprising result is that the weighted speedup occasionally decreases with increasing queue size. This effect arises because the weighted speedup compares the performance of each thread in the workload with its standalone performance on the *same* hardware. Thus the 32-entry IQ results use a single-threaded 32-entry IQ machine as a baseline, while the 64-entry IQ speedups are based on a single-threaded 64-entry IQ machine. Several of the individual benchmarks show substantial performance gains from increasing IQ size, thanks to greater overlapping of cache misses. These performance gains are smaller in the multi-thread case, as the sensitive benchmarks benefit from only a fraction of the overall IQ growth. The overall speedup decreases when the increase in IQ size benefits single-thread performance more than multi-thread performance. Nevertheless, absolute performance of the multi-threaded workloads does increase with IQ size, as expected; the line graphs in Figure 2 plot absolute IPCs of the fully shared configurations, with the scale on the right axis.

The primary result from Figure 2 is that, for a given IQ size, partitioning the IQ has almost no impact on SMT performance. (For a fixed IQ size, the single-thread IPCs are fixed,

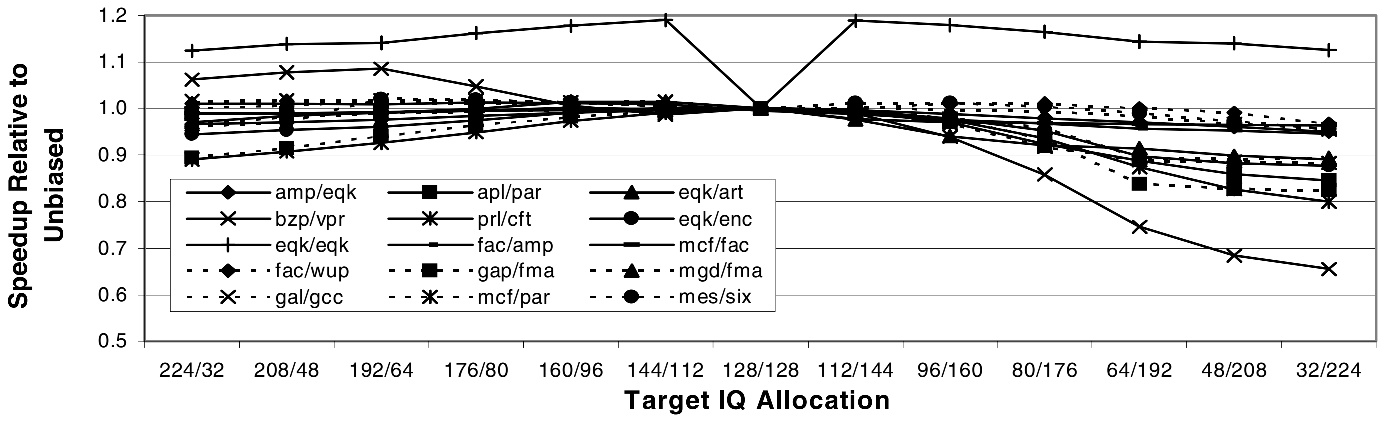


Figure 4. Impact of biased IQ allocation.

so improvements in weighted speedup translate directly to improvements in IPC.) Many of the results even show a slight gain moving from a fully shared IQ to a partially partitioned scheme, because the fully shared IQ does not prevent thread starvation. Although we do not have space to show results for individual workloads, the trend is representative: each specific case shows either a small improvement or no change in performance as the IQ is partitioned.

More surprisingly, with the exception of the four-thread workloads on the small 32-entry IQ, moving all the way to a statically divided IQ has negligible impact. (In the worst case, when four threads are constrained to a quarter of a small 32-entry IQ, we see a 2.2% drop in speedup.)

A key factor in this result is our use of the IC fetch policy [32], which biases the fetch stage to drive pipeline occupancy towards a uniform split. Static IQ partitioning simply enforces this uniformity. Without IQ partitioning, however, IC has some leeway to give a thread more than its share of resources. Our results show that this flexibility is at best needless, and can cause slight performance losses if no other mechanism to prevent starvation is in place.

To factor out the impact of IC on IQ partitioning, we repeated the previous experiments using the round-robin (RR) fetch policy, and report the results in Figure 3. The relatively poor performance of the fully shared IQ shows that RR is much more susceptible to starvation than IC, particularly in the four-thread case. When the instruction queue is fully partitioned, though, the performance of RR is within 5% of IC in every case. Thus most of the benefit of IC can be attained by partitioning the IQ, even with a simpler fetch policy. Designers may opt to trade the small remaining performance difference for avoiding the relative complexity of IC scheme.

Although the impact of static IQ partitioning is negligible under IC, it is possible that some unknown future policy would be capable of allocating IQ entries more intelligently, and would be hampered by a static partitioning. We examine the potential of such a policy by simulating a wide range of unequal allocations using a modified IC fetch policy.

Our “biased IC” policy adds static bias values to the computed instruction counts prior to thread selection. These biases cause the fetch stage to select the thread with the smaller bias value until the biased count values are equalized. For example, if a bias value of 32 is applied to thread A, with no bias on thread B, the processor will fetch from thread A only when thread B has at least 32 more instructions than thread A in the IQ. For a 256-entry IQ, this bias would tend to drive the per-thread occupancies to 112 and 144, respectively.

Figure 4 shows the weighted speedup of biasing the instruction counts for each thread across all our workloads, normalized to that of unbiased IC. We see that, with only two exceptions, allocating IQ entries differentially among threads does not improve performance noticeably. We also see that an inappropriately applied bias can degrade workload performance to a much greater degree. Many of the workloads actually achieve their best performance with no bias (the 128/128 case).

The two exceptions to this trend are bzip2/vpr and equake/equake. The bzip2/vpr workload is achieves up to a 9% speedup if IQ allocation is biased in favor of bzip2. For this workload, the benefit from increased overlap in L2 misses seen by bzip2 causes an increase in total speedup great enough to offset the reduction in throughput seen by vpr. Note that the downside for bzip2/vpr is even more significant, including potential losses of over 30%. The symmetric equake/equake workload is an unusual, and arguably artificial, case. With no bias, the two threads are perfectly synchronized, leading to inter-thread cache conflicts. A slight bias allows the threads to diverge in time, reducing these conflicts.

Although these results certainly do not prove that no worthwhile biased allocation policy exists, it appears to be quite challenging for a fetch policy to provide significant, consistent benefits from non-uniform IQ allocation. These results

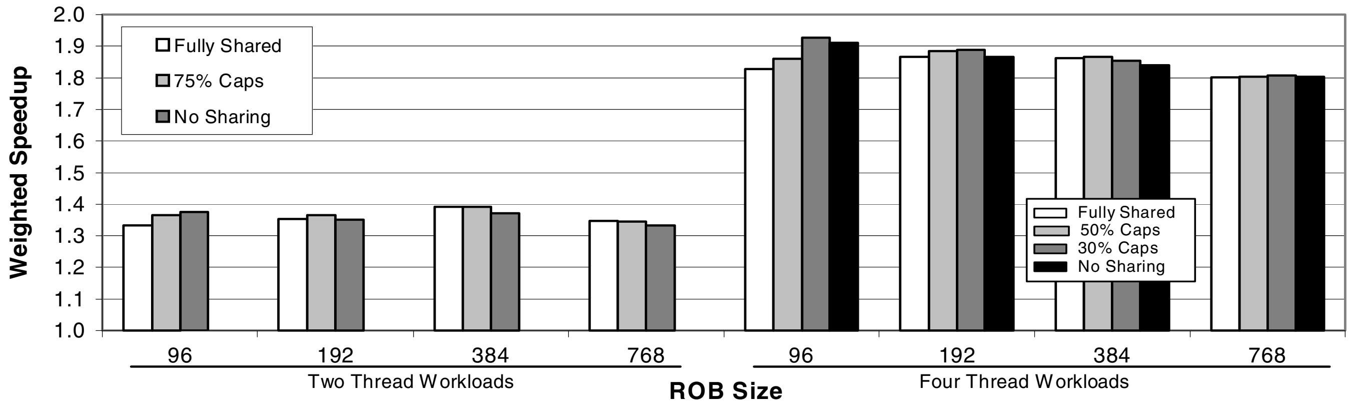


Figure 5. Reorder buffer partitioning

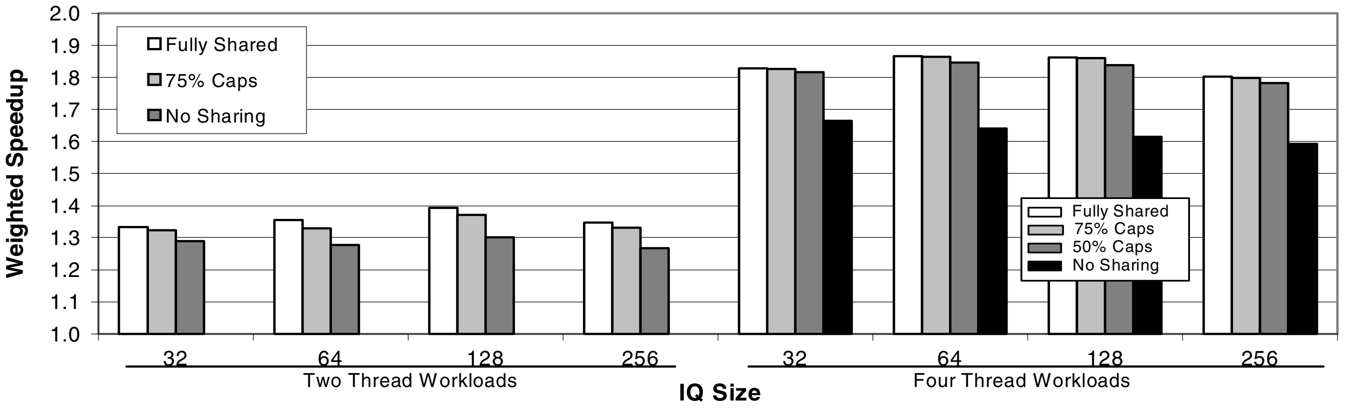


Figure 6. Issue bandwidth partitioning

thus support our argument that the benefits of flexible IQ sharing are likely to be modest at best.

4.2. Partitioning the reorder buffer

The other primary storage resource in the execution portion of the pipeline is the reorder buffer (ROB). The impact of partitioning the reorder buffer is illustrated in Figure 5. Once again, we see that in most cases there is very little performance impact from partitioning this structure statically. The largest performance loss is 1.4%, for two threads on a 384-entry ROB. For the smaller machines, partitioning improves performance by reducing thread starvation, by almost 5% for four threads on the smallest ROB. Our implementation of IC, following the original description [32], counts only instructions in the front end of the pipe and in the IQ, not those in the ROB. Thus it is somewhat less effective at avoiding ROB starvation than IQ starvation in the fully shared case.

4.3. Partitioning issue bandwidth

In this section, we partition issue bandwidth by applying caps to the number of instructions each thread is allowed to issue in a single cycle. Figure 6 presents the results. Unlike the situation with storage resources, performance uniformly *decreases* when issue bandwidth is partitioned. The performance drop from fully shared to fully partitioned is a modest 5% for the two-thread workloads, as each thread can still issue four instructions per cycle. For the four-thread workloads, fully partitioning the issue bandwidth limits each thread to only two instructions per cycle, and performance drops by 9% to 13%, depending on the IQ size.

There are two factors which contribute to the different impact partitioning has on bandwidth compared to storage. First, allocating a storage resource to a particular thread typically commits that resource for an indeterminate number of cycles. For example, once an instruction occupies an IQ slot, it may remain in that slot for several cycles, until it issues or is squashed. Overallocating resources to one thread—for example, when other threads are underutilizing the pipeline due to instruction cache misses—can starve these other threads, as seen in the previous sections. In contrast, nearly all bandwidth resources are reallocated every cycle (except for function units that are not fully pipelined). An overallocation of bandwidth to one thread in one cycle can be reversed in the following cycle, so starvation is avoided easily.^{1}

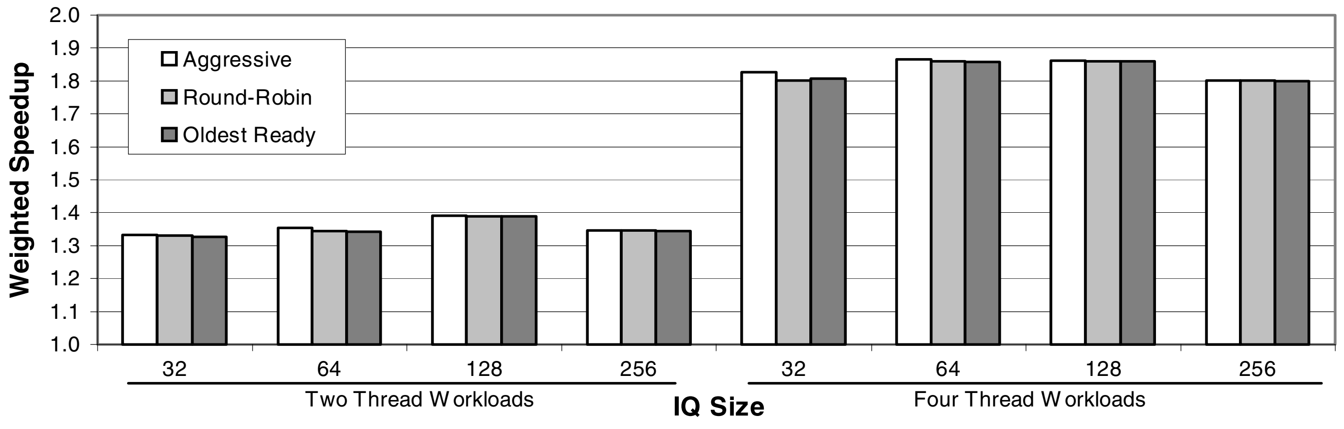


Figure 7. Commit bandwidth partitioning

The second factor differentiating storage partitioning from bandwidth partitioning is the burstiness of demand. The time derivative of a thread's demand for IQ and ROB slots is limited primarily by the system's fetch and commit bandwidths. A pipeline flush caused by a misprediction can cause a sudden decrease in demand, but sudden increases are not possible. In contrast, a thread's cycle-by-cycle demand for issue bandwidth is highly variable: a cache miss can induce several cycles in which no instructions are ready, while the completion of an instruction with a large fan-out can cause a sudden increase in the number of ready instructions. An SMT machine's ability to reallocate issue bandwidth dynamically to meet these bursty demands is a significant factor in its ability to provide increased throughput; static partitioning of issue bandwidth limits this ability.

4.4. Partitioning commit bandwidth

To maintain peak single-thread performance, an SMT processor must be capable of dedicating its full commit bandwidth to one thread. In addition, instructions must commit in program order, requiring centralized control. Thus, unlike the issue stage, we do not see a situation in which single-thread performance considerations will lead to a design that naturally partitions commit bandwidth within a single cycle.

A more challenging aspect of commit processing in SMT machines is the identification of committable instructions across multiple threads. A more likely partitioning of commit bandwidth arises from restricting the complexity of the commit logic by limiting the extent to which the processor can commit instruction from multiple threads in a single cycle. In this section, we examine the impact of partitioning commit bandwidth along this dimension.

1. Dorai and Yeung [8] made similar observations about SMT bandwidth and storage resources in a different context, studying architectural support for background threads that have minimal impact on foreground thread performance.

The commit logic in our base processor can commit completed instructions from one thread even when an older, unexecuted instruction from a different thread occupies a later ROB slot. This idealized logic examines instructions from oldest to youngest; an unexecuted instruction makes only younger instructions from the same thread ineligible to commit.

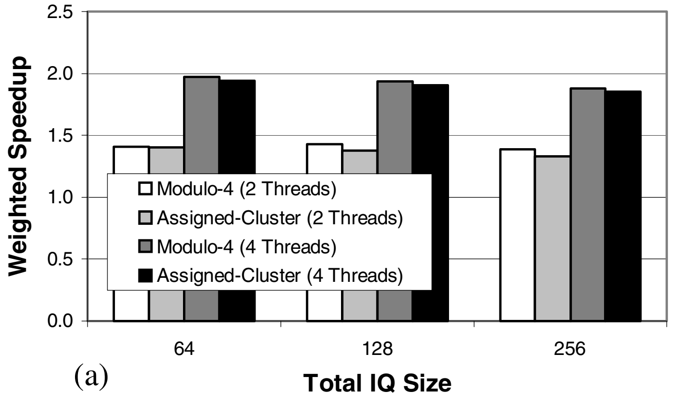
We compare our default scheme, which we label *aggressive*, with two lower complexity commit models: *round robin* and *oldest ready*. Both of these simpler schemes commit from only one thread in each cycle. The round-robin model selects the commit thread in a round-robin fashion. The oldest-ready model identifies and commits instructions only from the thread possessing the oldest committable instruction.

The results obtained from these commit models are given in Figure 7. For two-thread runs, the various models provide almost identical throughput. With only two threads, if the commit stage does not commit from a particular thread on a given cycle, it will likely commit from that thread soon. This short delay does not greatly impact performance due to the buffering effect of the ROB and the idle cycles caused by cache misses and branch mis-predictions.

For the four-thread workloads, there is a small performance degradation for the less-aggressive models when the ROB size is relatively small. In this case, there are more instructions eligible to be committed than in the two-thread case, so the wasted commit slots have a larger relative impact. The ROB becomes full less often as it becomes larger (the ROB has 96 entries for the 64-entry IQ), resulting in fewer dispatch stalls and almost no performance impact.

5. Application to clustering

In the previous section, we have shown that partitioning storage resources has relatively little impact on SMT workload performance, while partitioning execution bandwidth results in a more significant performance degradation. We now apply these results to design issues in multithreaded, clustered



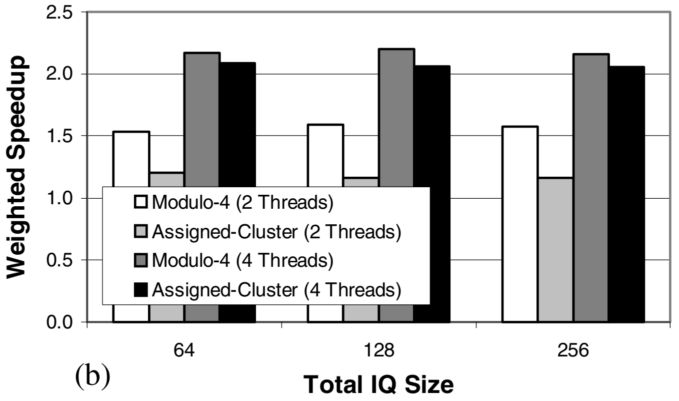


Figure 8. Performance of SMT workloads on a clustered machine
This diagram compares the performance of the modulo-4 instruction distribution scheme vs. the assigned-cluster scheme for (a) a two clusters and (b) four clusters.

microarchitectures. In particular, we consider machines where the execution stage is divided into *n* clusters, each containing an *n*th of the IQ slots and function units. Instructions assigned to an IQ slot in a given cluster execute only on function units assigned to that cluster. We assume that each cluster has a full copy of the register file, and result values are broadcast to all clusters to keep these copies coherent. However, results are available in the cluster where they are generated earlier than in other clusters, reflecting the inter-cluster communication latency. The Compaq Alpha 21264 [17] is a concrete example of this type of clustered architecture, with two clusters and a one-cycle inter-cluster latency.

Conceptually, the performance impact of clustering can be reduced by minimizing inter-cluster communication. In a single-threaded machine, this goal translates into assigning data-dependent instructions to the same cluster. A variety of both static and dynamic schemes have been proposed in the literature [3, 6, 11, 16, 18, 23].

A clustered SMT machine presents the opportunity to reduce inter-cluster communication by assigning threads to specific clusters. When the number of threads is at least equal to the number of clusters, inter-cluster communication will be eliminated completely. However, this policy has the effect of statically partitioning both the overall IQ capacity and issue bandwidth among the threads. As seen in the previous sections, the former is not likely to be a performance problem, but the latter may be.

We compare the performance of two different instruction placement approaches: an "assigned cluster" scheme, which binds threads to specific execution clusters; and a policy that allows each thread to distribute its instructions across all clusters. We use a "modulo-4" policy for the latter, which simply assigns blocks of four consecutive instructions to execution clusters in a round-robin fashion. A similar modulo-3 policy was the most effective non-adaptive distribution policy studied by Baniasadi and Moshovos [3]; we use modulo-4 as it represents a more likely implementation for our eight-wide pipeline.

Figure 8(a) compares these approaches for two- and four-thread workloads on a two-cluster machine with a one-cycle intercluster communication latency. The two approaches are nearly identical in performance; any benefit the assigned-cluster scheme derives from reducing communication is matched by the penalty of partitioning issue bandwidth.

Figure 8(b) reports results for the same experiments for a four-cluster machine with a single-cycle communication latency. For the two-thread workloads in this configuration, each thread is assigned to two clusters. Because the issue bandwidth is partitioned even more finely, with only two issue slots per cluster per cycle, binding threads to clusters is even more constraining. For two threads, the performance of the assigned-cluster scheme is reduced by an average of 25% compared to the modulo-4 scheme. Four-thread performance for the assigned-cluster policy is 5% lower than for the modulo-4 policy.

The importance of reducing inter-cluster communication is naturally dependent on the communication latency. In Figure 9, we plot the speedup of a two-cluster machine with 128 IQ entries running two- and four-thread workloads. We vary the latency from 1 to 8 cycles. We see that for latencies of up to four cycles, the choice of instruction-placement scheme is not critical. As the communications latency increases beyond four cycles, however, the penalty due to this latency increases to the point where the benefit of eliminating it using the assigned-cluster scheme outweighs the loss due to partitioned issue bandwidth.

The performance of the assigned-cluster scheme on our simulated machine drops slightly as the communication latency increases because instructions do not become eligible to commit until they have written back their results in all clusters. Thus increased inter-cluster latency results in increased

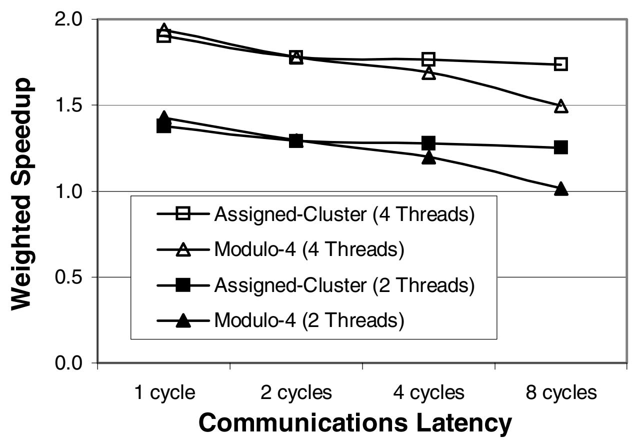


Figure 9. Cost of communication
Performance for various instruction-placement policies and inter-cluster communications latencies.

ROB pressure, which adversely effects all threads. If the ROB entry was freed at the time of the initial local-cluster writeback, the performance curve for the assigned-cluster scheme would extend horizontally from the point corresponding to a one-cycle latency.

Previous works [32] have studied the benefit of SMT versus conventional microprocessors. Our results allow us to compare the benefit of adding SMT to a clustered architecture versus a non-clustered architecture. Referring to Figure 2, we see that SMT applied to a non-clustered architecture improves throughput by an average of 56%, with individual configurations ranging from 33% to 87% improvement.

The speedups reported in Figure 8 are relative to the performance of the individual threads running on the same clustered architecture, and indicate that adding SMT to a clustered architecture improves total throughput an average of 70%, with results ranging from 16% to 120% depending on the number of threads, IQ size, and dispatch policy.

These results indicate that there is a slightly greater benefit in adding SMT to a clustered architecture than to a non-clustered architecture. This does not indicate, however, that the clustered SMT machine will have the greatest throughput. When the clustered throughput is calculated relative to the same baseline used for the non-clustered SMT architecture, we see an average improvement due to SMT of only 48%, with individual configurations ranging from a 5% loss (two thread, four cluster, using assigned-cluster policy) to an 82% improvement (four thread, two cluster, using modulo-4 policy).

6. Conclusions

Simultaneous multithreading's ability to improve throughput is well known. Unlike previous forms of multithreading, SMT enables flexible resource sharing both within and across pipeline stages. However, the significance of dynamic vs. static sharing for specific pipeline resources has not been widely studied.

This paper shows that flexible, dynamic sharing of the instruction queue and reorder buffer is not a significant contributor to SMT efficiency. Designers who opt to simplify implementation by dividing these resources statically among active threads are not forgoing significant performance opportunities. In fact, static partitioning can increase performance by avoiding starvation, and thus also reduces the need for sophisticated fetch policies.

In contrast, dynamic allocation of issue bandwidth across threads is a more significant component of SMT's effectiveness. Architectures with more restrictive policies, while simpler, will be surrendering a performance opportunity.

These results have practical implications in the design of clustered processors, which physically partition pipeline resources to mitigate wire latencies. It seems natural for a clustered multithreaded processor to exploit the absence of inter-thread communication by assigning threads to separate clusters. However, for small inter-cluster latencies, the loss in efficiency from partitioning execution bandwidth can cancel out the benefit of eliminating communication. Distributing threads across clusters provides similar performance using a simple policy; as researchers develop enhanced single-thread distribution policies, this approach to multithreading may prove superior.

Acknowledgments

This work was supported by NSF grant CCR-0105503, by grants from Intel and Compaq, and by a Sloan Research Fellowship.

References

[1] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. April: A processor architecture for multiprocessing. In *Proc. 17th Ann. Int'l Symp. on Computer Architecture*, pages 104–114, June 1990.

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

[3] A. Baniasadi and A. Moshovos. Instruction distribution heuristics for quad-cluster, dynamically-scheduled, superscalar processors. In *33rd Ann. Int'l Symp. on Microarchitecture*, pages 337–347, Dec. 2000.

[4] N. L. Binkert, E. G. Hallnor, and S. K. Reinhardt. Network-oriented full-system simulation using M5. In *Proc. Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads*, Feb. 2003.

[5] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: the SimpleScalar tool set. Technical Report 1308, Computer Sciences Department, University of Wisconsin–Madison, July 1996.

[6] R. Canal, J. M. Parcerisa, and A. González. Dynamic cluster assignment mechanisms. In *Proc. 6th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 133–142, Jan. 2000.

[7] G. E. Daddis Jr. and H. C. Torng. The concurrent execution of multiple instruction streams on superscalar processors. In *Proc. 1991 Int'l Conf. on Parallel Processing*, volume I, pages 76–83, Aug. 1991.

[8] G. K. Dorai and D. Yeung. Transparent threads: Resource allocation in SMT processors for high single-thread performance. In *Proc. 11th Ann. Int'l Conf. on Parallel Architectures and Compilation Techniques*, Sept. 2002.

[9] L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Quantifying the impact of input data sets on program behavior and its applications. *Journal of Instruction-Level Parallelism*, 5, Feb. 2003.

[10] J. Emer. Personal communication, Nov. 2002.

[11] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. The multicluster architecture: Reducing cycle time through partitioning. In *30th Ann. Int'l Symp. on Microarchitecture*, pages 149–159, Dec. 1997.

[12] 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 Ann. Int'l Symp. on Microarchitecture*, pages 146–156, Dec. 1995.

[13] R. Gonçalves, E. Ayguadé, M. Valero, and P. Navaux. Performance evaluation of decoding and dispatching stages in simultaneous multithreaded architectures. In *Proc. 13th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Sept. 2001.

[14] 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*, 5(1), Feb. 2001.

[15] H. Hirata et al. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *Proc. 19th Ann. Int'l Symp. on Computer Architecture*, pages 136–145, May 1992.

[16] G. A. Kemp and M. Franklin. PEWs: A decentralized dynamic scheduler for ILP processing. In *Proc. 1996 Int'l Conf. on Parallel Processing (Vol. I)*, pages 239–246, 1996.

[17] R. E. Kessler. The Alpha 21264 microprocesor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[18] H.-S. Kim and J. E. Smith. An instruction set architecture and microarchitecture for instruction level distributed processing. In *Proc. 29th Ann. Int'l Symp. on Computer Architecture*, pages 71–81, May 2002.

[19] J. Laudon, A. Gupta, and M. Horowitz. Interleaving: A multithreading technique targeting multiprocessors and workstations. In *Proc. Sixth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 308–318, Oct. 1994.

[20] J. Lo, S. Eggers, J. Emer, H. Levy, R. Stamm, and D. Tullsen. Converting thread-level parallelism into instruction-level parallelism via simultaneous multithreading. *ACM Trans. Computer Systems*, 15(3):322–254, Aug. 1997.

[21] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1), Feb. 2002.

[22] M. Nemirovsky, F. Brewer, and R. C. Wood. DISC: Dynamic instruction stream computer. In *24th Ann. Int'l Symp. on Microarchitecture*, pages 163–171, Nov. 1991.

[23] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proc. 24th Ann. Int'l Symp. on Computer Architecture*, pages 206–218, June 1997.

[24] R. P. Preston et al. Design of an 8-wide superscalar RISC microprocessor with simultaneous multithreading. In *Proc. 2002 IEEE Int'l Solid-State Circuits Conf.*, pages 334–335, Feb. 2002.

[25] SAS Institute. SAS/STAT. Computer software. http://www.sas.com.

[26] Y. Sazeides and T. Juan. How to compare the performance of two SMT microarchitectures. In *Proc. 2001 IEEE Int'l Symp. on Performance Analysis of Systems and Software*, Nov. 2001.

[27] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proc. Tenth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, pages 45–57, Oct. 2002.

[28] B. J. Smith. Architecture and applications of the HEP multiprocessor computer system. In *Proc. of the Int. Soc. for Opt. Engr.*, volume 298, pages 241–248, 1981. Real-Time Signal Processing IV.

[29] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Proc. Ninth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, pages 234–244, Nov. 2000.

[30] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proc. 8th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2002.

[31] J. E. Thornton. Parallel operation in the Control Data 6600. In *AFIPS Proceedings, 1964 Fall Joint Computer Conference*, pages 33–41, 1964. Reprinted in "Computer Structures: Principles and Examples", Sieworiek, Bell, and Newell, McGraw-Hill, 1982.

[32] D. 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 *Proc. 23rd Ann. Int'l Symp. on Computer Architecture*, pages 191–202, May 1996.

[33] D. M. Tullsen and J. A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *34th Ann. Int'l Symp. on Microarchitecture*, pages 318–327, Dec. 2001.

[34] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proc. 22nd Ann. Int'l Symp. on Computer Architecture*, pages 392–403, June 1995.

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