Predictable Performance in SMT Processors

Francisco J. Cazorla
DAC, UPC
Spain
fcazorla@ac.upc.es

Peter M.W. Knijnenburg
LIACS, Leiden University
The Netherlands
peterk@liacs.nl

Rizos Sakellariou
University of Manchester
United Kingdom
rizos@cs.man.ac.uk

Enrique Fernández
Univ. de Las Palmas de GC
Spain
efernandez@dis.ulpgc.es

Alex Ramirez
DAC, UPC
Spain
aramirez@ac.upc.es

Mateo Valero
DAC, UPC
Spain
mateo@ac.upc.es

ABSTRACT

Current instruction fetch policies in SMT processors are oriented towards optimization of overall throughput and/or fairness. However, they provide no control over how individual threads are executed, leading to performance unpredictability, since the IPC of a thread depends on the workload it is executed in and on the fetch policy used.

From the point of view of the Operating System (OS), it is the job scheduler that determines how jobs are executed. However, when the OS runs on an SMT processor, the job scheduler cannot guarantee execution time constraints of any job due to this performance unpredictability.

In this paper we propose a novel kind of collaboration between the OS and the SMT hardware that enables the OS to enforce that a high priority thread runs at a specific fraction of its full speed. We present an extensive evaluation using many different workloads, that shows that this mechanism gives the required performance in more than 97% of all cases considered, and even more than 99% for the less extreme cases. At the same time, our mechanism does not need to trade off predictability against overall throughput, as it maximizes the IPC of the remaining low priority threads, giving 94% on average (and 97.5% on average for the less extreme cases) of the throughput obtained using instruction fetch policies oriented toward throughput maximization, such as *icount*.

Categories and Subject Descriptors

C.1 [**Computer Systems Organization**]: PROCESSOR ARCHITECTURES; D.4 [**Software**]: OPERATING SYSTEMS; C.1.3 [**PROCESSOR ARCHITECTURES**]: Other Architecture Styles

CF'04, April 14–16, 2004, Ischia, Italy.

General Terms

Design, Performance, Measurement

Keywords

ILP, SMT, Performance Predictability, Real Time, multithreading, thread-level parallelism, Operating Systems

1. INTRODUCTION

In Simultaneous Multithreaded (SMT) architectures, first introduced in [11][23][25], several threads[1] are running together, sharing resources at the micro-architectural level. This allows an SMT to increase throughput with a moderate area overhead over a superscalar processor [2][3][13][17]. In an SMT, the front end of a superscalar is adapted in order to be able to fetch from several threads while the back end is shared among the threads. A *fetch policy*, e.g., *icount* [22], decides how instructions are fetched from the threads, thereby implicitly determining the way internal processor resources, like rename registers or IQ entries, are allocated to the threads. The common characteristic of many existing fetch policies is that they attempt to increase throughput and/or fairness [16] by stalling or flushing threads experiencing L2 misses [4][7][15][21], or reduce the effects of mispeculation by stalling on hard-to-predict branches [14]. These fetch policies have been quite successful in that they increase throughput and fairness, or reduce mispeculation.

However, a problem with all the fetch policies proposed until now is that it is unpredictable what the performance of a certain thread in a workload actually is. Figure 1 shows the IPC of the `gzip` benchmark when it is run alone (full speed) and when it is run with other threads using two different fetch policies, *icount* [22] and *flush* [21]. As we can see, its IPC varies much, depending on the fetch policy as well as characteristics of the other threads running in the context. For instance, in a 3 thread context, its IPC can be higher than in a 2 thread context. This is caused by the fact that management of resources (IQ entries, registers, FUs) is not explicit. Currently, there is no fetch policy that can enforce that resources are allocated to a particular thread in such a

[1]In this paper we use the terms job and thread interchangeably.

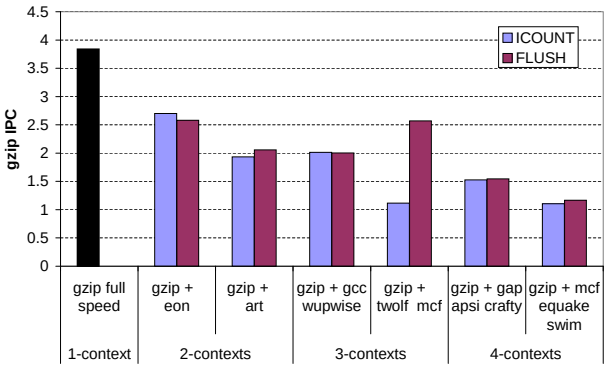

Figure 1: IPC of gzip for different contexts and different fetch policies

way that this thread would perform similarly regardless its context.

The key issue is that in traditional SMTs, although the OS still assembles the workload, it is now the processor that decides how to execute this workload and hence part of the traditional responsibility of the OS has "disappeared" into the processor. The result is that running times of jobs become unpredictable and depend on the characteristics of the context in which these jobs execute. Hence, the job scheduler cannot control how fast threads are executed. One consequence is that the OS may not be able to guarantee time constraints on the execution of a thread if that thread is to be run concurrently with other threads, even though the processor has sufficient resources to do so. The only way out is that the OS runs time critical applications alone on the machine so that its knowledge about the execution time of an application can be exploited to satisfy deadlines. However, this goes against the spirit of SMT processors and obviously under-utilizes resources. We would like to be able to run several applications at the same time and still be able to predict the execution time of at least one application. To deal with this situation, the OS should be able to exercise more control over how threads are executed and how they share the processor's internal resources. The hardware should guarantee the OS some kind of Quality of Service that can be used by the OS to better schedule jobs.

Thus, if we want to be able to control the speed of a particular thread on an SMT processor, current approaches to resource management by means of instruction fetch policies are no longer adequate. Hence, a new paradigm for resource management inside SMT processors is required.

We would like to have a mechanism that controls the execution time of certain threads, independent of the other threads in the workload, and that enables a much more powerful dialogue between the OS and the SMT hardware. Such a dialogue and control over resources is needed for general purpose high performance SMT usage. It might also be particularly useful in soft real-time and embedded systems, for which there is a growing interest in using SMT processors due to their high throughput at low cost [1]. In such applications, the OS needs to satisfy certain real-time requirements for the jobs running. The SMT should be in a position to interact with the OS in order to meet such requirements.

In this paper, we present a new OS/SMT collaboration. The OS selects a workload consisting of several programs and indicates to the processor that a certain thread should be considered as a High Priority Thread (HPT) and must execute at a certain target IPC that represents a given percentage of its full speed. We propose and evaluate a novel mechanism that uses a dynamic allocation of resources and that accomplishes the previous goal more than 99% of times, while allows full use of all internal resources achieving 94% of the throughput obtained with *icount*. The solution of above problem means that we have got execution time predictability in High Performance SMT Processors and also, that a capability for full execution control can be offered to the OS with minimum performance penalty. Hence, we do not trade off QoS against performance.

This paper is structured as follows. We present our novel approach to the OS/SMT collaboration in section 2. In Section 3, we give a discussion of the problems addressed in this paper. Section 4 presents our mechanism to solve these problems. In section 5, we discuss our experimental environment. Section 6 shows the experimental results. In section 7, we discuss related work. Finally, Section 8 is devoted to conclusions.

2. A NOVEL APPROACH TO COLLABORATION BETWEEN OS AND SMT

It is clear that a hardwired fetch policy that is geared towards one particular goal (e.g., throughput maximization) cannot solve this problem in the general case. Therefore, we propose to address it from a different point of view. We consider the SMT as having a collection of sharable resources. We add a mechanism to control how this sharing is exactly done. In our view, there is a tight interaction between OS and processor. The OS gives the processor requests and/or mandates. The processor, in turn, decides upon how to fulfill these requests by assigning resources to threads.

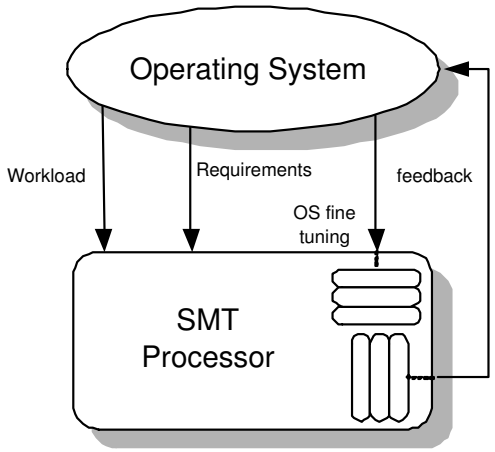

Figure 2: Interaction between OS and architecture to enforce QoS requirements.

We approach resource sharing for SMT as a *Quality of Service* (QoS) problem. Instead of trying to find a policy that would optimize for a predefined objective, we assume that there is an OS-processor interaction in which the SMT

processor supports policies that can be tuned by the OS. Our view is that this can be achieved by having the SMT processor provide 'levers' through which the OS can fine tune the internal operation of the processor as needed. A graphical illustration is given in Figure 2. Such levers can include prioritizing instruction fetch for particular threads, reserving parts of the resources like IQ entries, prioritizing issue, etc. Existing policies do not allow such fine tuning since they assume that there is a statically predefined objective. Their orientation towards achieving this objective (whether this is maximization of total IPC or minimization of power consumption) does not leave space for much flexibility, when an application's needs dictate so.

This approach can in principle be applied in very many situations. In this paper, we concentrate on one particular instance of the QoS approach, namely, guaranteeing a required performance for a High Priority Thread and giving best effort to the remaining Low Priority Threads, maximizing their throughput.

3. A QUALITY OF SERVICE PROBLEM

In this paper we focus on general high performance out-of-order SMT processors. We propose a mechanism that provides control over the execution speed of a designated High Priority Thread. At the same time, we want to give best effort to the remaining Low Priority Threads and maximize their throughput.

In traditional general purpose systems, the OS can decide to give the machine for a certain fraction of the time to a particular job to meet a deadline, or it can decide the moment when a job needs to be fired up the latest. The job scheduler in the OS can also decide to assemble a particular job schedule in order to maximize the utilization of peripherals, etc. Compared to a traditional superscalar processor, there exists one degree of freedom more in an SMT processor: jobs can be given a certain *share* of the available resources. This gives the OS more freedom in scheduling jobs and allows for the concurrent execution of other important but not so time critical jobs. Moreover, in the present approach the total throughput of the system can actually increase compared to existing fetch policies like *icount*, as we show in Section 6. This mechanism opens a kind of collaboration between the Operating System and the SMT processor, not addressed before in the literature. A possible use for such a mechanism can be soft real time systems. Therefore, in this paper we focus on the following challenge:

> *Given a workload of N applications*[2] *and a High Priority Thread (HPT) in this workload, find a resource management policy that ensures that the HPT runs at (at least) a given target IPC that represents X% of its IPC when it would run alone on the machine, while at the same time maximize the throughput for the remaining* $N-1$ *Low Priority Threads (LPTs).*

Note that a naive option is to run the HPT alone on the machine. However, in that case, the throughput of the LPTs would be zero, which is clearly not the maximal throughput they could achieve. We assume that the OS has some goals in mind and decides that a certain high priority job needs a certain percentage of its full speed in order to satisfy this goal. The OS subsequently instructs the processor to realize this percentage. We show that our solution indeed is capable of fulfilling this requirement. Note also that the decision about which jobs are co-scheduled is still the responsibility of the OS. The processor simply accepts a workload as given to it by the OS and, moreover, it accepts a QoS requirement in the form of a high priority job and a percentage of the full speed of this job it is required to realize.

[2]We assume throughout the paper that the workload is smaller than or equal to the number of hardware contexts supported by the processor.

4. SOLUTION OF THE QOS PROBLEM

A key point in our mechanism is that programs experience different phases in their execution in which their IPC varies significantly. Hence, if we want to realize a certain percentage of the full speed of a program, we need to take into account this variable IPC. We illustrate this by an example. Figure 3 shows local IPC values for `gap` for a period of 4.5 million cycles in which each value has been determined over an interval of 15,000 cycles. Assume that the OS requires the processor to run this thread at 80% of its full speed. The solid line is the average IPC for this period and the dashed line represents the value to be achieved by the processor. It is easily seen that during some periods it is impossible to achieve this 80% of the global IPC, even if the thread were given all the processor resources. Moreover, if the processor achieves this 80% of the global IPC during the first part of the interval and subsequently gives all resources to this thread to achieve full speed during the second part, then the overall IPC value it would realize would be lower than 80% of the global IPC.

The basis of our mechanism for dynamic resource allocation rests on the observation that in order to realize $X\%$ of the overall IPC for a given job, it is sufficient to realize $X\%$ of the maximum possible IPC *at every instance* through the execution of that job. This is illustrated in Figure 3 by the bold faced curve labeled "80% of local IPC". Hence, the mechanism needs to determine the variations in IPC of the HPT. In order to do this, we employ two phases in our proposed mechanism that are executed in alternate fashion.

- During the first phase, the *sample phase*, all shared resources are given to the HPT and LPTs are temporarily stopped. As a result, we obtain an estimate of the current full speed of the HPT during this phase which we call the *local IPC*.
- During the second phase, the *tune phase*, our mechanism dynamically varies the amount of resources given to the HPT in order to achieve a *target IPC* that is given by the local IPC computed in the last sample period times the required percentage given by the OS.

Obviously, if we are able in the sample and tune phases to measure and realize a percentage X if the real IPC, then we obtain an overall IPC for the HPT that is about $X\%$ of the IPC it would have had when executed alone on the processor. In the next two subsections we discuss the sample phase and the tune phase in more detail.

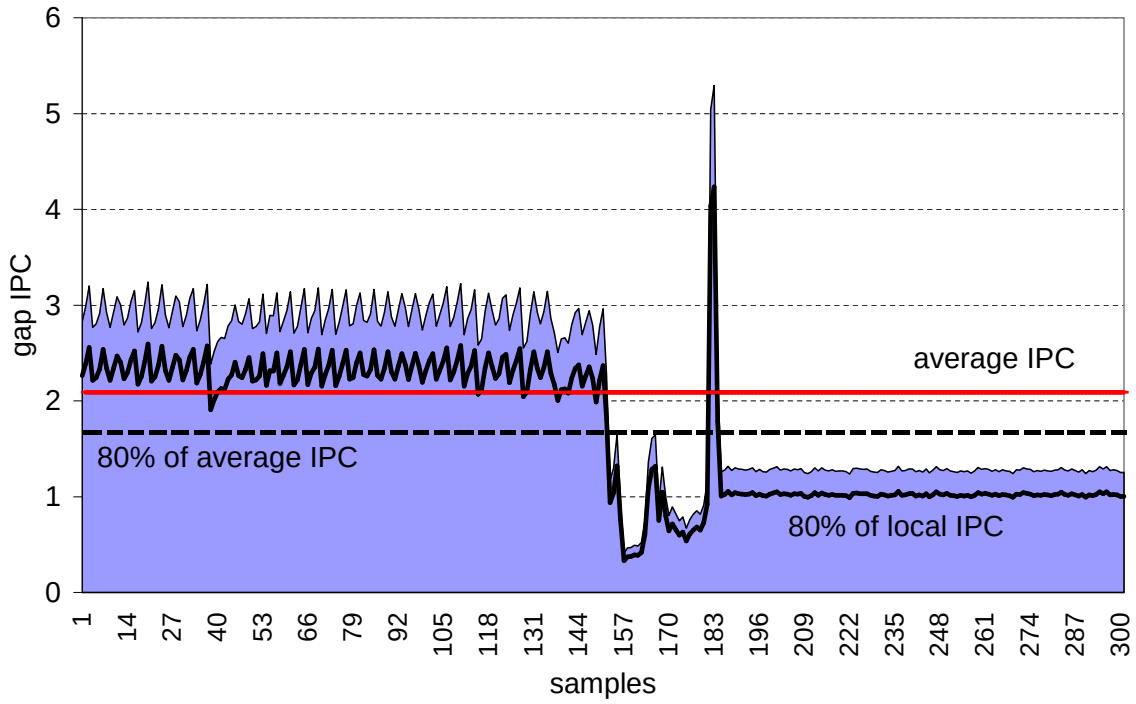

Figure 3: Local IPC of the gap benchmark

4.1 Sample phase: determining the local IPC of the HPT

During the sample phase, we determine the local IPC of the HPT by giving it all shared resources and hence suspending the LPTs momentarily. Note that the longer the sample phase, the longer the time that the SMT is dedicated to only one thread, reducing its overall performance and starving the Low Priority Threads. Hence, we have to determine the local IPC of the HPT thread as fast as possible. In this section we show that interference in shared resources renders determination of the isolated IPC of the HPT difficult and also propose mechanisms to counteract these interferences. The resources shared among threads are the following.

Caches the L1 data and instruction caches, and the L2 unified cache.

TLB the data and instruction TLB.

Branch Predictor branch target buffer (BTB) and the direction predictor (PHT).

Other shared resources the issue queues (integer, floating point and, load/store) and the physical registers (integer and floating point).

Neither the issue queues nor the physical registers present a problem because both these resources are fully dedicated to the HPT during the sample phase. However, there is interference from the LPTs in the other shared resources. In order to get more insight into this interference, we show in Figure 4 how many inter-thread conflicts the HPT suffers during a 100,000 cycle-long sample phase, averaged over the entire run of a workload consisting of `twolf` as HPT and `mcf`, `equake`, and `swim` as LPTs. We observe that as the sample phase progresses, the number of conflicts goes towards zero for the instruction cache, data cache, TLB, and BTB. From the figure we conclude that after a *warm-up period* of 50,000 cycles most interference in these shared resources is removed. The branch predictor (PHT) takes much longer to clear: we have measured that it takes more than 5,000,000 cycles before inter-thread misses have disappeared. However, we have also measured that this interference is mostly neutral, giving a small loss in the PHT hit rate less than 1%. Hence, we ignore the interference in the PHT. The interference in the L2 cache is more serious: it extends for about 1.5 million cycles and gives rise to a significant performance degradation (more than 30% for some benchmarks). This high number of cycles shows that we cannot deal with the interference in the L2 cache by simply extending the warmup phase. We address this problem below.

The solution we propose consists of splitting each sample period into two sub-phases.

- During the first sub-phase, the *warmup phase*, that consists of 50,000 cycles, the HPT is given all resources but its IPC is not yet determined.

- In the second sub-phase, the *actual-sample phase*, that consists of 10,000 cycles, the HPT keeps all resources and moreover its IPC is determined.

The duration of the actual-sample phase is based on observations in [24]. In this way, we try to re-create as much as possible the state of the processor as this state would have been when the processor would have executed the HPT in isolation.

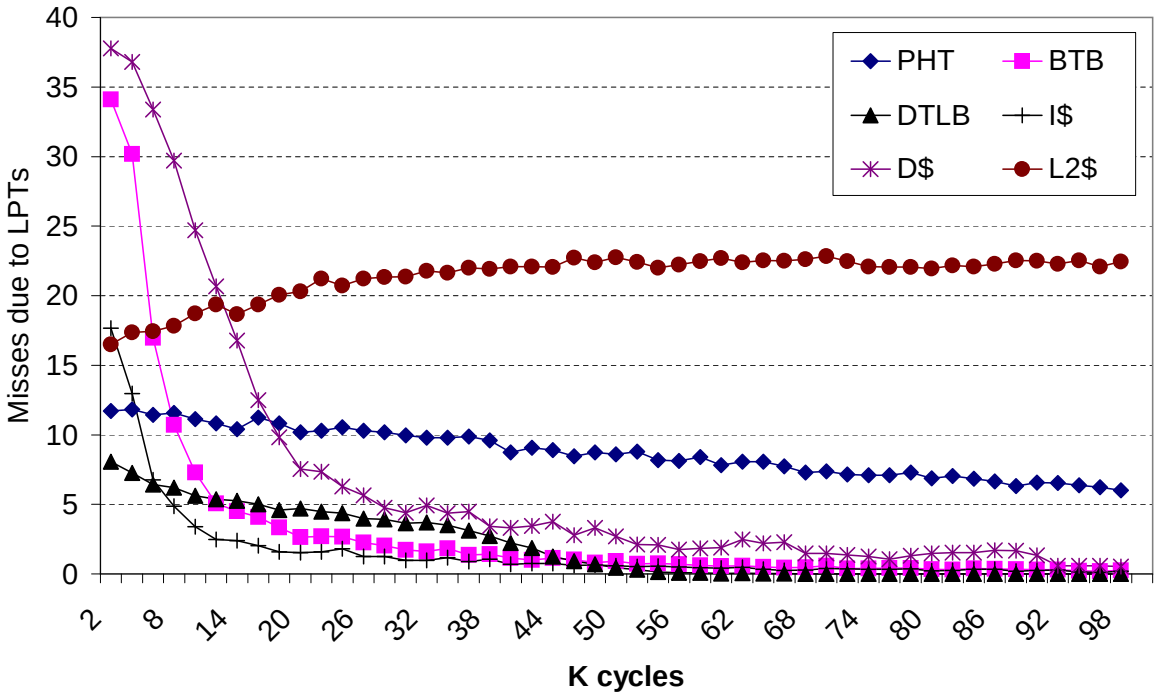

Figure 4: Misses suffered by the HPT due to the LPTs interference

Our solution to inter-thread interference in the L2 cache consists of *partitioning* this cache into two parts. One part is dedicated to the HPT and the other part can be used by the entire workload, LPTs and HPT. In order to meet the demands for varying workloads and program behavior, we employ *dynamic cache partitioning*. We assume that the L2 cache is N-way set associative, where N is not too small: in our simulator, L2 is 8-way set associative. We use a bit vector of N bits that indicates which "ways" or elements in the set are reserved for the HPT. The cache behaves like a "normal" cache, except in the placement and replacement of data. The HPT is allowed to use the entire L2 and only the LPTs are restricted to use a restricted subset of all the ways that exist in a set. An extra, most significant, LRU bit is required for each way. This bit is set to one for the reserved ways and to zero for the other ways so that the lines reserved for the HPT always have a higher access count than the lines in the shared part of the cache. The replacement algorithm is the standard LRU replacement algorithm. Hence, when it is invoked for the LPTs, it always selects a line that is a member of the shared part of the cache. When the replacement algorithm is invoked for the HPT, we mask this extra bit and, in addition, on a replacement we first select a victim line that belongs to an LPT, if possible. If there does not exist such a line, the LRU line of the entire set is selected as the victim. Note that this extension allows the cache to be used normally when the SMT does not execute a workload with a designated HPT. Either all bits are used in the LRU replacement, or the extra bit is always masked. In [5] a different cache partitioning technique called column caching has been proposed. However, this technique addresses a much more general problem of cache partitioning. Therefore, the technique is too heavy weight to be used for our present purposes where the simply mechanism described below suffices.

Based on this cache partitioning scheme, we propose an iterative method that dynamically varies the number of ways reserved for the HPT.

- During each sampling period, every time the HPT suffers an inter-thread miss a counter is incremented.
- At the end of the sampling period, if the value of the counter is higher than a threshold of 8, the number of ways reserved for the HPT is increased by 1. The value of this threshold has been determined empirically.
- If, on the other hand, the counter is lower than the threshold, this number is decreased by 1.

In this way, if the HPT experiences few L2 misses, we reduce the number of ways reserved for it. Likewise, if it experiences many misses, then we increase the number of reserved ways.

4.2 Tune phase: realizing the target IPC

After every sample phase (60,000 cycles), there is a tune phase of 1.2 million cycles where we try to achieve the required percentage of the local IPC measured in the previous sample period. This required percentage of the local IPC is called the *local target IPC*. This is accomplished by giving priority to the HPT in the utilization of the fetch and the issue bandwidth, and dividing each resource in two parts: a part reserved for the HPT and the remaining part that is dedicated to the LPTs. The amount of resources dedicated to the HPT is dynamically varied in order achieve the local target IPC, as follows.

- Each tune phase is split in sub-phases of 15,000 cycles.
- At the end of every sub-phase, the average IPC of the HPT is computed.
- If this IPC is lower than the local target IPC, then the amount of resources given to the HPT is increased.
- Otherwise, if this IPC is higher than the local target IPC, then the amount of resources given to the HPT is decreased.

This increase and decrease is by a fixed amount that depends on the type of resource and the number of its instances. We divide this number by a *granularity factor* of 8 to obtain the *change amount*. The value of the granularity factor has been determined empirically. This change amount is used in the algorithm above for decreasing and increasing the amount of resources dedicated to the HPT.

Until now, in each sub-phase of the tune phase the only target is to achieve the given percentage of the local IPC measured in the last sample phase. However, these sampled local IPC values are sometimes lower than they should be due to interference from LPTs. In order to counteract this effect, we also must take into account the *global IPC* of the HPT: at the end of each sub-phase we check whether the total IPC of the HPT up to this cycle is lower than the target IPC given by the OS. We introduce a *compensation term* for this effect.

- This term is initially zero.
- If the total IPC is smaller than the target IPC, we increase the compensation term by 5. This value of 5 has been determined empirically.
- On the other hand, if the total IPC is larger than the target IPC, we decrease this term by 5.

However, we stipulate that the compensation term is not smaller than zero. Also, X plus the compensation term saturates at 100. Hence, the local target IPC to achieve is given by (where X is the target percentage of the full speed)

$$\text{local target IPC} = \frac{(X + \textit{compensation term})}{100} \times \text{local IPC}$$

5. EXPERIMENTAL SETUP

To evaluate the performance of our mechanism, we use a trace driven SMT simulator derived from SMTSIM [23]. The simulator consists of our own trace driven front-end and an improved version of SMTSIM's back-end. The simulator allows executing wrong path instructions by using a separate basic block dictionary that contains all static instructions. Table 1 shows the main parameters of the simulated processor, which has a 12-stage pipeline. We use a 2.8 fetch mechanism, which means that we can fetch 8 instructions per cycle from up to two threads. First, we fetch instructions from the HPT and if there is some unused fetch bandwidth, we fetch instructions from the LPTs, breaking ties with *icount*.

Traces are collected of the most representative 300 million instruction segment, following the idea presented in [18]. The workloads consist of all programs from the SPEC2000 integer benchmark suite. Each program is executed using the reference input set and compiled using the DEC Alpha AXP-21264 C/C++ compiler with the *-O2 -non_shared* options. Programs are divided into two groups based on their cache behavior (see Table 2): those with an L2 cache miss rate higher than 1%[3] are considered memory bounded (MEM). The others are considered ILP.

We consider combinations where the High Priority Thread is ILP or MEM, and where the Low Priority Threads are ILP or MEM. The simulation ends when the HPT thread ends. Any LPT in the workload that finishes earlier is re-executed. A workload is identified by three parameters: the type of the HPT, the type of the LPTs, and the number of threads. For example, a workload of type IM3 means that the HPT is ILP, the LPTs are MEM and that it contains 3 threads (one HPT and two LPTs). For each workload type, we create four different sets of threads to avoid that our results are biased toward a specific set of threads by taking all possible combinations from Table 3. In the result section we present average results for each group in each workload.

We do not include workloads with more than 4 threads for several reasons. First, several studies [9][10][23] have shown that for workloads with more than 4 contexts, performance saturates or even degrades. This situation is counter produc-

[3]The L2 and L1 miss rate are calculated with respect to the number of dynamic loads

Table 1: Baseline configuration

Processor Configuration	
Fetch /Issue /Commit Width	8
Fetch Policy	ICOUNT 2.8
Queues Entries	32 int, 32 fp, 32 ld/st
Execution Units	6 int, 3 fp, 4 ld/st
Physical Registers	320 int, 320 fp
ROB Size / thread	256 entries
Branch Predictor Configuration	
Branch Predictor	16K entries gshare
Branch Target Buffer	256 entry, 4-way associative
RAS	256 entries
Memory Configuration	
Icache, Dcache	64K bytes, 4-way, 8-bank, 64-byte lines, 1 cycle access
L2 cache	512K bytes, 8-way, 8-banks, 10 cycles lat., 64-byte lines
Main Memory latency	100 cycles
TLB miss penalty	160 cycles

Table 2: Cache behavior of threads and classification based on this behavior

	L2 miss rate	Thread type
mcf	29.6	MEM
twolf	2.9	
vpr	1.9	
parser	1.0	
gap	0.7	ILP
vortex	0.3	
gcc	0.3	
perlbmk	0.1	
bzip2	0.1	
crafty	0.1	
gzip	0.1	
eon	0.0	

	L2 miss rate	Thread type
art	18.6	MEM
swim	11.4	
lucas	7.47	
equake	4.72	
apsi	0.9	ILP
wupwise	0.9	
mesa	0.16	
fma3d	0.01	

Table 3: Workloads

	HPT	LPTs
ILP	gzip	eon
	bzip2	gcc, wupwise
	mesa	gap, apsi, crafty
	fma3d	
MEM	twolf	art
	mcf	twolf, mcf
	art	mcf, equake, swim
	lucas	

tive because cache and branch predictor conflicts counteract the additional ILP provided by the additional threads. Second, the feasibility of implementing future SMT processors with more than 4 contexts is unclear [8].

Regarding the length of the phases, we use 50,000 cycle warm-up phases, 10,000 cycle actual-sample phases, and 1,200,000 cycle tune phases that are split in sub-phases of 15,000 cycles each. Hence, the time dedicated for running in isolation the HPT represents 5% of the total execution time. Furthermore, the maximum number of reserved ways of the L2 cache for the HPT is 4. This value has been determined empirically.

6. RESULTS

In this section we show the results obtained from our strategy, focusing on two main points. First, we show the average performance obtained for the HPT for each workload type. Second, we show the performance obtained for the LPTs.

6.1 HPT performance

In Figure 5 we show, for the different workloads and different target percentages, the overall percentage of full program speed that we have obtained using our mechanism. On the x-axis, the target percentage of the full speed of the HPT is given, ranging from 10% to 90%. For each size of the workload (2, 3, or 4 threads) the achieved IPC for the High Priority Thread as a percentage of its full IPC is given. We see that over the entire range of different workloads and target percentages, we achieve this target or a little bit more (approximately 3%). Only on the two extreme ends of the range of targets, we are somewhat off target. We discuss the discrepancies for the 10 and 90% cases, since they give most insight in how our mechanism works.

If the target IPC should be 10% of the full speed, we achieve percentages between 13 and 21. To explain this, first consider the II2 workload in which two ILP threads are running. Suppose both threads have a full speed of 4 IPC. Then, in 5% of the time during the sample phase, the HPT reaches this full speed. During the remaining 95% of the time, it reaches 0.4 IPC. Hence, in total it reaches $0.05 \times 4 + 0.95 \times 0.4 = 0.58$ IPC which is 15% of its full speed of 4 IPC. Hence, the sample phase that takes 5% of the total running time of the program causes the resulting total throughput of the HPT to be larger than it should be. From Figure 5 it is also clear that for the workloads II3 and II4 the achieved percentage is closer to 10. This can be explained because, as the number of threads increases, the IPC value of the HPT in the sampling period is lower due to more interference of the other threads. As a result, the overall IPC of the HPT drops a little.

Next, consider the IM workload. In this case, the memory bounded LPTs causes L2 cache pollution, more than is the case for the II workloads. Hence, the measured IPC of the HPT during the sample phase is lower than it should be and during the tune phases the HPT also suffers from interference. Therefore, the effects described for the II case above do not show up as profoundly in the MI case and the overall throughput is closer to 10% as it should be.

For the MI workload, the `mcf` benchmark has a full speed of 0.15 IPC. Hence, 10% of this full speed is only 0.015 IPC. Due to the duration of the sample phase, we reach a slightly higher overall IPC than this. However, the absolute numbers are so small that such a minimal deviation causes a high relative error: we measured a 30% deviation. Hence, the error in the IPC of `mcf` dominates the average results shown in the figure and therefore the large difference is due to this benchmark. Moreover, in general MEM benchmarks have low IPC values and when they are used as HPT, small differences in their IPCs again cause large relative errors.

For the MM workload, the same explanation holds as for the MI workload.

On the other end of the spectrum, when the required percentage is 90, the realized percentage is 2 to 5 percent lower than it should be. To explain these differences, if the LPTs are memory bounded, then they cause much pollution in the L2 cache. Hence, the IPC of the HPT we measure during the sample phases is lower than it should be. Moreover, during the tune phase, memory bounded LPTs cause much interference in the L2 cache also. Therefore, the relative IPC of the IM workloads is lower than the IPC of the II workloads. Therefore, during the tune phase we achieve an IPC value that is too low also. However, in case the LPTs are ILP, this pollution is much less and therefore, achieved IPC values are higher than for the previous case. We can conclude that when the required percentage is 90 it can be more preferable to run the HPT in isolation and reach 100% of its full speed.

These observations show that it can be profitable, especially for the extreme ends of the spectrum, to use a dynamic adjustment of the duration of the sample and tune phases based on the required target percentages and the characteristics of the workloads. We are currently working on this issue. In more common situations where target percentages range from 30 to 80, we already achieve these percentages almost exactly, being less than 1% over target on average and hence in these case no dynamic adjustment is needed.

6.2 LPTs performance

In order to discuss the performance of the Low Priority Threads, we compare their throughput to the throughput the workload obtains under the *icount* fetch policy that we consider to be the base case. Consider Table 4 where we give a schematic picture of the throughput under both policies. Under *icount*, we reach a total throughput of $x + y$. Under *QoS*, the throughput of the HPT, x', is enforced by the OS. Hence, we want to maximize y'. That is, we like the fraction $\frac{x'+y'}{x+y}$ to be as large as possible. If it is 1, or 100%, we achieve the same throughput. This fraction is shown in Figure 6. Secondly, we show the fraction $\frac{y'}{y}$ in Figure 7 to indicate how the LPTs on their own fare in our policy.

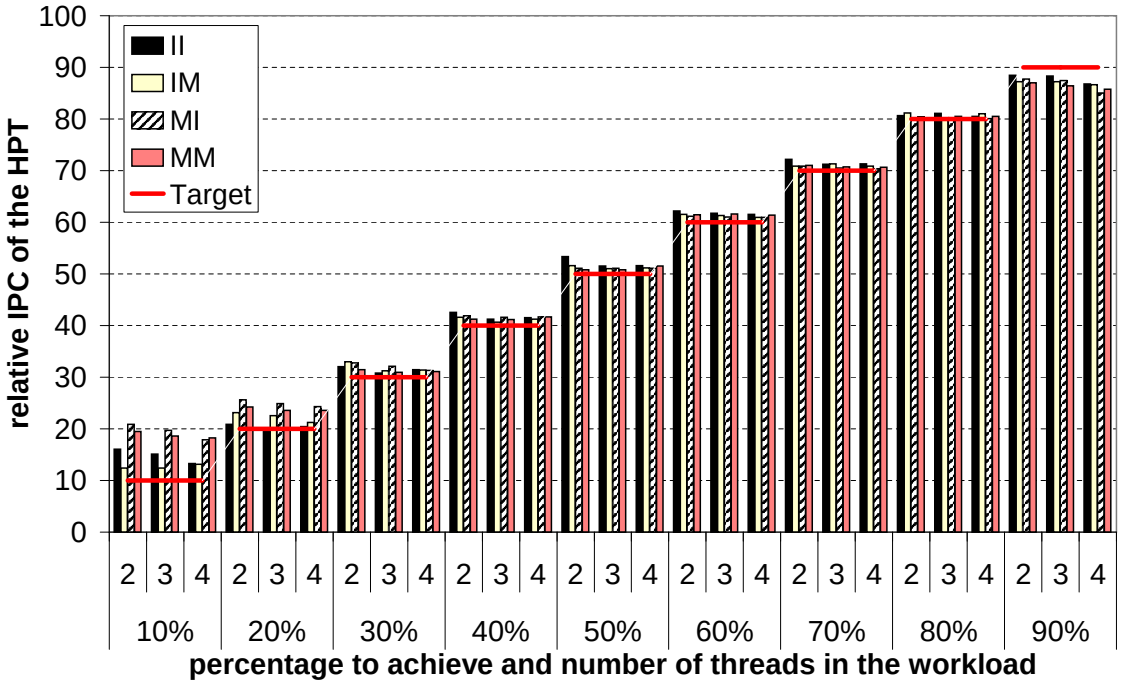

Figure 5: Realized IPC values for the HPT. The x-axis shows the target percentage of full speed of the HPT and size of the workloads. The four different bars represent the four different types of workload discussed in Section 5

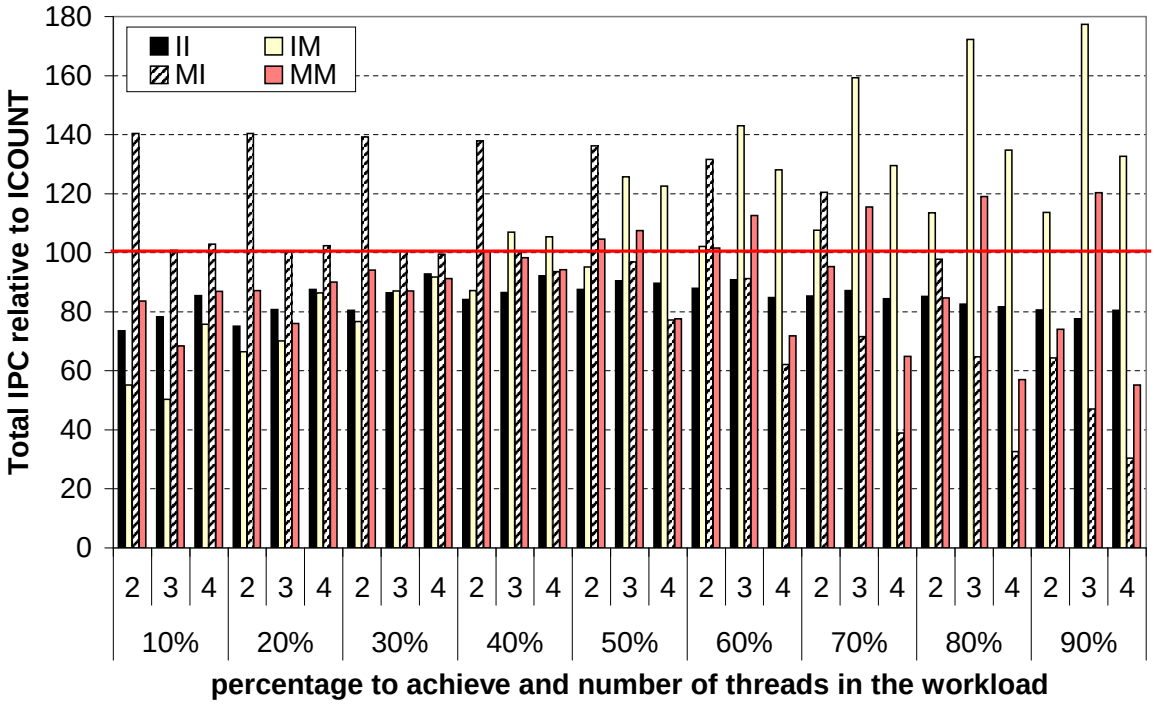

Figure 6: Total IPC for each workload relative to the total IPC of that workload under *icount*.The x-axis shows the target percentage of full speed of the HPT and size of the workloads. The four different bars represent the four different types of workload discussed in Section 5

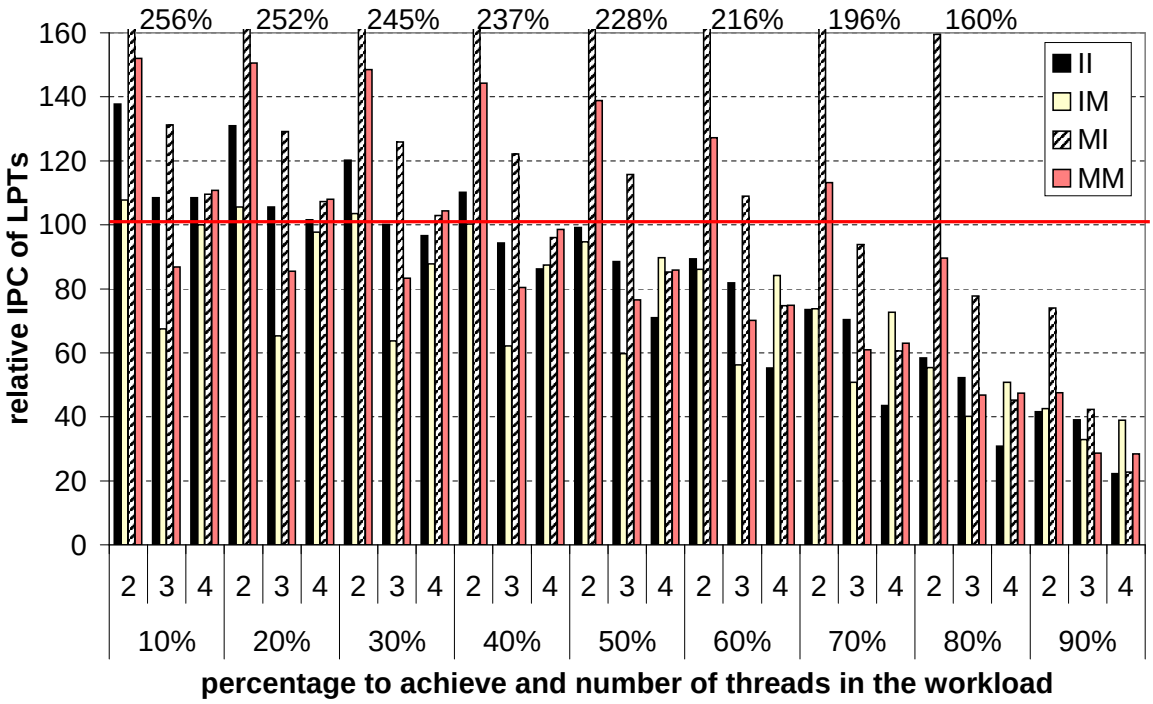

Figure 7: Relative IPC for LPTs with respect to the total IPC of the LPTs when the entire workload runs using *icount*. The x-axis shows the target percentage of full speed of the HPT and size of the workloads. The four different bars represent the four different types of workload discussed in Section 5

	icount	*QoS*
HPT	x	x'
LPs	y	y'
total	$x+y$	$x'+y'$

Table 4: Schematic view of IPC values for *icount* and *QoS*

Distinguishing the different types of the workloads, we can make the following observations. First, if the workload is of type II, the total throughput delivered by *icount* is always larger than the throughput delivered by our QoS mechanism as can be seen in Figure 6. This is to be expected because *icount* is geared toward throughput maximization of ILP workloads. There are few cache misses and instructions from each thread stay in the pipeline only a short time. If we disrupt this behavior by reserving L2 cache lines for one particular thread and dedicate the entire machine to it during the sample phases, it is to be expected that the low priority threads suffer. This loss is higher for 2-thread workloads and when the HPT is executed at extreme (10% or 90%) target percentages because in these cases many resources are given to an ILP thread that, in fact, does not use all of them. Nevertheless, on average we only lose less than 15% in throughput while at the same time being able to provide QoS service.

Second, if the workload is of type IM, if we need to realize a high percentage of the full speed of the high priority ILP thread, then the low priority MEM threads suffer because of lack of resources. Nevertheless, due to the high throughput of the HPT, the total throughput of the entire workload is larger than for *icount*. For a 3 thread IM workload, this improvement can be as much as 78% for the 90% case. The key observation is that under *icount*, the MEM threads tend to occupy resources for a long time when they experience L2 misses causing the ILP thread to suffer. This effect is greatly diminished when we impose a high target percentage for the ILP thread. Also, because the MEM threads have a low IPC value of their own, if they reach only a fraction of this value, the absolute value of the total IPC is not affected strongly. On the other hand, if we achieve a low percentage of the full speed of the high priority ILP thread, we give many resources to the low priority MEM threads. This causes these MEM threads to reach a high relative speed (Figure 7) but since their throughput is low, the total throughput is less than it is for *icount* since the ILP thread is running slowly.

Third, if the workload is of type MI, effects opposite to the IM case occur. If we require a low percentage of the full speed of the high priority MEM thread, there are many resources available to the low priority ILP threads. Hence throughput is as good or better than it is under *icount* since the low priority LPT can have more access to shared resources than under *icount*. The relative speed of the LPTs is much better than it is under *icount* (see Figure 7). Since part of the resources are reserved for the LPTs, the harmful effect that a MEM thread occupies many resources for a long time due to L2 misses, is reduced and even for high target percentages the relative speed of the low priority ILP threads is over 100%. This can be seen in particular for the MI2 case, where total IPC is 100% or more of *icount* even

for target percentages of 70 and 80.

Fourth, if the workload is of type MM, the results are complementary to the II case. Resources are used in much the same way by both HPT and LPTs. Hence, if we assign more resources to the HPT, the LPTs suffer to about the same degree as the LPT benefits. The net result is that total IPC remains fairly close to the total IPC obtained using *icount*. The differences that we observe in Figure 6 are mostly due to the behavior of the `mcf` benchmark.

Summarizing, we conclude that our QoS mechanism is capable of realizing a target IPC for a particular High Priority Thread within an error margin of less than 1% for realistic situations. At the same time, it maximizes throughput for the remaining Low Priority Threads, achieving relative IPC of over 80% for these threads compared to their speed under *icount* and a total throughput that is 94% of the total throughput when using *icount* and for target percentages ranging from 30 to 80% we even reach 97.5% of the throughput of *icount*.

7. RELATED WORK

In [22] it is observed that the total throughput of an SMT processor is highly sensitive to the instruction fetch policy. Other researchers have suggested policies to improve the usage of SMT resources in cases where a thread is stalled as a result of a cache miss or a mispredicted branch [4][7][15][19][21]. All these fetch policies try to optimize total IPC and/or reduce energy consumption. They do not allow control of how threads are executed to meet particular requirements, in contrast to the approach adopted in this paper.

To the best of our knowledge, there does not exist much work on real time constraints for SMT architectures. Jain *et al.* [12] study soft real time scheduling for SMT, but they look at how specific workloads can be assembled from a pool of tasks that is larger than the number of available contexts. Therefore, they address the so-called *co-scheduling* problem for SMT processors. In contrast, this paper concentrates on how internal resources of the processor should be allocated to a given workload in order to guarantee a certain required performance for a High Priority Thread, the so-called *resource sharing* problem. Dorai and Yeung [6] propose transparent threads, which is a mechanism that allows background threads to use resources that a foreground thread does not require for running at almost full speed. Their proposal does not allow the foreground thread to run at a given percentage of its full speed as is the case in our proposal. In a certain sense, this work addresses the problem of job prioritization from the opposite side as we do: whereas we propose mechanisms to assign resources to the High Priority Thread in order to meet constraints, they propose mechanisms to utilize resources by background threads that are left over by the foreground thread. Since they only solve the problem of running a foreground thread at its full speed, their approach is much less flexible than ours. In [20], Snavely et al. propose several OS level job schedulers to enforce priorities. Mostly, these schedulers find co-schedules from a pool of runnable jobs that is larger than the number of hardware contexts. Their *SOS* policy runs jobs alone on the machine to determine their full speed, runs several job mixes in order to determine the best mix that exhibits symbiosis, and finally runs jobs alone in order to meet priorities. This approach obviously under-utilizes the machine resources since in many time frames jobs are running alone, in contrast to our approach in which the mix almost always runs together. Next, they propose an extension to the *icount* fetch policy by including handicap numbers that reflect the priorities of the jobs. This approach suffers from the same shortcomings as the standard *icount* policy, namely, that resource management is implicitly done by the fetch policy. Therefore, running times of jobs are still hard to predict, rendering this approach unsuited for real time constraints. For example, high priority memory bounded threads will clog the pipeline after L2 misses, even more than is the case in standard *icount*. Their approach can be considered as a way of interacting between one goal and one policy. Instead, the view adopted in this paper is that the OS-processor interaction would need to support different policies as well as different goals, which should be satisfiable by the policies.

8. CONCLUSION

In this paper, a mechanism for SMT processors that enables us to satisfy a QoS criterion has been proposed for the first time. This mechanism consists of designating a High Priority Thread in a workload and the requirement that this thread runs at a given percentage of throughput that it would obtain when run in isolation on the machine. In turn, this mechanism enables SMT processors for use in a real time environment. The mechanism basically consists of two recurring phases: a sample phase in which the local IPC of the HPT is determined, and a tune phase in which the actual IPC of the HPT in the context of a number of Low Priority Threads is measured and shared resources are dynamically allocated to the HPT in order to obtain the given target percentage for the HPT.

We have shown that we indeed can realize target percentages ranging from 10 to 90% for HPTs that are high ILP as well as memory bounded, in contexts of 1, 2, or 3 other threads that can be high ILP as well as memory bounded also. This wide range of behavior of studied workloads as well as the wide range of target percentages shows that our proposal is extremely robust and likely to perform well in a wide spectrum of cases. At the same time, we have shown that the performance of the LPTs suffers minimally. This ensures that the primary purpose of using SMTs, namely, very fine grained resource sharing in order to maximize throughput for collections of applications, is maintained when our QoS mechanism is added: we reach on average 94% of the throughput of *icount* and even 97.5% on average for the range from 30 to 80% of required full speed. This means that our QoS mechanism enables SMTs to meet real time constraints while, at the same time, it continues to execute many applications concurrently, thereby maximizing exploitation of all available resources.

Acknowledgments

This work was supported by an Intel fellowship, by the EC IST programme (contract HPRI-CT-2001-00135), and by the Ministry of Science and Technology of Spain under contract TIC-2001-0995-C02-01, and under grant FP-2001-2653 (Francisco J. Cazorla). The authors would like to thank Oliverio J. Santana, Fernando Latorre and Ayose Falcón for their comments and work in the simulation tool. The authors also would like to the reviewers for their valuable comments.

9. REFERENCES

[1] D. Alpert. Will microprocessors become simpler? *Microprocessor Report*, Nov. 2003.

[2] J. Burns and J.-L. Gaudiot. Quantifying the SMT layout overhead- does SMT pull its weight? *Proceedings of the 6th Intl. Conference on High Performance Computer Architecture*, pages 109–120, Jan. 2000.

[3] J. Burns and J.-L. Gaudiot. SMT layout overhead and scalability. *IEEE Transactions on Parallel and Distributed Systems*, 13(1):142–155, Feb. 2002.

[4] F. J. Cazorla, E. Fernandez, A. Ramirez, and M. Valero. Improving memory latency aware fetch policies for SMT processors. *Proceedings of the 5th International Symposium on High Performance Computing*, Oct. 2003.

[5] D. Chiou, P. Jain, S. Devadas, and L. Rudolph. Dynamic cache partitioning via columnization. *Proceedings of Design Automation Conference*, June 2000.

[6] G. K. Dorai and D. Yeung. Transparent threads: Resource sharing in smt processors for high single-thread performance. *Proceedings of the 11th Intl. Conference on Parallel Architectures and Compilation Techniques*, pages 30–41, Sept. 2002.

[7] A. El-Moursy and D. Albonesi. Front-end policies for improved issue efficiency in SMT processors. *Proceedings of the 9th Intl. Conference on High Performance Computer Architecture*, Feb. 2003.

[8] P. N. Glaskowsky. IBM previews Power5. *Microprocessor Report*, Sept. 2003.

[9] M. Gulati and N. Bagherzadeh. Performance study of a multithreaded superscalar microprocessor. *Proceedings of the 2nd Intl. Conference on High Performance Computer Architecture*, pages 291–301, Feb. 1996.

[10] S. Hily and A. Seznec. Contention on 2nd level cache may limit the effectiveness of simultaneous multithreading. Technical Report 1086, IRISA, Feb. 1997.

[11] 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. *Proceedings of the 19th Annual Intl. Symposium on Computer Architecture*, pages 136–145, May 1992.

[12] R. Jain, C. Hughes, and S. Adve. Soft real-time scheduling on simultaneous multithreaded processors. *Proceedings of the 5th International Symposium on Real-Time Systems Symposium*, pages 134–145, Dec. 2002.

[13] R. Kalla, B. Sinharoy, and J. Tendler. SMT implementation in POWER 5. *Hot Chips*, 15, Aug. 2003.

[14] P. Knijnenburg, A. Ramirez, J. Larriba, and M. Valero. Branch classification for SMT fetch gating. *Proceedings of the 6th Workshop on Multithreaded Execution, Architecture, and Compilation*, pages 49–56, 2002.

[15] C. Limousin, J. Sebot, A. Vartanian, and N. Drach-Temam. Improving 3D geometry transformations on a simultaneous multithreaded SIMD processor. *Proceedings of the 15th Intl. Conference on Supercomputing*, pages 236–245, May 2001.

[16] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pages 164–171, Nov. 2001.

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

[18] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. *Proceedings of the 10th Intl. Conference on Parallel Architectures and Compilation Techniques*, Sept. 2001.

[19] R. Shin, S.-W. Lee, and J. L. Gaudiot. Dynamic scheduling issues in smt architectures. *Proceedings of the International Parallel and Distributed Processing Symposium*, Apr. 2003.

[20] A. Snavely, D. Tullsen, and G. Voelker. Symbiotic job scheduling with priorities for a simultaneous multithreaded processor. *Proceedings of the 9th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 234–244, Nov. 2000.

[21] D. Tullsen and J. Brown. Handling long-latency loads in a simultaneous multithreaded processor. *Proceedings of the 34th Annual ACM/IEEE Intl. Symposium on Microarchitecture*, Dec. 2001.

[22] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. *Proceedings of the 23th Annual Intl. Symposium on Computer Architecture*, pages 191–202, Apr. 1996.

[23] D. Tullsen, S. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. *Proceedings of the 22th Annual Intl. Symposium on Computer Architecture*, 1995.

[24] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling. *Proceedings of the 30th Annual Intl. Symposium on Computer Architecture*, pages 84–97, June 2003.

[25] W. Yamamoto and M. Nemirovsky. Increasing superscalar performance through multistreaming. *Proceedings of the 4th Intl. Conference on Parallel Architectures and Compilation Techniques*, pages 49–58, June 1995.