# Predictable Performance in SMT Processors: Synergy between the OS and SMTs

Francisco J. Cazorla, Peter M.W. Knijnenburg, Rizos Sakellariou,
Enrique Fernández, Alex Ramirez, and Mateo Valero, *Fellow*, *IEEE*

**Abstract**—Current Operating Systems (OS) perceive the different contexts of Simultaneous Multithreaded (SMT) processors as multiple independent processing units, although, in reality, threads executed in these units compete for the same hardware resources. Furthermore, hardware resources are assigned to threads implicitly as determined by the SMT instruction fetch (Ifetch) policy, without the control of the OS. Both factors cause a lack of control over how individual threads are executed, which can frustrate the work of the job scheduler. This presents a problem for general purpose systems, where the OS job scheduler cannot enforce priorities, and also for embedded systems, where it would be difficult to guarantee worst-case execution times. In this paper, we propose a novel strategy that enables a two-way interaction between the OS and the SMT processor and allows the OS to run jobs at a certain percentage of their maximum speed, regardless of the workload in which these jobs are executed. In contrast to previous approaches, our approach enables the OS to run time-critical jobs without dedicating all internal resources to them so that non-time-critical jobs can make significant progress as well and without significantly compromising overall throughput. In fact, our mechanism, in addition to fulfilling OS requirements, achieves 90 precent of the throughput of one of the best currently known fetch policies for SMTs.

**Index Terms**—Multithreaded processors, simultaneous multithreading, ILP, thread-level parallelism, performance predictability, real time, operating systems.

## 1 Introduction

Current processors take advantage of Instruction Level Parallelism (ILP) to execute several instructions from a single stream in parallel. However, there is only a limited amount of parallelism available in each thread, mainly due to data and control dependences. As a result, hardware resources added to exploit this limited amount of ILP may be utilized only occasionally, thus significantly degrading the performance/cost ratio of these processors. A solution to overcome this problem is to share hardware resources among different threads. There exist different approaches to resource sharing, ranging from multiprocessors (MPs) to high performance SMTs. In the former case, mostly only the higher levels of the cache hierarchy are shared. Examples of such processors include the Power4 [4]. In the case of SMTs [18], [23], like the Pentium4 [17] or the Power5 [12], many more hardware resources (instruction queue (IQ) entries, physical registers, etc.) are shared among threads. Thus, SMTs have a better cost/performance trade-off [15]. Current trends in processor architecture indicate that many future microprocessors will have some form of SMT. This includes both general-purpose processors, like the Power5, which combines two 2-context-SMT processors on a single chip, and embedded processors, like the Meta [15].

The current collaboration between the OS and the SMT is inherited from the traditional collaboration between the OS and MPs: The OS perceives the different contexts of an SMT as multiple, independent *virtual processors*. As a result, the OS schedules threads onto what it regards as independent processing units operating. However, in an SMT, threads share many resources. Hence, these virtual processors are not truly independent as threads scheduled at any given time compete with each other for the resources of a single processor. The way that these resources are allocated at the microarchitectural level clearly affects their performance.

On the other hand, current SMTs are designed with the main objective of increased throughput, lacking flexibility in providing other objectives. An Ifetch policy 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 maximize throughput [22] and/or fairness [16], possibly by stalling or flushing threads that experience L2 misses [5], [6], [7], [21], or reducing the effects of misspeculation by stalling on hard-to-predict branches [13].

Although objectives such as throughput or fairness might be acceptable in many systems, one can easily imagine situations where the Operating System (OS) may

- *F.J. Cazorla is with the Barcelona Supercomputing Center, Jordi Girona 29, Edificion Nexus II, Despacho 112, 08034 Barcelona, Spain. E-mail: francisco.cazorla@bsc.es.*
- *P.M.W. Knijnenburg is with the Computer Systems Architecture Group, Informatics Institute, University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands. E-mail: peterk@science.uva.nl.*
- *R. Sakellariou is with the School of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, UK. E-mail: rizos@cs.man.ac.uk.*
- *E. Fernández is with the Institute for Cybernetic Sciences, Universidad de Las Palmas de Gran Canaria, Edificio de Informatica y Matematicas, Campus Universitario de Tafira 35017, Las Palmas de Gran Canaria, Spain. E-mail: efernandez@dis.ulpgc.es.*
- *A. Ramirez and M. Valero are with the Barcelona Supercomputing Center (BSC) and the Computer Architecture Department, Universitat Politecnica de Catalunya, Jordi Girona 1-3, D6 105, 08034 Barcelona, Spain. E-mail: {aramirez, mateo}@ac.upc.edu.*

*Manuscript received 13 July 2004; revised 23 July 2005; accepted 7 Dec. 2005; published online 22 May 2006.*
*For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0235-0704.*

need to impose additional objectives. These may relate only to some of the threads and may get priority over generic objectives such as fairness and/or throughput. For instance, in order to meet some (soft) real-time requirements, the OS may require that a designated thread run at a certain percentage of the maximum speed it would get if it had the machine alone. In a conventional uniprocessor system, an easy way to achieve such a requirement would be to guarantee that, over a period of time, the designated thread would be allocated an amount of CPU time, which is equal to the percentage of speed requested. However, this approach would not be useful in an SMT. When the designated thread is running, it will typically be sharing processor resources with other threads (if it was running on its own, it would clearly underuse the SMT and defy the whole point about having SMTs). As a result, the designated thread's performance requirement cannot be guaranteed because the fetch policy used for resource sharing would assign resources in order to meet its own objectives rather than those of the OS. If it would serve its purpose to allocate fewer resources to the designated thread, it would do so.

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. In other words, the hardware should provide the OS with some kind of *Quality of Service* (QoS) 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, both the traditional OS-level job scheduler and current SMT approaches to resource sharing by means of Ifetch policies are no longer adequate and a closer interaction is required.

In this paper, we present novel architectural support for OSs on SMT processors that enables a close interaction between both. This is used to guarantee that jobs in a workload can achieve a certain performance requirement. As opposed to traditional monolithic approaches, where the OS has no control over the resource sharing of an SMT architecture, our approach is based on a resource sharing policy that is continuously adapted as it needs to be to take into account OS requirements. The advantages of the proposed approach are twofold.

First, the OS can control the execution of jobs at any given time. The OS selects a workload consisting of several threads and indicates to the processor that certain threads should be considered as *Predictable Performance Threads* (PPTs) and must execute at a certain percentage of their full speed, that is, the speed that a thread can achieve when it is run alone on the processor. We have experimented with up to two PPTs using our mechanism. When one thread is required by a given IPC, we obtain successful results, with an error lower than 2 percent, for target percentages ranging from 10 percent to 80 percent. When there are two PPTs, we run both jobs exactly at the required percentage for a broad range of percentages. For another broad range of required percentages, our mechanism can realize the target for the thread with the highest priority and give a close approximation of the required speed for the thread with the second highest priority. This degradation only arises when there are insufficient number of resources inside the processor to meet both requirements at the same time.

Second, we show that the remaining *Unpredictable Performance Threads* (UPTs) can make good use of the resources that are not needed by the PPTs. As a result, our prioritization mechanism not only maintains total throughput, but is in fact capable of outperforming traditional fetch mechanisms in this respect. Thus, a resource conscious scheduler, by using our resource allocation mechanism, can perform better than a traditional OS job scheduler using a fetch policy as a single solution for all cases. In fact, our mechanism achieves at least 90 percent of the performance of one of the best currently known fetch policies for SMTs, like FLUSH++ [5].

This paper is structured as follows: Section 2 describes related work and our novel approach to an OS-SMT collaboration is placed in context. Section 3 defines the problem we solve in this paper as well as our approach to solve it. Section 4 elaborates the mechanism used to achieve this collaboration. Section 5 shows the experimental setup used for evaluation. Section 6 shows the results obtained when prioritizing jobs using our mechanism. Finally, we draw some conclusions in Section 7.

## 2 Background and Problem Statement

As illustrated in the introduction, the approaches currently employed for resource sharing in SMTs, I-fetch policies, may disregard any target set by the OS. This is a consequence of the additional level of scheduling decisions, introduced by resource sharing inside the SMT, which has been largely examined independently of the more traditional OS level *coscheduling* phase.

In [3], the authors conclude that current fetch policies that attempt to maximize throughput, like *icount*, should be "balanced" for minimum forward progress of real-time tasks; otherwise, real-time tasks may miss their deadline. In Fig. 1a, we show an example of how the performance of an application varies depending on the workload it is executed in when it is run in an SMT. Fig. 1a 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]. We can see that its IPC varies a lot, depending on the fetch policy as well as the nature of the other threads running in the context. This is caused by the fact that management of resources (IQ entries, registers) is not explicit. This shows that the current collaboration between the OS and the SMT hardware, in which the OS has no control over the resource sharing of an SMT architecture, does not provide control over the execution of applications.

To the best of our knowledge, only a few papers have identified the need for a closer interaction between coscheduling algorithms and the resource sharing made in SMT processors in order to force priorities. In [20], an extension to the *icount* fetch policy is proposed by including handicap numbers that reflect the priorities of 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, although this mechanism is able to prioritize threads to some extent,

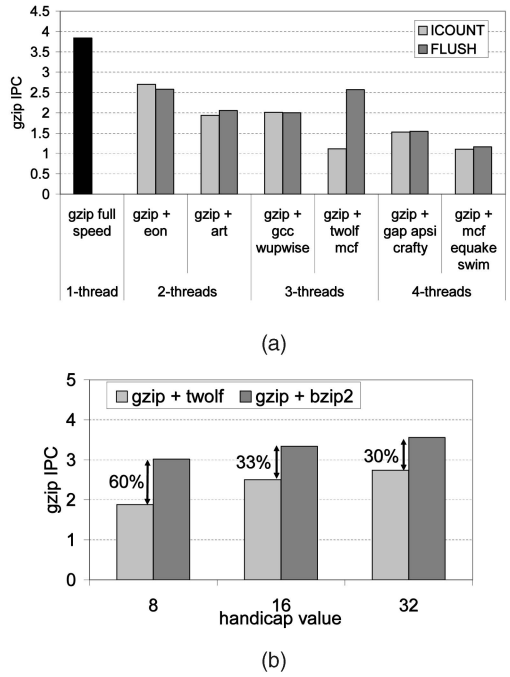


Fig. 1. Motivation of this paper. From top to bottom: (a) gzip IPC for different contexts. (b) gzip IPC for different handicap values.

the running times of jobs are still hard to predict, rendering this approach unsuited for real-time constraints. For example, in Fig. 1b, we show the IPC of the `gzip` benchmark for different handicap numbers and workloads. We observe that running `gzip` with the same handicap leads to different IPC values, with a variation of up to 60 percent, depending on the workload. Hence, even when the *icount* handicap prioritization mechanism achieves some prioritization of threads, it is still far from the objective of this paper, namely, ensuring that several threads in a workload run at a given target IPC.

Finally, the Power5 [12] processor uses a mechanism to control the decode bandwidth. In our architecture, this control provides the same results as controlling the fetch bandwidth. However, given that no information on the internal resource allocation of the Power5 has been released, we cannot compare the efficiency of this mechanism and ours.

## 3 QoS Problem Definition

In this paper, we move a step further than existing work and we propose a novel approach for a *dynamic* interaction between the OS and the SMT which allows the former to pass specific requests onto the latter. In particular, we focus on the following challenge: Given a workload of $N$ jobs[1] and one or two Predictable Performance Threads (PPT0 and PPT1) in this workload, where PPT0 has higher priority than PPT1, find a resource sharing policy to:

- Ensure that PPT0 runs at (at least) a given target IPC that represents $X$ percent of the IPC it would get if it were to be executed alone on the machine.

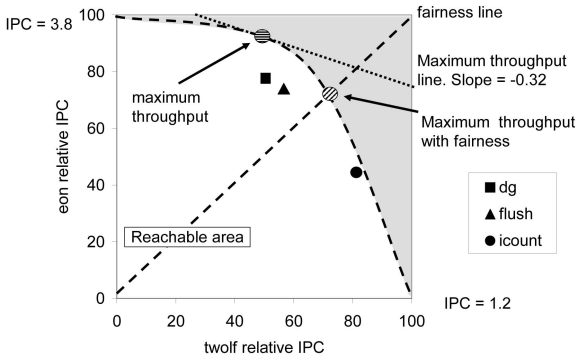


Fig. 2. QoS space for three fetch policies and important QoS points.

- Second, ensure that PPT1 (if PPT1 is given) runs at a given target IPC that represents $Y$ percent of the IPC it would get if it were to be executed alone on the machine or as close as possible to this $Y$ percent when there are insufficient resources to realize the targets for both PPT0 and PPT1 at the same time.
- Third, maximize the throughput for the remaining $N-1$ (or $N-2$ if a PPT1 is given) Unpredictable Performance Threads (UPTs) in the workload.

To tackle a challenge such as the one described above, we propose a generic approach to resource sharing for SMTs which addresses such challenges as a QoS problem. This approach is inspired by QoS in networks in which processes are given guarantees about bandwidth, throughput, or other services. Analogously, in an SMT, resources can be reserved for threads in order to guarantee a required performance. Our view is that this can be achieved by having the SMT processor provide "levers" through which the OS can finetune the internal operation of the processor as needed. Such levers can include prioritizing instruction fetch for particular threads, reserving parts of the resources like IQ entries, etc.

In order to measure the effectiveness of a solution to a QoS problem, we have proposed the notion of *QoS space*. We observe that, on an SMT processor, each thread, when running as part of a workload, reaches a certain percentage of the speed it would achieve when running alone on the machine. Hence, for a given workload consisting of $N$ applications and a given instruction fetch policy, these percentages give rise to a point in an $N$-dimensional space, called the *QoS space*.[2] For example, Fig. 2 shows the QoS space for two threads, `eon` and `twolf`. In this figure, both the $x$ and $y$-axis span from 0 to 100 percent. We have used three fetch policies: *icount* [22], *flush* [21], and data gating (*dg*) [9] in our baseline configuration. Theoretically, if a policy leads to the point $(x, y)$, then it is possible to reach any point in the rectangle $(0,0),(x,y)$ by judiciously inserting empty fetch cycles. Fig. 2 shows a more general picture in which the dashed curve indicates points that intuitively could be reached using some fetch policy. Obviously, by assigning all fetch slots and resources to one thread, we reach 100 percent of its full speed. Conversely, it is impossible to reach 100 percent of the

1. We assume throughout the paper that the workload is smaller than or equal to the number of hardware contexts supported by the SMT.

2. As shown in Section 6, the notion of QoS space is applicable to other quantities, not only a percentage of IPC.

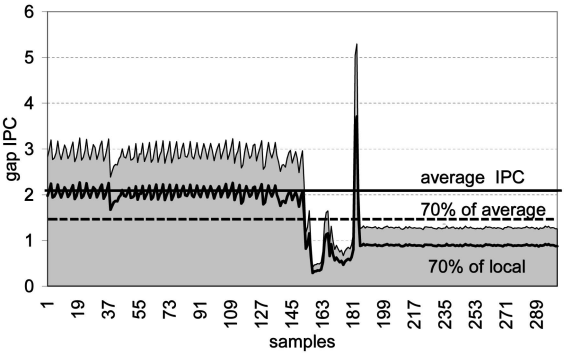


Fig. 3. Local IPC of the gap benchmark.

speed of each application at the same time since they share resources.

In Fig. 2, we see that the representation of the QoS space also provides an easy way to visualize other metrics used in the literature. Points of equal throughput lie on a single line whose slope is determined by the ratio of the maximum IPCs of each thread (in this case, $-1.2/3.8 = -0.32$). Such a point with maximum throughput is also indicated in the figure. Finally, points near the bottom-left top-right diagonal indicate fairness, in the sense that each thread achieves the same proportion of its maximum IPC. In either case, maximum values lie on those lines that have a maximum distance from the origin.

Each point or area (set of points) in the reachable subspace entails a number of properties of the execution of the applications: maximum throughput, fairness, real-time constraints, power requirements, a guarantee, say 70 percent, of the maximum IPC for a given thread, any combination of the above, etc. In other words, each point or area in the space represents a solution to a *QoS requirement*. It is the responsibility of the OS to select a workload and set a QoS requirement and it is the responsibility of the processor to provide ways to enable the OS to enforce such requirements. The processor should dynamically adjust the resource allocation and attempt to converge to a point or area in the QoS space that corresponds to a QoS requirement.

## 4 Solution of the QoS Problem

The notion of QoS implies the existence of hardware mechanisms in the SMT processor that can offer effective control over the execution of threads. It also implies that the OS should have some knowledge of the full speed of jobs in order to be able to give requirements to the hardware, such as "execute this job at a certain percentage of its full speed." That is, our mechanism requires that all instances of an application have more or less the same IPC. This assumption does not hold in general as the IPC of applications can change depending on the input data. Fortunately, it has been shown [11] that the IPC of media applications is roughly the same between frames. Hence, our proposal can be applied to emerging multimedia applications.

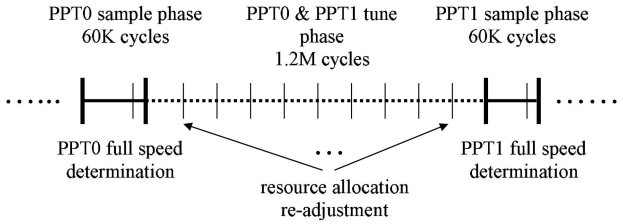


Fig. 4. The Sample and Tune phases used in our mechanism.

### 4.1 Overview/Concept

In the challenge considered in this paper, we want to guarantee a certain speed for the PPTs. We obtain this by reserving the proper resources for them, proceeding hierarchically. First, we reserve resources for PPT0 as it has the highest priority. Then, we reserve resources for the subsequent lower in priority, PPT1, from those resources not used by PPT0, and so on. Finally, the remaining resources are given to the UPTs.

The QoS mechanism we propose combines three key ideas. First, we monitor the characteristics and execution progress of each thread and give feedback to the OS. Second, we employ resource administration and allocation, guided by the demands of threads and the external objective given by the OS. Third, we shield the PPTs against the destructive interference of the other threads.

Another 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 varying IPC. We illustrate this by an example. Fig. 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 70 percent 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 70 percent of the global IPC, even if the thread were given all the processor resources. Moreover, if the processor achieves this 70 percent 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 70 percent of the global IPC.

The basis of our mechanism for dynamic resource allocation rests on the observation that in order to realize $X$ percent of the overall IPC for a given job, it is sufficient to realize $X$ percent of the maximum possible IPC *at every instant* throughout the execution of that job. This is illustrated in Fig. 3 by the boldfaced curve labeled "80% of local IPC." Hence, the mechanism needs to determine the variations in IPC of the PPT. In order to do this, we distinguish two phases in our proposed mechanism that are executed in alternate fashion, as shown in Fig. 4.

- *Sample phase*: During this first phase (60,000 cycles), all shared resources are given to a PPT and the other threads are temporarily stopped. As a result, we

obtain an estimate of the current full speed of that PPT during this phase which we call the *local sampled IPC*.

- *Tune phase*: During this phase (1.2M cycles), UPTs are allowed to run. Our mechanism dynamically varies the amount of resources given to both PPTs to achieve the *local target IPC*. It is given by the local sampled IPC computed in the last sample period times the required percentage given by the OS, which we call *target percentage*.

Clearly, if we are able in the sample phase to measure, reasonably accurately, the full speed of a given PPT and in the tune phase to realize a percentage $X$ percent of that sampled IPC, then we obtain an overall IPC that is about $X$ percent of the IPC of that PPT would have when executed alone in the processor. In the next two subsections, we discuss the sample phase and the tune phase in more detail.

### 4.2 Sample Phase: Determining the Local IPC of a PPT

During the sample phase, we determine the local IPC of a PPT by giving it all shared resources and, hence, suspending the others 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 Unpredictable Performance Threads. Hence, we have to determine the local IPC of the PPT thread as fast as possible. Our mechanism dedicates only 5 percent of the total execution time to the sample phase.

#### 4.2.1 Shared Resources

In our simulated architecture, as given in Section 5, there are the following shared resources: fetch and issue slots, IQ entries, physical registers, caches, TLBs, and the branch predictor. Our mechanism takes into account the following: L2 cache, IQs, physical registers, and the fetch and issue bandwidth. The rest are *freely* shared among all threads. These resources are allocated in the following way:

First, the allocation of the IQs and the physical registers is as follows: Depending on the particular needs of the PPTs during their execution, the number of resources allocated to them is changed. When a shared resource is partitioned, one part is dedicated to PPT0, another part to PPT1, and the remaining part is dedicated to the UPTs.

Second, fetch and issue bandwidth is shared hierarchically. The PPT0 has priority to use it and, when PPT0 cannot use the entire bandwidth, it is given to the PPT1. The remaining bandwidth is used by UPTs, breaking ties with *icount*.

Third, unlike previous resources, caches, TLBs, and the branch predictor can suffer destructive interference because an entry given to one thread can be evicted by another thread. In order to get more insight into this interference, we show in Fig. 5 how many interthread conflicts the PPT suffers during a 100,000 cycle-long sample phase, averaged over the entire run of a workload consisting of `twolf` as PPT and `mcf`, `equake`, and `swim` as UPTs. We observe that, as the sample phase progresses, the number of conflicts goes toward 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

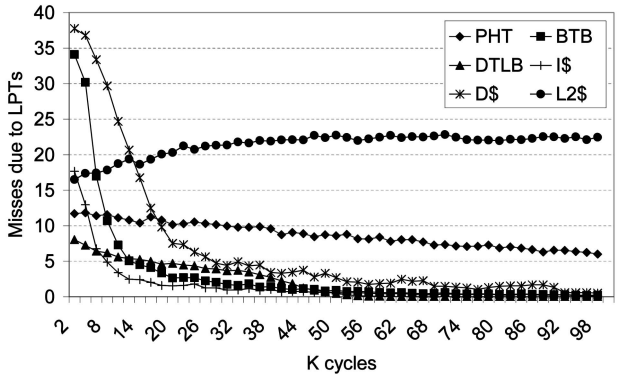


Fig. 5. Misses suffered by the PPT due to the UPTs interference.

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 interthread misses have disappeared. However, we have also measured that this interference is mostly neutral, giving a small loss in the branch predictor hit rate of less than 1 percent. Hence, we ignore the interference in the branch predictor. 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 percent for some benchmarks). This high number of cycles shows that we cannot deal with the interference in the L2 by simply extending the warmup phase. We address this problem below.

The solution we propose to erase the conflicts in all shared resources, except the L2 cache, consists of splitting each sample period into two subphases. During the first subphase, the *warmup phase*, that consists of 50,000 cycles, the PPT is given all resources, but its IPC is not yet determined. In the second subphase, the *actual-sample phase*, the PPT keeps all resources and, moreover, its IPC is determined. The duration of this subphase is 10,000 cycles and it was determined empirically and is in accordance with the results published in [24]. In this study, the authors present a statistical study showing how to sample the trace of a given thread in order to obtain an accurate measure of its IPC.

#### 4.2.2 Reducing L2 Cache Misses Due to Interthread Interference

The L2 cache is an important source of unpredictability because the effects of threads interferences remain for a long time and because they affect the critical path processor-memory.

Our solution to interthread interference in the L2 cache consists of *partitioning* this cache into several parts. One part is dedicated to each PPT and the other part can be used by the entire workload, UPTs and PPTs. 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 each PPT. The cache behaves like a "normal" cache, except in the placement and replacement of

data. The PPT0 is allowed to use the entire L2, the PPT1 is allowed to use the entire cache except the ways reserved for the PPT0. Finally, the UPTs are restricted to using the remainding 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 a PPT always have a higher access count than the lines in the shared part of the cache. The bit of the PPT0 is more significant than the bit of the PPT1.

Let us assume that a load/store instruction misses in the L2 cache and causes a replacement/eviction. If that instruction was issued by a UPT, then only lines belonging to the shared part of the cache are selected for replacement using the LRU algorithm. If it was issued by the PPT1, then we mask this extra bit and we first select a victim line that belongs to an UPT, if possible. If there does not exist such a line, the LRU line from the lines belonging to the PPT1 and in the shared part is selected as the victim. If that instruction was issued by the PPT0, then we first select a victim that belongs to a UPT. Next, to the PPT1 and, finally, if all data in the set belong to the PPT0, the LRU line of the entire set is selected as victim.

Note that this extension allows the cache to be used normally when the SMT does not execute a workload with a designated PPT: The extra bit is always masked. In [8], a different cache partitioning technique, called column caching, has been proposed. However, this technique addresses a much more general problem of cache partitioning. Therefore, that technique is too heavily weighted to be used for our purposes where the simple mechanism described below suffices.

We propose an iterative method that dynamically varies the number of ways reserved for the PPT. The control is local to the cache and only receives from the processor information about how many, if any, PPTs there are. It also receives information about which phase, sample, or tune these PPTs are currently in.

During each actual sample phase, every time a PPT suffers an intrathread miss, a per-thread-counter is incremented. We consider that a thread has experienced an intrathread miss when it is going to use a cache position used by other thread. At the end of the sampling period, if the value of the counter is higher than a threshold[3] of 8, then the number of ways reserved for the PPT is increased by 1. If, on the contrary, the counter is lower than the threshold, this number is decreased by 1.

In this way, if PPTs experience few L2 misses due to interference, we reduce the number of ways reserved for them. Likewise, if they experience many misses, then we increase the number of reserved ways. The maximum number of ways that each PPT can reserve is an empirically derived value that depends on the configuration and on the relative order of priorities. Of course, the higher the number of ways in the L2 cache, the better this algorithm works. This is not a limiting factor, as current trends in computer architecture show that the number of ways in the outer cache level of processors is increasing. For example, the Sparc64 VI [14] has a 12-way L2 cache. The L2 of the Power5 [12] is 10-way. The AMD K8 has a 16-way L2 cache [1].

In our configuration, we can reserve up to 4 ways for the PPT0 and up to 2 for the PPT1.

### 4.3 Tune Phase: Realizing the Target IPC

After each sample phase of 60,000 cycles, there is a tune phase where we try to achieve the target percentage of the local IPC measured in the previous sample period, that is, the *local target IPC*. As stated in Section 4.1, we want the sample phase to be at most 5 percent of the total execution time. For this reason, each tune phase takes 1.2 million cycles.

We proceed as follows: First, we adjust the partitioning of the L2 cache that remains the same for the entire tune phase. This adjustment has been described in the previous section. Second, the amount of the other shared resources (IQ entries and the physical registers) dedicated to each PPT is dynamically varied as follows: Each tune phase is split into 80 subphases of 15,000 cycles. At the end of every subphase, the average IPC of the PPTs in this subphase is computed. If the IPC of the PPT0 is lower than its local target IPC, then the amount of resources given to it is increased. We proceed similarly with the PPT1. Otherwise, if this IPC is higher than the local target IPC, then the amount of resources given to this PPT is decreased.

We vary the number of instances of each resource dedicated to a PPT by a fixed amount that equals the total number of instances of that resource divided by a *granularity factor*, which has been empirically set to 16. For example, for the 32-entry issue queues, this amount is $32/16 = 2$. Our results show that this factor does not affect the final result much as long as its value is not too low. In that case, our mechanism would need many tuning phases to increase the amount of resources given to the PPTs so that their IPC converge to the targetIPC.

Finally, we have observed that the local sampled IPC values are sometimes lower than they should be due to interference from LPTs, mainly in the L2 cache. This results in local target IPCs that are lower than they should be and, thus, the final IPC obtained for the PPTs is also lower than the target IPC given by the OS. In order to counteract this effect, we also take into account the *global IPC* of the PPT: At the end of each subphase, we check whether the total IPC of the PPT under consideration up to this cycle is lower than the target IPC given by the OS. We introduce a *compensation term* $C$ for this effect, as shown in (1). This term artificially increases the local target IPC so that the final IPC of that PPT converges to the target IPC given by the OS. In this formula, $X$ is the target percentage.

$$local\ target\ IPC = \frac{(X + C)}{100} \times local\ IPC. \qquad (1)$$

Initially, $C$ is zero. If the total IPC is smaller than the target IPC, we increase $C$ by 5. Conversely, if the global IPC is larger than the target IPC, we decrease $C$ by 5. However, we stipulate that $C$ does not become smaller than zero or greater than $100 - X$.

3. The value of this threshold was determined empirically based on the memory latency and the duration of the sample phase. A change of any of these parameters requires an adjustment of this value.

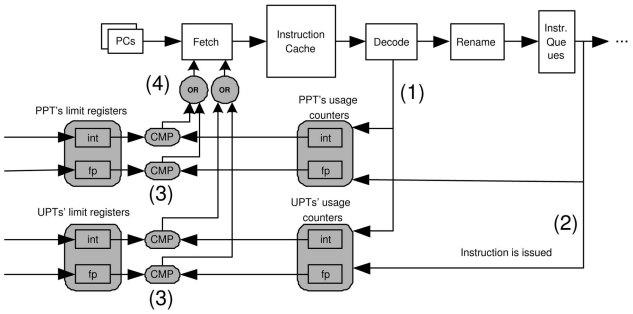


Fig. 6. Hardware required to implement our mechanism.

### 4.4 Hardware Implementation

The implementation of our mechanism involves two main parts: tracking the resources used by each thread and tracking the phase and subphase the mechanism is in. The implementation we propose is valid for any number of PPTs. For simplicity, we explain our implementation when there is one PPT.

#### 4.4.1 Hardware to Track Resource Usage

The objective of this hardware is to track the amount of resources given to each thread and avoid having a thread use more resources than given to it. It has three main blocks of logic.

**Track**: We need a *resource usage counter* for each resource under control, both for each PTT and the UPTs. Each counter tracks the number of slots that each thread has of that resource. Fig. 6 shows the counters required for a 2-context SMT with one PPT if we track the issue queues. Resource usage counters are incremented in the decode stage (indicated by *(1)* in Fig. 6). Issue queue usage counters are decremented when the instruction is issued (2). We also control the occupancy of the physical registers. Hence, two limit resources and two queue usage counters are required, one for each register file (integer and fp). Register usage counters are decremented when instructions are committed.

All extra registers are special purpose registers. This has a twofold implication. First, they do not belong to the register file. The design of the register file is left unchanged with respect to the baseline architecture. Second, these counters are changed implicitly by our mechanism. These counters are not visible to the OS.

The implementation cost of these counters depends on the particular architecture. However, we believe that it is low due to the fact that current processors have tens of performance and event counters, e.g., the Intel Pentium4 has more than 60 performance and event counters [2].

**Compare**: We also need two registers, called *limit registers*, that contain the maximum number of entries that each PPT and the UPTs are entitled to use. In the example shown in Fig. 6, we need six counters: one for each type of issue queue (integer, fp, and load/store) for both each PPT and the UPTs. Every cycle we compare the resource usage counters of each thread with the limit registers (3). If a thread is using more slots than given to it, then a signal is sent to the fetch stage to stall instructions fetch from this thread (4).

**Stall**: If this signal is activated, the fetch mechanism does not fetch instruction from that thread until the number of entries used for this thread decreases. Otherwise, the thread is allowed to compete for the fetch bandwidth as determined by the fetch policy.

#### 4.4.2 Tracking Phases

At any given time, we have to track the phase/subphase our mechanism is in. We use a Finite State Machine (FSM) to do this, as shown in Table 1. This FSM has only six states and is quite simple and can be implemented with four counters and simple control logic.

The FSM starts by establishing the PPT whose full speed we are going to sample in this sample phase, $PPTcurrent$ (S0). All resources are given to that PPT. This is done by simply setting its IQ and physical register limit registers to the maximum number of resources, and resetting the other entries. Next, at the end of the warm-up phase (S2), we begin to compute the IPC of the PPTcurrent. At the end of the actual-sample phase (S4), we compute the local target IPC and set the resource allocation to converge to the local target IPC. We also vary the number of ways reserved for the PPTs. At the end of each tune su-phase (S6), we vary the resource allocation again. After 80 tune subphases, we restart the process by changing the PPT.

## 5 Experimental Setup

In this section, we describe our baseline architecture and the benchmarks we use to evaluate our proposal.

TABLE 1
FSM to Track Phases

| State | Description | Next State 1 | Next State 2 | Actions |
|---|---|---|---|---|
| S0 | Select PPT | S1 | - | (1)PPTcurrent = 1 - PPTcurrent (2) Init all counters (3)Give all resources to PPTcurrent |
| S1 | Warm-up phase | $counter1 < 50k \rightarrow S1$ | $counter1 = 50k \rightarrow S2$ | counter1++ |
| S2 | Actual Sample Start | S3 | - | Start local IPC sampling |
| S3 | Actual Sample phase | $counter2 < 10k \rightarrow S3$ | $counter2 = 10k \rightarrow S4$ | counter2++ |
| S4 | Tune sub-phase Start | S5 | - | (1)Compute local IPC and local target IPC (2)Compute L2 reserved ways for PPTs |
| S5 | Tune sub-phase | $counter3 < 15k \rightarrow S5$ | $counter3 = 15k \rightarrow S6$ | counter3++ |
| S6 | Adjust resources | $counter4 < 80 \rightarrow S5$ | $counter4 = 80 \rightarrow S0$ | (1)counter4++.(2)counter3=0.(3)Adjust resource allocation to achieve local target IPC |

TABLE 2
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 integer, 320 fp |
| ROB size (per thread) | 256 entries |
| Branch Predictor | 16K entries gshare |
| Branch Target Buffer | 256-entry, 4-way associativity |
| Return Address Stack | 256 entries |
| Icache, Dcache | 64 Kbytes, 2-way, 8-bank, 64-byte lines, 1 cycle access |
| L2 cache | 512 Kbytes, 8-way, 8-bank, 64-byte lines, 10 cycle access |
| Main memory latency | 100 cycles |
| TLB miss penalty | 160 cycles |

TABLE 3
L2 Cache Behavior of Benchmarks

| Benchmark type | Benchmark name | L2 cache miss rate |
|---|---|---|
| MEM | mcf | 29.6 |
| | twolf | 2.9 |
| | vpr | 1.9 |
| | parser | 1.0 |
| | art | 18.6 |
| | swim | 11.4 |
| | lucas | 7.47 |
| | equake | 4.72 |
| ILP | gap | 0.7 |
| | vortex,gcc | 0.3 |
| | perl,bzip2,crafty | 0.1 |
| | gzip,mesa | 0.1 |
| | eon,fma3d | 0.0 |
| | apsi,wupwise | 0.9 |

### 5.1 Simulation Tool

In this paper, 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 2 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 eight instructions per cycle from up to two threads.

### 5.2 Benchmarks

The present paper is a "proof of concept" of how controlling SMT internal resource can enable SMTs to execute real-time applications. Our purpose is to show the robustness of our method. For this reason, we have used benchmarks that have high resources demands, which makes it difficult to ensure a minimum IPC for a given PPT.

MediaBench benchmarks have a lower cache miss rate than SPEC CPU2000 benchmarks. Our results show that, on average in our baseline architecture, SPEC CPU benchmarks experience one L2 miss every 126 committed instructions, while MediaBench benchmarks experience a miss every 19,400 committed instructions. As shown in [25], MediaBench benchmarks are mainly CPU bounded rather than memory bounded. Given that SPEC CPU benchmarks experience more L2 misses, their resource demands are higher. The present paper is a "proof of concept" of how controlling SMT internal resources can enable SMTs to be used in real-time systems. Our purpose is to show the robustness of our method. For this reason, we have used SPEC benchmarks that have high resource demands, which makes it difficult to ensure a minimum IPC for a given PPT. It is clear that, if our mechanism works with applications with high resource requirements, like SPEC CPU benchmarks, it will work with applications with fewer resource requirements, like MediaBench benchmarks.

Traces of SPEC benchmarks were collected of the most representative 300 million instruction segment, following an idea presented in [19]. The workloads consist of integer and fp programs from the SPEC2000 benchmarks. Each program was compiled using the DEC Alpha AXP-21264 C/C++ compiler with the *-O2 -non_shared* options and executed using the reference input set. Programs are divided into two groups based on their cache behavior (see Table 3): Those with an L2 cache miss rate higher than 1 percent are memory bounded (MEM) programs. The others are ILP programs. The L2 miss rate is calculated with respect to the number of dynamic loads.

## 6 Results

In this section, we show the results obtained with our strategy. We first analyze results when there is only one PPT. After that, we discuss results when there are two PPTs at the same time. The results focus on three main points. First, we show the average performance obtained for the PPT(s) for each workload type. Second, we show the throughput of the UPTs. And, third, we show the total throughput obtained.

### 6.1 One PPT

In this experiment, we consider all combinations in which the PPT is ILP or MEM and the UPTs are ILP or MEM also. A workload is identified by two parameters: the type of the PPT and the type of the UPTs. For example, a workload of type IM means that the PPT is ILP and the UPTs are MEM. For each of the four workload types, we create four different sets of threads to avoid having our results be biased toward a specific set of threads by taking all possible combinations from Table 4. That is, we vary the PPT, but keep the UPTs fixed in order to not create a prohibitively large number of workloads to examine. We selected as MEM benchmarks those with the highest L2 miss rate (`twolf` and `mcf` for

TABLE 4
Workloads for 1-Thread Prioritization

| | PPT | UPTs |
|---|---|---|
| ILP | gzip | eon |
| | bzip2 | gcc, wupwise |
| | mesa | gap, apsi, crafty |
| | fma3d | |
| MEM | twolf | art |
| | mcf | twolf, mcf |
| | art | mcf, equake, swim |
| | lucas | |

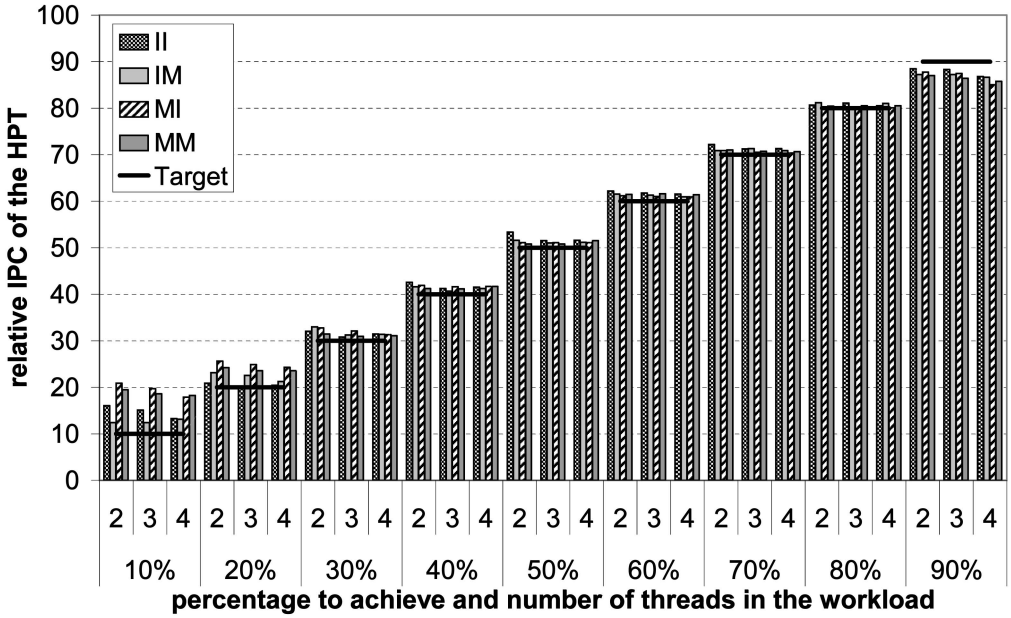


Fig. 7. Realized IPC values for the PPT. The $x$-axis shows the target percentage of full speed of the PPT and size of the workloads. The four different bars represent the four different types of workload discussed in Section 5.

integer benchmarks and `art`, `swim`, `lucas`, and `equake` for FP). ILP benchmarks have been selected randomly. In this section, we present average results for each group in each workload. For example, the II set with 3 threads (II3) is the average results of workloads `gzip+(gcc+wupwise)`, `bzip2+(gcc+wupwise)`, `mesa+(gcc+wupwise)`, and `fma3d+(gcc+wupwise)`. For all workloads, the simulation ends when the PPT finishes. Any UPT in the workload that finishes earlier is reexecuted.

### 6.1.1 PPT Performance

In Fig. 7, 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 PPT is given, ranging from 10 percent to 90 percent. For each size of the workload (two, three, or four threads), the achieved IPC for the PPT 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 percent). Only on the two extreme ends of the range of targets are we somewhat off target. We discuss the discrepancies for the 10 and 90 percent cases since they give the most insight into how our mechanism works.

**Ten percent case**: If the target IPC should be 10 percent 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 four IPC. Then, during 5 percent of the time during the sample phase, the PPT reaches this full speed. During the remaining 95 percent 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 percent of its full speed of four IPC. From Fig. 7, 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 PPT in the sampling period is lower due to more interference from the other threads. As a result, the overall IPC of the PPT drops a little.

Next, in the case of the IM workload, the memory bounded UPTs causes L2 cache pollution, more than in the case for the II workloads. Hence, the measured IPC of the PPT during the sample phase is lower than it should be and, during the tune phases, the PPT also suffers from interference. Hence, 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 percent, as it should be.

For the MI workload, the `mcf` benchmark has a full speed of 0.15 IPC. Hence, 10 percent 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: Ee measured a 30 percent 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 PPT, small differences in their IPCs again cause large relative errors. For the MM workload, the same explanation holds as for the MI workload.

**Ninety percent case**: At 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 UPTs are memory bounded, then they cause much pollution in the L2 cache. Hence, the IPC of the PPT we measure during the sample phases is lower than it should be. Moreover, during the tune phase, memory bounded UPTs 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 and, during the tune phase, we achieve an IPC value that is too low also.

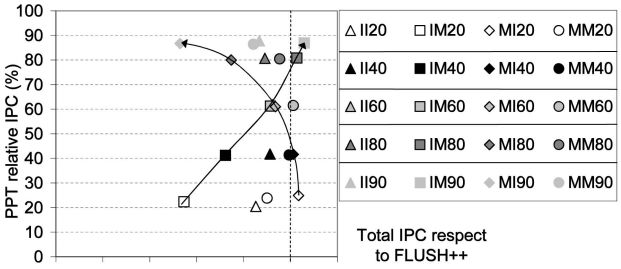


Fig. 8. QoS space where the $y$-axis shows the target percentage of full speed of the PPT and the $x$-axis the throughput obtained by our mechanism with respect to *flush++*. The legend shows different workload types as well as target percentages required for the PPT.

However, in case the UPTs 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 PPT in isolation and reach 100 percent of its full speed.

**Thirty percent to eighty percent case**: In more common situations where target percentages range from 30 to 80, we already achieve these percentages almost exactly, being less than 1 percent over target on average.

#### 6.1.2 Total Performance

In Fig. 8, we show a small variation of the QoS space presented in Section 2. In this case, the $y$-axis indicates the relative IPC of the unique PPT and the $x$-axis the throughput obtained by our mechanism with respect to *flush++* [5]. This fetch policy is an improved version of the *flush* policy proposed in [21]. Results are averaged for workloads with two, three, and four threads. The legend shows different workload types as well as the relative IPC required for the PPT. For example, the point II20 is the average result when the PPT is ILP and the UPTs are ILP (workload sets II2, II3, and II4), for which the PPT is run at 20 percent of its full speed. For clarity, we only show those percentages that are a multiple of 20. However, we also show the case of 90 percent because this high percentage poses particular problems.

In the II workloads (the triangles in Fig. 8), all threads are ILP and have a high throughput and do not occupy hardware resources for a long time. That means that both, the PPT and the UPTs highly profit resources. As a result, for intermediate target percentages, overall performance does not drop much compared to *flush++*. Only for the extreme target percentages is this performance slightly less. On average, our mechanism achieves 90 percent of the throughput obtained with *flush++* for the II workloads. In addition, we should take into account that this slowdow is also due to stopping the UPTs during the sample phases.

For the IM workloads (the squares in Fig. 8), the UPT threads are MEM and experience many loads that miss in the L2 cache. Hence, these threads have low IPC and tend to occupy resources for a long time, which has an adverse effect on the speed of the other threads in standard fetch policies. When the PPT is required to run at a low speed, the UPTs receive many resources and run at a speed that is close to the speed they would obtain under *flush++*. However, since the total throughput for *flush++* mainly derives from the speed of the PPT, total throughput is degraded. On the other hand, when we require a high relative IPC for the PPT which is ILP, total throughput increases because the PPT runs at a higher speed than it would under *flush++*. When the relative IPC of the PPT is 80 percent or higher, throughput is higher than with *flush++* because, although *flush++* is designed to deal with this situation by flushing a thread after an L2 miss, it also needs to refetch and reissue all flushed instructions, degrading its performance.

The MI workloads (the diamonds in Fig. 8) present the opposite situation of the previous one. In this case, the total throughput in *flush++* comes largely from the UPT. When the PPT is required to run at a low speed, the UPTs get many resources and can run faster than they would under *flush++*. Hence, total throughput is higher than under *flush++*. When the relative IPC of the PPT is 40 percent or less, throughput is higher than with *flush++*. Conversely, when the PPT runs at a high relative IPC, it is allocated many shared resources to achieve this. As a result, total throughput drops since the full speed of the PPT is low and also because the UPTs are denied many resources so that their speed becomes less than under *flush++*.

For the MM workloads (the circles in Fig. 8), neither the PPT nor the UPTs can make efficient use of resources. Hence, given that the PPT and the UPTs use resources in a similar way, there is not a significant variation in throughput when varying the target relative IPC of the PPT and it is almost the same as under *flush++*. Only for the extreme ends does total throughput drop since, in these cases, resources are occupied by either PPT or UPTs when some of these resources would have been used better under *flush++*.

#### 6.1.3 Summary

We can draw two main conclusions from these results. The first and most important one is that our QoS mechanism is capable of realizing a target IPC for a particular PPT within an error margin of less than 2 percent on average for relative IPCs from 10 percent to 80 percent. Another conclusion is that, at the same time, it maximizes total throughput, achieving relative IPCs of over 90 percent compared with the throughput obtained using *flush++*. In fact, *flush++* can be improved up to 40.2 percent, for the IM4 workload (`gzip` as the PPT and (`mcf + equake + swim`) as the UPTs), when the PPT requires 90 percent of its full speed. The OS level job scheduler could take advantage of the trends shown in Fig. 8 in order to improve throughput. For example, if the scheduler needs to deal with a workload consisting of a nontime critical MEM thread and a number of ILP threads, it can be advisable to run this MEM thread as an PPT with a low target percentage of its full IPC: The overall throughput can be larger than for *flush++*.

### 6.2 Two PPTs

In our experiments with two PPTs, we vary the required target IPC for both High Priority Threads from 10 percent to 80 percent with a step of 10 percent. Hence, we study our mechanism for target IPCs for PPT0 and PPT1 of $10/10, 10/20, \ldots, 80/80$, a total of 64 different combinations

TABLE 5
Subset of Threads for 2-Thread Prioritization

| | PPT0 | PPT1 | UPTs |
|---|---|---|---|
| ILP | gzip<br>mesa | gcc<br>gap | wupwise<br>apsi, crafty |
| MEM | twolf<br>art | twolf<br>mcf | mcf<br>equake, swim |

of target percentages. Furthermore, we consider combinations where the PPT0, the PPT1, and the UPTs can all be either ILP or MEM; this leads to eight possible types of workload. We also study the situation when there are both one and two UPTs. Hence, there are $8 \times 2 = 16$ types and, for each type, 64 combinations of target percentages, hence $64 \times 16 = 1,024$ experiments.

A complete study of all benchmarks is not feasible due to excessive simulation times. We have used the benchmarks shown in Table 5. For each type of workload, we use two different sets of threads to minimize any bias toward a specific set of threads. Hence, a total of $1,024 \times 2 = 2,048$ experiments have been carried out. Table 6 shows the composition of each of the eight types of workload. Since we want to have two workloads for each type, we also use the workloads obtained by replacing `gzip` and `twolf` as PPT0 by `mesa` and `art`, respectively. In this table, the "key" column denotes the type of the workload, i.e., III means that the PPT0, the PPT1, and the UPT(s) are ILP.

### 6.2.1 Speed of PPT0 and PPT1

Fig. 9a and Fig. 10b show the resulting QoS spaces for each type of workload.[4] In these figures, the $x$-axis shows the relative IPC of PPT0 and the $y$-axis the relative IPC of PPT1. The legend shows the target percentage required for the PPT0 and for the PPT1, respectively. For clarity, we only show those percentages that are multiples of 20. The main conclusions we can draw from these charts are the following:

- In all cases, we achieve the required percentage for the PPT0 or with an error lower than 2 percent on average. This indicates us that we are effectively isolating the execution of the PPT0 from the other threads. Furthermore, the addition of PPT1 does not affect PPT0.
- There are almost no differences when the type of the UPTs is changed (parts (a) and (b) in each figure). This indicates that, in our mechanism, the UPTs do not affect the execution of PPT0 and PPT1.
- For the PPT1, we do not always achieve the required percentage.[5] We differentiate three cases.
  - When PPT0 is ILP (the I$XX$ workloads), the combinations that fail are 40/80, 60/80, 80/40, 80/60, and 80/80.
  - When PPT0 is MEM (the M$XX$ workloads), the combinations that fail are 60/80, 80/40, 80/60, and 80/80.
  - If both PPT0 and PPT1 are MEM (the MM$X$ workloads), the combinations 60/60 and 40/80 also fail.

As a rule of thumb, we can realize the required percentages $X$ and $Y$ for PPT0 and PPT1, respectively, for all types of workload if $X + Y \leq 100$.

That we fail in some cases to realize the target percentages for both PPT0 and PPT1 is not due to incorrectly isolating PPT1, but to an insufficient amount of resources inside the processor. Fig. 11 shows the amount of shared resources (IQ entries and physical regs) and ways of the L2 cache required to achieve a given percentage of the relative IPC of the PPT0. Obviously, the higher the required percentage, the higher the amount that needs to be reserved. It is interesting to observe that, when PPT0 is ILP and a relative percentage higher than 70 percent is required, then the amount of reserved resources is 50 percent or higher. This means that we cannot run both PPT0 and PPT1 at a relative IPC of 70 percent or higher. The same happens when the PPT is MEM when the required percentage is 60 percent or more.

Concerning the number of ways in the L2 cache, we observe that, as we increase the required percentage of the full speed of PPT0, the amount of ways that need to be reserved decreases for both types of PPT0. This is because, as we increase the percentage to achieve, the number of conflicts in the tune periods due to the other threads decreases since these threads are given fewer opportunities to run.

For the MEM threads, four ways of the 8-way cache need to be reserved in order to avoid significant performance degradation and failure to achieve the given target percentage. Only when the target IPC for the PPT is 70 percent or more do fewer ways need to be reserved. This shows that, when both threads are MEM, the conflicts in the L2 cache can slow down the PPT1 because it can only reserve up to two ways. For the ILP threads, the number of reserved ways required is much lower and this number decreases as the target IPC increases.

### 6.2.2 Total Throughput

In this section, we take a closer look at the throughput obtained by our mechanism for each of the different workload types and target percentages in turn. In all cases, two main observations characterize the obtained performance. First, better results are achieved if more resources are given to ILP threads than when these resources are devoted to MEM threads. Second, when we have to divide resources between threads with similar characteristics (all threads are MEM or ILP), then nonextreme target percentages lead to better performance results.

Fig. 12a to Fig. 13b show, for each workload type and required percentages for the PPT0 and the PPT1, the performance obtained with our policy with respect to the performance obtained with *flush*++. The results depending on the workload type are the following: As before, we only

4. Due to lack of space, we only show the I$XX$ workloads. The trends for M$XX$ workloads are the same that for I$XX$ workloads

5. In Fig. 9 and Fig. 10, the shadowed area represents, approximately, those points which our mechanism is not able to accomplish with the OS requirements

TABLE 6
Workloads for 2-Thread Prioritization

| Thread type | | | Key | 3 threads | 4 threads |
|---|---|---|---|---|---|
| PPT0 | PPT1 | UPTs | | workload 0 | workload 0 |
| I | I | I | III | gzip, gcc, wupwise | gzip, gap, apsi, crafty |
| | | M | IIM | gzip, gcc, mcf | gzip, gap, equake, swim |
| | M | I | IMI | gzip, twolf, wupwise | gzip, mcf, apsi, crafty |
| | | M | IMM | gzip, twolf, mcf | gzip, mcf, equake, swim |
| M | I | I | MII | twolf, gcc, wupwise | twolf, gap, apsi, crafty |
| | | M | MIM | twolf, gcc, mcf | twolf, gap, equake, swim |
| | M | I | MMI | twolf, twolf, wupwise | twolf, mcf, apsi, crafty |
| | | M | MMM | twolf, twolf, mcf | twolf, mcf, equake, swim |

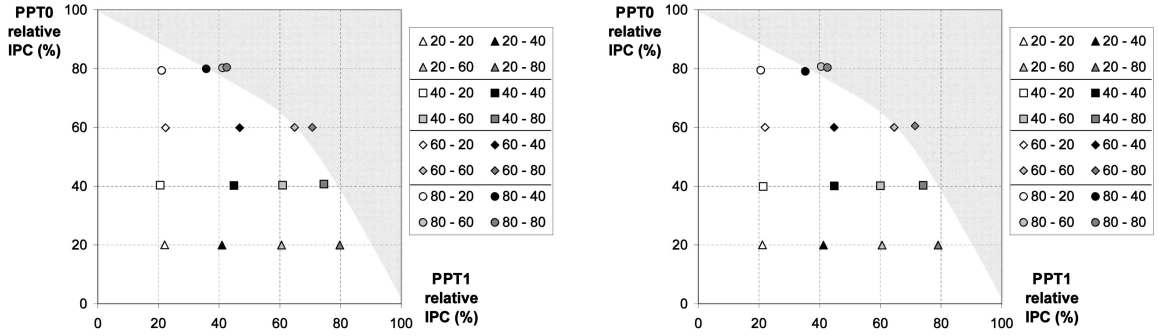


Fig. 9. Quality of Service (QoS) space for II*X* workloads: (a) III workloads and (b) IIM workloads.

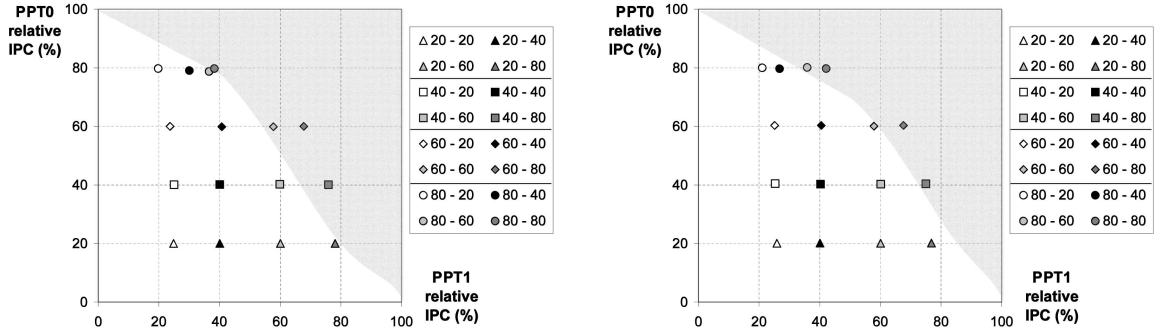


Fig. 10. Quality of Service (QoS) space for IM*X* workloads: (a) IMI workloads and (b) IMM workloads.

show the result for I*XX* workloads. The conclusions for M*XX* workloads are the same.

**Type III.** Given that all threads have similar behavior, the best results are achieved for intermediate percentages, that is, for percentages between 30 and 70 percent. The best result is achieved when the PPT0 requires 50 percent of its full speed and the PPT1 60 percent, obtaining 91 percent of the throughput obtained with *flush*++. This small drop in throughput is due to the fact that *flush*++ is implemented on top of *icount*. *icount* achieves good results for ILP threads and the division of resources disrupts its behavior. In some cases, resources are reserved for threads that are stalled when these resources could have been used by the other threads. However, our mechanism obtains 82.4 percent of the performance obtained with *flush*++ on average, showing that our mechanism does not need to pay too high a price in order to meet OS requirements.

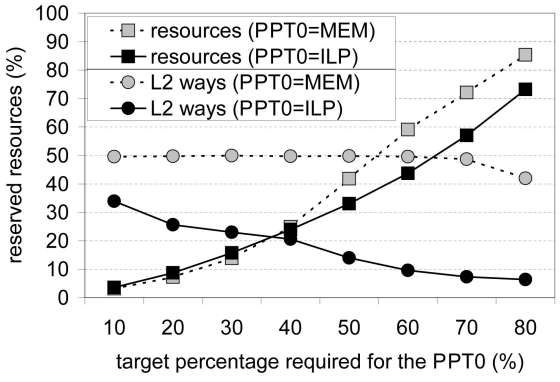


Fig. 11. Shared resources required to achieve a given relative IPC.

**Type IIM.** In this case, the higher the rIPC of both the PPT0 and the PPT1, the higher the throughput. This is caused by the fact that, in these cases, a lower number of resources is given to the UPTs that are MEM and that have a low IPC by themselves. Hence, under *flush*++, the UPTs tend to occupy resources for a long time which degrades the performance of the ILP threads. In our situation, the UPTs are not given the opportunity to hold many resources and,

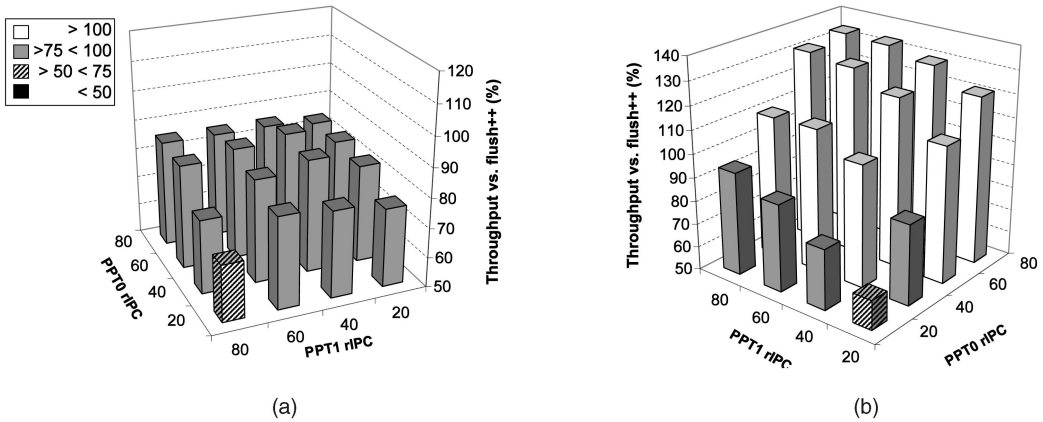


Fig. 12. Total throughput with respect to the *flush++* instruction fetch policy for II*X* workloads: (a) III workloads and (b) IIM workloads.

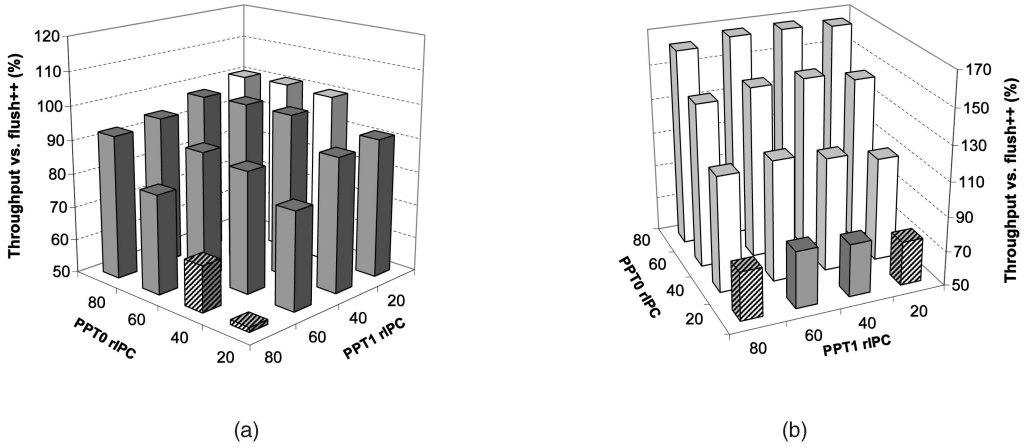


Fig. 13. Total throughput with respect to the *flush++* instruction fetch policy for IM*X* workloads: (a) IMI workloads and (b) IMM workloads.

as a result, the PPTs reach a higher speed than under *flush*++ and total throughput increases.

**Type IMI.** Better results are achieved as the rIPC of the PPT0 increases, excluding the extreme cases of 20/80 and 40/80. For a given rIPC of the PPT0, better results are achieved when the rIPC of the PPT1 is small because this PPT1 is MEM and then receives fewer resources. Since the UPTs are ILP, they obtain a high speed under *flush*++. Using our mechanism, they reach a much lower speed since many resources are dedicated to the PPTs and they cannot use these resources, even if they would be idle. Hence, throughput is degraded compared to *flush*++ since PPT1 is allocated many resources.

**Type IMM.** The higher the rIPC of the PPT0, the better the results since more resources are given to this ILP thread. For a given rIPC of the PPT0, better results are obtained for intermediate values of the PPT1. Using *flush*++, PPT0 would obtain a speed that is larger than 20 percent of its full speed. Hence, fixing the required speed of PPT0 at 20 percent, many resources go to slow MEM threads. This increases their throughput, but, since this throughput is small to start with, total throughput is less than for *flush*++. Conversely, if PPT0 receives many resources in order to realize a high target percentage, it runs much faster than under *flush*++. Since the type of the workload is IMM, this implies that total throughput is much higher than for *flush*++.

### 6.2.3 UPTs Throughput

Fig. 14a shows the performance of the UPTs when they are ILP, that is, the average of the III, IMI, MII, and MMI workloads (*XX*I). Fig. 14b shows the UPTs performance when they are MEM. In these figures, *rIPC* denotes the target percentage (or relative IPC) of a thread requested by the OS. As expected, as the sum of the target percentages becomes less, UPTs receive more resources and their IPC increases. Conversely, if this sum becomes larger, the throughput of the UPTs decreases. However, their throughput never becomes zero and, even in the cases where a high

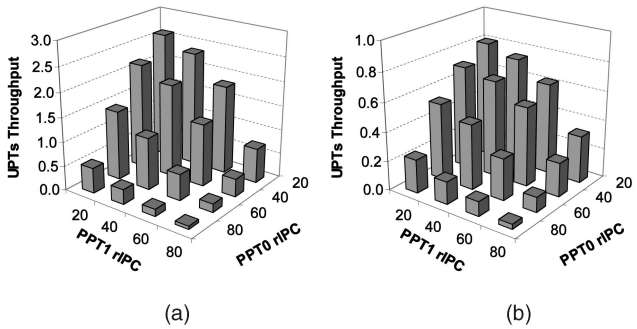


Fig. 14. UPTs throughput: (a) *XX*I workloads and (b) *XX*M workloads.

target percentage for both PPT0 and PPT1 is to be reached, the UPTs still have some throughput. This shows that our mechanism reserves a minimal amount of resources for the PPTs to realize the required target throughput for them and can successfully use the remaining resources for the UPTs, even when there are only a limited number of fetch slots, IQ entries, etc., that are not claimed by the PPTs.

#### 6.2.4 Summary

The results have shown that our mechanism always achieves the required target IPC for the PPT0 with an error less than 2 percent on average. The addition of a minimum required IPC for a second thread, PPT1, does not affect the behavior of PPT0. For PPT1, the target is not achieved when the amount of resources is too small to realize both percentages at the same time.

Compared to a traditional superscalar processor and previous SMT fetch policies, our mechanism enables one degree of freedom more in scheduling jobs on an SMT processor: Jobs can be given a certain *share* of the available resources. Our experiments show that a resource conscious job scheduler, by using this level of freedom, could significantly improve a traditional one on top of a fetch policy. On average, our mechanism achieves 90.4 percent of the performance achieved by *flush*++. For some workloads, we achieve an improvement of 228 percent: For the IMM3 workload ($mesa$ as PPT0, $twolf$ as PPT1, and $mcf$ as UPT), *flush*++ achieves an IPC of 1.782, whereas our mechanism, when the target percentage for the PPT0 and the PPT1 is 80 percent, leads to an IPC of 4.074.

## 7 Conclusion

Current Operating Systems view the different contexts of an SMT as multiple independent virtual processors. As a result, the OS schedules threads onto what it regards as processing units operating in parallel. However, in an SMT, those threads are competing with each other for the resources of a single processor. On the other hand, in current SMT processors, resource allocation is done implicitly as determined by the fetch policy. Both factors, lead to performance unpredictability as the OS is not able to guarantee priorities or time constraints on the execution of a thread if that thread is to be run concurrently with other threads. As a result, we need to run time-critical applications in isolation, degrading overall performance. We have proposed a novel strategy that enables a bidirectional interaction between the OS SMT processors, allowing the OS to run up to two time-critical jobs at a predetermined IPC, regardless of the workload that these threads are executed in. As a consequence, this enables the OS to run time-critical jobs without dedicating all internal resources to them and, so, other low priority jobs can make significant progress as well.

We have tested our mechanism in a 4-context SMT with up to two time-critical jobs (PPT0 and PPT1), for which we require a minimum IPC. When one thread is prioritized, our mechanism achieves, for the entire range of workloads and target percentages, up to 80 percent, the required percentage or a little more. When two threads are to be prioritized at the same time, we always reach the target percentage for the PPT0 with an error less than 2 percent on average. The addition of a minimum required IPC for a second HPT, the PPT1, does not affect PPT0. For the PPT1, the target is not achieved only when the amount of resources is too small to accomplish this. This allows the OS, by designating pme or two PPTs as well as their target percentages, to control the execution of these threads regardless of the workload they are executed in. As an additional advantage, in some cases, our mechanism improves throughput compared to *flush*++, achieving 90 percent of the performance of *flush*++. This shows that, if the target were to maximize performance, then the synergy of the OS job scheduler and our workload and resource conscious policy could significantly outperform the best currently known fetch policies for SMT.

## Acknowledgments

This work has been supported by the Ministry of Science and Technology of Spain under contract TIN-2004-07739-C02-01 and grant FP-2001-2653 (Francisco J. Cazorla), the HiPEAC European Network of Excellence, an Intel fellowship, and the EC IST program (contract HPRI-CT-2001-00135). The authors would like to thank Oliverio J. Santana, Ayose Falcón, and Fernando Latorre for their work in the simulation tool and the reviewers for their helpful comments.

## References

[1] http://www.cpuid.org/k8/index.php, 2006.
[2] *IA-32 Intel Architecture Software Developer's Manual. Volume 3: System Programming Guide,* 2006.
[3] A. Anantaraman, K. Seth, K. Patil, E. Rotenberg, and F. Mueller, "Virtual Simple Architecture (VISA): Exceeding the Complexity Limit in Safe Real-Time Systems," *Proc. 30th Int'l Symp. Computer Architecture (ISCA),* pp. 350-361, June 2003.
[4] D.C. Bossen, J.M. Tendler, and K. Reick, "Power4 System Design for High Reliability," *IEEE Micro,* vol. 22, no. 2, pp. 16-24, 2002.
[5] F.J. Cazorla, E. Fernandez, A. Ramirez, and M. Valero, "Improving Memory Latency Aware Fetch Policies for SMT Processors," *Proc. Fifth Int'l Symp. High Performance Computing (ISHPC),* pp. 70-85, Oct. 2003.
[6] F.J. Cazorla, E. Fernandez, A. Ramirez, and M. Valero, "DCache Wam: An I-Fetch Policy to Incrase SMT Efficiency," *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS 2004),* Apr. 2004.
[7] F.J. Cazorla, E. Fernandez, A. Ramirez, and M. Valero, "Dynamically Controlled Resource Allocation in SMT Processors," *Proc. 37th MICRO,* pp. 171-182, 2004.
[8] D. Chiou, P. Jain, S. Devadas, and L. Rudolph, "Dynamic Cache Partitioning via Columnization," *Proc. Design Automation Conf.,* June 2000.
[9] A. El-Moursy and D.H. Albonesi, "Front-End Policies For Improved Issue Efficiency in SMT Processors," *Proc. Ninth Int'l Symp. High Performance Computer Architecture (HPCA),* pp. 31-42, Feb. 2003.
[10] S. Hily and A. Seznec, "Contention on 2nd Level Cache May Limit the Effectiveness of Simultaneous Multithreading," Technical Report 1086, IRISA, 1997.
[11] R. Jain, C.J. Hughes, and S.V. Adve, "Soft Real-Time Scheduling on Simultaneous Multithreaded Processors," *Proc. Fifth Int'l Symp. Real-Time Systems Symp.,* Dec. 2002.
[12] R. Kalla, B. Sinharoy, and J. Tendler, "SMT Implementation in POWER 5," *Proc. Hot Chips,* Aug. 2003.
[13] P.M.W. Knijnenburg, A. Ramirez, J. Larriba, and M. Valero, "Branch Classification for SMT Fetch Gating," *Proc. Sixth Workshop Multithreaded Execution, Architecture, and Compilation (MTEAC),* pp. 49-56, 2002.

[14] K. Krewell, "Fujitsu Makes SPARC See Double," *Microprocessor Report,* Nov. 2003.
[15] M. Levy, "Multithreaded Technologies Disclosed at MPF," *Microprocessor Report,* Nov. 2003.
[16] K. Luo, J. Gummaraju, and M. Franklin, "Balancing Throughput and Fairness in SMT Processors," *Proc. Int'l Symp. Performance Analysis of Systems and Software (ISPASS),* pp. 164-171, Nov. 2001.
[17] 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 J.,* Feb. 2002.
[18] M.J. Serrano, R. Wood, and M. Nemirovsky, "A Study on Multistreamed Superscalar Processors," Technical Report #93-05, Univ. of California Santa Barbara, 1993.
[19] T. Sherwood, E. Perelman, and B. Calder, "Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications," *Proc. 10th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT),* Sept. 2001.
[20] A. Snavely, D.M. Tullsen, and G. Voelker, "Symbiotic Job Scheduling with Priorities for a Simultaneous Multithreaded Processor," *Proc. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-9),* pp. 234-244, Nov. 2000.
[21] D. Tullsen and J. Brown, "Handling Long-Latency Loads in a Simultaneous Multithreaded Processor," *Proc. 34th Int'l Symp. Microarchitecture,* pp. 318-327, 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," *Proc. 23rd Int'l Symp. Computer Architecture (ISCA),* pp. 191-202, Apr. 1996.
[23] D.M. Tullsen, S. Eggers, and H.M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," *Proc. 22nd Int'l Symp. Computer Architecture (ISCA),* pp. 392-403, 1995.
[24] R.E. Wunderlich, T.F. Wenisch, B. Falsafi, and J.C. Hoe, "SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling," *Proc. 30th Int'l Symp. Computer Architecture (ISCA),* pp. 84-97, June 2003.
[25] Z. Xu, S. Sohoni, R. Min, and Y. Hu, "An Analysis of Cache Performance of Multimedia Applications," *IEEE Trans. Computers,* vol. 53, no. 1, pp. 20-38, Jan. 2004.

**Francisco J. Cazorla** received the PhD degree from the Polytechnic University of Catalonia (UPC) in 2005. He is currently a full-time researcher in the Barcelona Supercomputing Center (BSC). He was a summer student intern with IBM's T.J. Watson Research Laboratory in New York for six month. His research interests include high-performance architectures with emphasis on multithreaded architecture.

**Peter M.W. Knijnenburg** received the PhD degree of computer science from Utrecht University in 1993. He is currently an assistant professor in computer science with the Computer Systems Architecture Group at the University of Amsterdam, The Netherlands. He has been a visiting researcher at the University of Illinois at Urbana-Champagne, Edinburgh University, INRIA Rennes, and UPC Barcelona. His main research interests are in adaptive and iterative compilation and the interaction of compilers and computer architectures.

**Rizos Sakellariou** received the PhD degree in computer science from the University of Manchester in 1997. Since January 2000, he has been a lecturer in computer science at the University of Manchester. Prior to his current appointment, he was a visiting assistant professor at the University of Cyprus (fall 1999) and a postdoctoral research associate at Rice University (1998-1999) and the University of Manchester (1996-1998). His primary area of research is in parallel and distributed systems, but his research interests also include compilers, computer architecture, performance modeling and evaluation, scheduling, and the interactions between them.

**Enrique Fernández** received the Industrial Engineering degree from the Polytechnic University of Las Palmas in 1983 and the PhD degree in computer science from the University of Las Palmas de Gran Canaria (ULPGC) in 1999. He is an assistant professor in the Informática y Sistemas Department at ULPGC. His current research interests are in the field of high-performance architectures.

**Alex Ramirez** received the PhD degree in computer science from the Universitat Politècnica de Catalunya (UPC), Spain, in 2002. He is an associate professor in the Computer Architecture Department of UPC and leader of the Computer Architecture Group at the Barcelona Supercomputing Center. His research interests focus on high performance embedded processor architectures, including multithreading and vector architectures. Currently, he is involved in research and development projects with Intel and IBM.

**Mateo Valero** received the PhD degree from the Universitat Politècnica de Catalunya (UPC), Spain, in 1980. He is a professor in the Computer Architecture Department at UPC. His research interests focus on high-performance architectures. He has published approximately 300 papers on these topics. Dr. Valero has been honored with several awards, including the King Jaime I by the Generalitat Valenciana and the Spanish national award "Julio Rey Pastor" for his research on IT technologies. In 2001, he was appointed a fellow of the IEEE, in 2002, an Intel Distinguished research Fellow, and, in 2003, a fellow of the ACM. Since 1994, he has been a foundational member of the Royal Spanish Academy of Engineering. In 2005, he was elected a correspondent academic of the Spanish Royal Academy of Mathematics, Physics, and Natural Sciences. In 2006, he was elected an academic of the Royal Academy of Science and Arts.