Architectural Support for Real-Time Task Scheduling in SMT Processors

Francisco J. Cazorla
UPC[*] and BSC[†]
Barcelona. 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
Universidad de Las Palmas
de Gran Canaria. Spain
efernandez@dis.ulpgc.es

Alex Ramirez
UPC and BSC
Barcelona. Spain
aramirez@ac.upc.es

Mateo Valero
UPC and BSC
Barcelona. Spain
mateo@ac.upc.es

ABSTRACT

In Simultaneous Multithreaded (SMT) architectures most hardware resources are shared between threads. This provides a good cost/performance trade-off which renders these architectures suitable for use in embedded systems. However, since threads share many resources, they also interfere with each other. As a result, execution times of applications become highly unpredictable and dependent on the context in which an application is executed. Obviously, this poses problems if an SMT is to be used in a real-time system.

In this paper, we propose two novel hardware mechanisms that can be used to reduce this performance variability. In contrast to previous approaches, our proposed mechanisms do not need any information beyond the information already known by traditional job schedulers. Nor do they require extensive profiling of workloads to determine optimal schedules. Our mechanisms are based on dynamic resource partitioning. The OS level job scheduler needs to be slightly adapted in order to provide the hardware resource allocator some information on how this resource partitioning needs to be done. We show that our mechanisms provide high stability for SMT architectures to be used in real-time systems: the real time benchmarks we used meet their deadlines in more than 98% of the cases considered while the other thread in the workload still achieves high throughput.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: realtime and embedded systems; C.1.3 [**Processor Architectures**]: Other Architecture Styles pipeline processors

General Terms

Design, Performance

Keywords

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

[*]Universitat Politènica de Catalunya.
[†]Barcelona Supercomputing Center.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CASES'05, September 24–27, 2005, San Francisco, California, USA.
Copyright 2005 ACM 1-59593-149-X/05/0009 ...$5.00.

1. INTRODUCTION

Current processors take advantage of Instruction Level Parallelism (ILP) to execute in parallel several independent instructions from a single instruction stream (thread). However, the amount of ILP available in each thread may be limited due to data and control dependences [12]. Providing as many hardware resources to a thread as could potentially be used in some phases during its execution means that in other phases those resources would sit idle. Clearly, this degrades the performance/cost ratio of these processors.

A solution to improve the performance/cost ratio of processors is to allow threads to share hardware resources. In current processors, resource sharing can occur in different ways. At one extreme of the spectrum, there are multiprocessors (MPs) that only share some levels of the memory hierarchy. On the other extreme, there are 'full-fledged' simultaneous multithreaded processors (SMTs) that share many hardware resources.

SMT processors are a viable option in embedded real-time systems. On the one hand, real-time systems increasingly require high computation rates and SMTs are able to provide such computational power. On the other hand, SMTs have a good performance/cost and performance/power consumption ratio due to their high resource sharing [9], which is desirable in real-time systems. However, SMT processors have a problem that makes their use in real-time systems difficult. In SMT processors, threads interfere because they share many resources. This implies that the speed a thread obtains in one workload can differ significantly from the speed it has in another workload [4]. We refer to this by

saying that an SMT processor has a high *variability*. Obviously, high variability is an undesirable property in a real-time environment. In such environments, not only does the job scheduler need to take into account Worst Case Execution Times (WCETs) and deadlines when selecting a workload, but it should also know how the workload can affect the Worst Case Execution Time. Note that redefining WCET as the longest execution time in an arbitrary workload is not an option. By carefully selecting a workload, the WCET could be made arbitrarily large. Moreover, analytical approaches to WCET would fail miserably if they would need to take a context into consideration.

The execution time of a thread depends on the amount of execution resources given to it. In a system with single-thread processors, a thread has exclusive access to all available hardware resources, hence this problem is not present here. In an SMT processor, the amount of resources given to a thread varies dynamically. An *instruction fetch policy*, e.g., *icount* [11], decides how instructions are fetched from the threads, thereby implicitly determining the way internal processor resources are allocated to the threads. The key point is that current fetch policies are designed with the main objective of increasing processor throughput: this may differ from the objective of the global system, in our case, a real-time environment. This could compromise the objective of the real-time system to meet deadlines of tasks [2] if some action is not taken.

To sum up, in current systems the performance/cost versus variability trade-off is clear. MPs have low variability but a worse performance/cost ratio than SMTs. SMT implies a good cost-performance relation but high variability in the execution time of an application depending on the context in which it is executed. We would like to add an SMT some mechanisms to provide low variability while overall performance and performance/cost ratio remain as unaffected as possible.

In the literature, several solutions have been proposed in order to improve this trade-off in SMTs [5][6][8][10]. The common characteristic of these solutions is that they assume knowledge of the average number of Instructions Per Cycle (IPC) of applications when they are executed in isolation, called IPC_{alone}. In other words, these solutions are *IPC based*. This implies, as we show later, that these solutions are applicable for a subset of real-time applications, where the IPC_{alone} of applications can be a priori determined.

As far as we know, there does not exist a proposal that deals with this problem when the IPC_{alone} of applications is not known. In this case, time-critical threads are given all the resources of the SMT in existing approaches [2]. This, of course, solves the problem but provides low throughput. In this paper, we propose a novel mechanism to enforce real time constraints in an SMT based system. This mechanism consists of a small extension of the OS level job scheduler and an extension of the SMT hardware, called a *Resource Allocator*. Our approach is *resource based* instead of IPC based. By this, we mean that it relies on the amount of resources given to the time-critical thread. The job scheduler assembles a workload for the SMT processor and instructs the Resource Allocator to dedicate at least a certain amount of resources to the time critical thread so that it is guaranteed to meet its deadline. Apart from this, the Resource Allocator tries to adjust the resource allocation in order to maximize performance. The current paper is focused on the Resource Allocator. In future work, we hope to give a working implementation of the job scheduler as well. Using our method, time-critical applications meet their deadline more than 98% of the cases considered while the non-critical applications obtain high throughput.

This paper is structured as follows. Section 2 presents some background on real-time scheduling. Section 3 presents related work. Section 4 explains our experimental setup. Section 5 presents our two proposals and section 6 the simulation results. In section 7, we explain the changes required to implement our mechanisms. In Section 8 we compare all the techniques we present throughout this paper. Finally, section 9 presents some conclusions.

2. BACKGROUND ON REAL-TIME SCHEDULING

In this section we discuss some of the challenges to develop a real-time scheduler for SMT processors. We focus on real-time systems with periodic task sets. For each task, the scheduler knows three main parameters. First, the *period*, that is, the interval at which new instances of a task are ready for execution. Second, the *deadline*, that is, the time before which an instance of the task must complete. For simplicity, the deadline is often set equal to the period. This means that a task has to be executed before the next instance of the same task arrives in the system. Third, the *Worst Case Execution Time* ($WCET$) is an upper bound on the time required to execute any instance of the task, which is guaranteed never to be exceeded.

In soft-real time scheduling, many algorithms have been used to schedule the task set in single-threaded systems (e.g., EDF or LLF). However, these algorithms are no longer sufficient in an SMT processor, since the execution time of a thread is unpredictable when this thread is scheduled with other threads. Algorithms should be adapted to meet this new situation.

As shown in [8], the problem of scheduling a task set turns into two different problems in SMT systems. First, to select the set of tasks to run. This problem is called the *workload selection problem*. Second, to determine how resources are shared between threads. In this paper, we focus on the latter problem that is also known in the literature as the *resource sharing problem*.

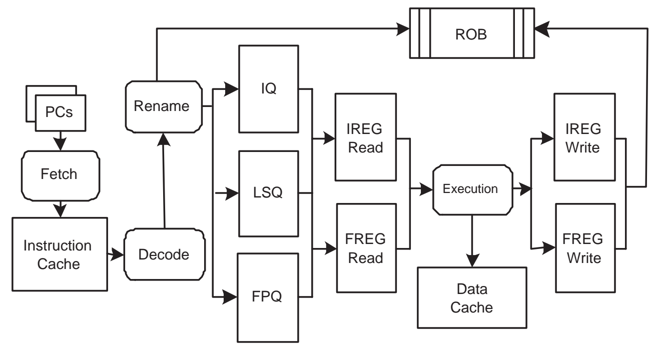
The high variability of SMTs implies that the task of a real-time job scheduler for SMT processors is much more complex and challenging than for single-threaded processors. When scheduling a job, the job scheduler must take into account the amount of resources given to a thread, which is implicitly decided by the instruction fetch policy in current systems, in order to ensure that it meets its deadline.

3. EXISTING APPROACHES

In [2], the authors propose an approach where the WCET is specified assuming a virtual simple architecture (VISA). At execution time, a task is executed on the actual processor. Intermediate virtual deadlines are established based on the VISA. If the actual processor is an SMT and a task fails to meet its intermediate deadlines, the SMT is switched to single-threaded mode, to ensure that tasks can meet their deadlines. The authors conclude that fetch policies that attempt to maximize throughput, like *icount*, should be "balanced" for minimum forward progress of real-time tasks.

Processor Configuration	
Pipeline depth	12 stages
Fetch/Issue/Commit Width	8
Queues Entries	64 int, 64 fp, 64 ld/st
Execution Units	6 int, 3 fp, 4 ld/st
Physical Registers	256 integer, 256 fp
(shared)ROB size	512 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	2048 Kbytes, 8-way, 8-bank, 64-byte lines, 20 cycle access
Main memory latency	300 cycles
TLB miss penalty	160 cycles

(a) Main parameters



(b) Schematical view

Figure 1: Baseline configuration

This is precisely the target of our paper: we ensure a minimum amount of resources for a given time-critical thread so that it meets its deadline regardless of the other threads executed in its workload. Our approach is orthogonal to the VISA framework, making it suitable for hard-real time systems: a time-critical thread is executed on the actual SMT processor that provides the thread with a given percentage of resources. In the event that the task does not meet its intermediate deadlines, instead of switching the processor to single-threaded mode, we can increase the amount of resources given to it, so that it meets its deadline and its overall performance does not drop drastically.

To the best of our knowledge, there are three main studies dealing with real-time scheduling for SMTs. Two of these studies focus on real-time systems [5][8], while the third focuses on general-purpose systems [10]. In [8], the authors focus on workload selection in soft-real time systems, although they also briefly discuss the resource sharing problem. The authors propose a method to solve the problem of high variability of SMTs by profiling all possible combinations of tasks. By comparing the IPC of a thread when it is executed in a given workload, IPC_{SMT}, with the IPC that the thread achieves when it is run in isolation, IPC_{alone}, the slowdown that the thread suffers from being executed in a context is determined. This information is given as additional input to the scheduler and is used to maximize performance, since the scheduler selects those workloads that lead to the highest *symbiosis* among threads and thus the highest performance. The main drawback of this solution is the prohibitively large number of profiles required. For a task set of N tasks and a target processor with K contexts, we have to profile all $\frac{N!}{K!(N-K)!}$ possible combinations. A similar solution is proposed in [10].

Finally, in [5][6] we have proposed a hardware mechanism to run a given thread at a given percentage of its full speed, IPC_{alone}, in an arbitrary workload. If it is required to run a thread at a target IPC that is X% of the IPC_{alone} of that thread, then the IPC of the critical thread is periodically measured and the mechanism tries to run that thread at X% of the last measured speed. It has been shown that this approach can realize an arbitrary required percentage of the $IPCalone$ of a critical thread in widely different workloads.

A common characteristic of these studies is that they are *IPC based*, that is, they require the IPC_{alone} of threads. By comparing the IPC of a thread in a workload with its IPC_{alone}, these methods estimate the execution time slowdown suffered by the critical thread in a given workload. In this way, previous methods determine if the critical thread is going to meet its deadline. In this paper, we propose a different way of approaching the problem. Instead of using the IPC of applications to drive the solution, we use *resource allocation* that normally is implicitly driven by the instruction fetch policy. Our method makes explicit to the scheduler the amount of resources used by each thread. The scheduler adjusts this allocation to guarantee that applications meet their deadlines.

The advantage of our method is two-fold. First, it is well known that IPC values can be highly dependent on the input of applications. For some types of real-time applications, such as multimedia applications, this dependence is weak [7]: the IPC of such an application is roughly independent from the input. But for other types of applications this is not the case. Our method does not require this information so that it is applicable to all types of real-time applications. Second, we achieve a similar or even better success rate than the approaches discussed above, while improving overall performance.

4. EXPERIMENTAL ENVIRONMENT

In this section, we discuss our baseline architecture used to run our experiments, the benchmarks we use, and the metrics we employ to compare the different proposals.

4.1 SMT Simulator

We use a trace driven SMT simulator derived from smtsim [11] to evaluate the performance of the different policies. 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.

We use an aggressive configuration, shown in figure 1(a): many shared resources (issue queues register, functional units, etc.), out-of-order execution, very wide superscalar, and a deep pipeline for high clock frequency. These features cause the performance of the processor to be very unstable, depending on the mix of threads. Thus, this configuration represents an unfavorable scenario where we evaluate our proposals. It is clear that, if those proposals work in this hard configuration, they will work better in narrower processors with fewer shared resources.

Figure 1(b) gives a schematical view of our processor. Our baseline instruction fetch policy is *icount*. It determines from which threads instructions are fetched, implicitly allocating internal processor resources. This is a major cause of performance variability in SMTs: the speed of a thread highly depends on the context in which it is executed. Next, instructions are decoded and renamed in order to track data dependences. When an instruction is renamed, it is allocated an entry in the window or issue queues (integer, floating point and load/store) until all its operands are ready. Each instruction also allocates one Re-Order Buffer (ROB) entry and a physical register in the register file. ROB entries are assigned in program order and instructions wait in this buffer until all earlier instructions are resolved. When an instruction has all its operands ready, it reads its operands, executes, writes its results, and finally commits.

4.2 Benchmarks

We use workloads consisting of two threads. The first thread is the *critical thread* (CT) that represents the thread with the most critical time restriction or the soft real-time thread. The second thread is a *non-critical thread* (NCT) that is assumed either to have less-critical time restrictions or to have no time restrictions at all. As critical threads we use programs from the MediaBench benchmark suite, namely, *adpcm*, *epic*, *g721*, *gsm*, and *mpeg2*. We used both the coder and the decoder of these media applications. Hence, we use 10 media applications as critical threads. Table 1 shows the inputs for each of the MediaBench benchmarks.

In this paper, we want to check the efficiency of the resource allocator under scenarios where the NCT requires many resources, and thus, where the performance of the CT could be more affected. For this reason, we use as non-critical threads benchmarks from the SPEC2000 integer and fp benchmark suite that require more resources than media applications. Each of the ten media applications is executed with 8 different SPEC benchmarks as non-critical thread. We have used *gzip*, *mesa*, *perlbmk*, *wupwise*, *mcf*, *twolf*, *art* and *swim*. These benchmarks were selected because they exhibit widely varying behavior. Some are memory bounded, which means that they generate many cache misses. Others are not, but consume many computational resources. In our experiments below, we have used all pairs of media and general purpose applications, giving us a total of 80 workloads.

In order to check the efficiency of our Resource Allocator, we consider three different scenarios that differ in the stress that is put on our mechanism. The *worst-case utilization* U_w is defined as the fraction $U_w = \frac{WCET}{P}$ where $WCET$ is the Worst Case Execution Time and P is the period of an application. If the utilization is low, then $WCET$ is much smaller than the period and hence it should be relatively easy to guarantee deadlines. If, on the other hand, the utilization is high, then the critical thread must be given many or even all resources. In this case, it may happen more frequently that a CT misses a deadline.

In this paper, the WCET of an application is set equal to its real execution time when it is run in isolation in the SMT processor, called $ExecTime_i$. We consider three worst-case utilization factors called *low*, *medium*, and *high*. In the first case, we model a situation where the job scheduler has to schedule one task with a low worst-case utilization of 30%: we establish as a deadline for each task $3.3 \times ExecTime_i$. Hence, $U_i = \frac{WCET_i}{P_i} = \frac{ExceTime_i}{3.3 \times ExceTime_i} = 30\%$. In the second scenario, we model a medium utilization of 50% so that for each task its deadline is $2 \times ExecTime_i$. Finally, in the worst-case scenario, we use a high utilization of 80%. In this case, the deadline for each task is $1.25 \times ExecTime_i$.

Table 1: MediaBench benchmarks we have used.

Benchmark name	Media	Language	input
adpcm	speech	C	clinton.pcm
epic	image	C	test_image.pgm
gsm	speech	C	clinton.pcm
g721	speech	C	clinton.pcm
mpeg2	video	C	test2.mpeg

4.3 Metrics

In all our experiments, we run the CT until completion. If the NCT finishes earlier, it is started again. When the CT finishes, we measure three values. First, the success rate (SR) that indicates the frequency the CT finishes before its deadline. In typical real-time systems, it is the responsibility of the OS level job scheduler to provide a high success rate. In our approach, this responsibility is shared between the job scheduler and the resource allocator. Second, we measure the performance of the non-critical thread. We want to give a *minimum* amount of resources to the critical thread to meet its deadline. The remaining resources are given to the non-critical thread in order to maximize its throughput.

Both these values are required to quantify the efficiency of our approach. For example, if a given CT has an utilization of 30% and the scheduler orders the resource allocator to assign to it 100% of the resources, the thread will obviously meet its deadline. This provides a success rate of 100% but it does not provide high throughput nor does it show the efficiency of the resource allocator. Analogously, if a thread has an utilization of 90% and the scheduler orders the allocator to give it 10% of the resources, the thread likely misses its deadline and we do not know anything about the efficiency of the resource allocator.

As a third measure, in addition to the success rate, we measure the extra time required to finalize the CT for those cases in which the CT misses its deadline. Assume that the time required to execute the CT in a given workload, denoted by $Exectime_{CT}$, is larger than its deadline, $deadline_{CT}$, so that the CT misses its deadline. Then the *variance* is computed as: $variance_{CT} = \frac{(Exectime_{CT} - deadline_{CT})}{deadline_{CT}} \cdot 100\%$. For a given policy, we take the five cases in which the variance is highest and compute the average of these variances. We call this metric *Mean5WorstVariance*. If a policy has a success rate of 1, we have that $Exectime_{CT} \leq deadline_{CT}$ for each workload and in this case the variance is 0.

5. DYNAMIC RESOURCE PARTITIONING

In this section, we discuss the extensions to the OS level job scheduler and the SMT hardware for implementing our scheme.

5.1 Overview of our approach

The basis of our mechanism is to *partition* the hardware resources between the critical and the non-critical thread and to reserve a *minimum* fraction of the resources for the CT that enables it to meet its deadline. In this way, we can also satisfy our second objective, namely, to increase as much as possible the IPC of the NCT. It is the responsibility

of the job scheduler to provide the resource allocator with some information so that it can reserve this fraction for the critical thread.

When the WCET of a task is determined, it is assumed that this task has full access to all resources of the platform it should run on. However, when this task is executed in a multithreaded environment together with other tasks, it uses a certain fraction of the resources. It is obvious that when the amount of resources given to a thread is reduced, its performance decreases as well. The relation between the amount of resources and performance is different for each program and may vary for different inputs of the same program. For the benchmarks used in this paper, we have plotted this relation in Figure 2. This figure shows the *relative IPC* [1] of each multi media application when it is executed alone on the SMT as we vary the amount of window entries and physical registers given to it. This relative IPC is the percentage of the IPC the application achieves when executed alone on the machine and given all the resources, called IPC_{alone}. From this figure, we can see that if we dedicate 10% of the resources to the *epic* decoder, we obtain 50% of the speed it would have were it given the entire machine. Likewise, 10% of the resources dedicated to the *adpcm* decoder gives 95% of its IPC_{alone}.

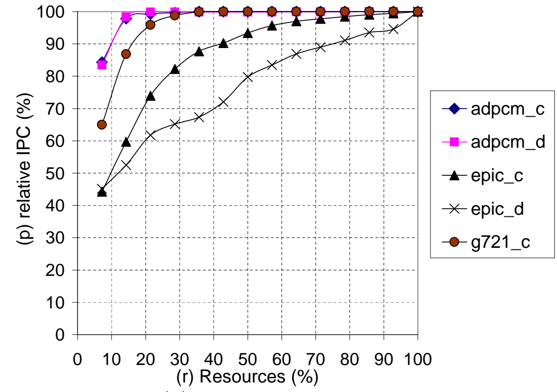
Our proposed method exploits the relation between the amount of resources given to the critical thread and the performance it obtains. When the OS level job scheduler wants to execute a critical thread, given its $WCET$ and a period P, it simply computes the allowable performance slow down, S, given by $S = \frac{P}{WCET}$. For such a value of S, each instance of this job finishes before its deadline. Suppose the real execution time of this instance is T_i. Then, $T_i \leq WCET$. Hence, $S \cdot T_i = \frac{P}{WCET} \cdot T_i \leq \frac{P}{WCET} \cdot WCET = P$. Therefore, the value of S is a critical piece of information needed to establish a resource partitioning.

The following two issues need to be addressed. First, we need to determine which resources are being controlled by the resource allocator. Second, we need to decide whether the job scheduler or the resource allocator determines the exact amount of resources given to the critical thread. In the first case, a resource allocation is fixed for the entire period a critical thread is executing. We call this approach the *static* approach. In the second case, the resource allocator can dynamically vary the amount of resources dedicated to the critical thread. We call this approach the *dynamic* approach.

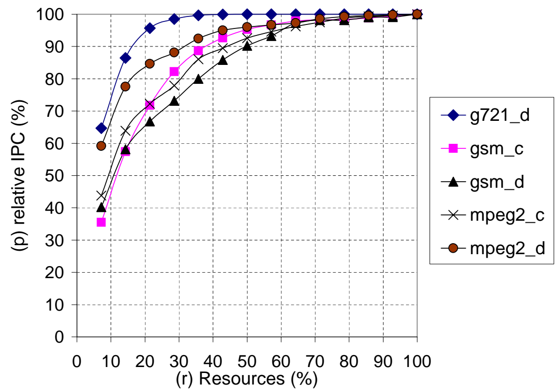
5.2 Resource allocator

The *resource allocator* controls the amount of resources that can be used by applications. It consists of a number of *resource usage* counters that track the amount of resources used by each application, one counter per resource. These counters are incremented each time a thread needs an additional instance of a resource and they are decremented each time an instruction releases an instance resource. For each thread in the SMT, there are also *limit* registers for each resource that contain the maximum number of instances the thread is allowed to use. These limit registers can be written by either the job scheduler in the static method or the resource allocator itself in the dynamic method. If an application tries to use more resources than it is assigned, its instruction fetch is stalled until resources are freed. We would like to emphasize that all added registers are special purpose registers. They do not belong to the register file. We discuss the hardware cost of our mechanisms in more detail in Section 7.

[1] Recall that IPC is inverse to performance: $ExecutionTime = CycleTime \times \#Instructions \times (1/IPC)$.



(a) First five applications



(b) Last five applications

Figure 2: Relation between the amount of resources given to a task and its IPC.

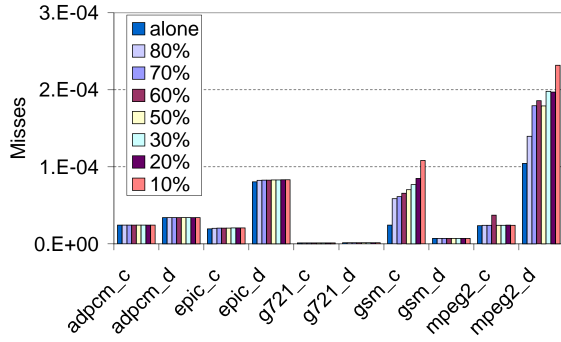
5.3 Resources

The first step in our approach is to determine the set of shared resources that has to be controlled to provide stability. In our architecture, the shared resources are the following: the fetch bandwidth, the issue queues, the issue bandwidth, the physical registers, the instruction cache, the L1 data cache, the unified L2 cache, and the TLBs. We have conducted a number of experiments to examine what the influence on variability is when we partially dedicate each of these resources to the CT.

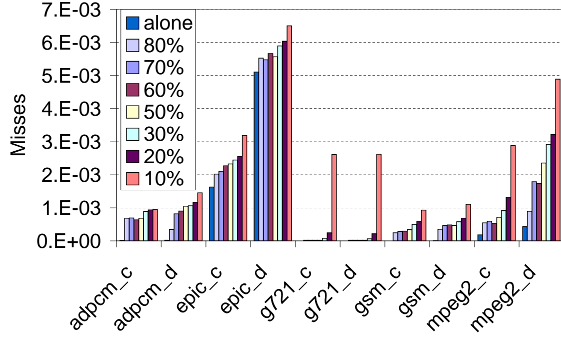
5.3.1 Caches and TLBs

Regarding TLBs, on average for all the experiments made in this paper, the number of data TLB misses per instruction is 3.6×10^{-4} and the number of instruction TLB misses is 8.2×10^{-7}. Hence, the influence on the execution time of the CT is small. For this reason we do not control TLBs.

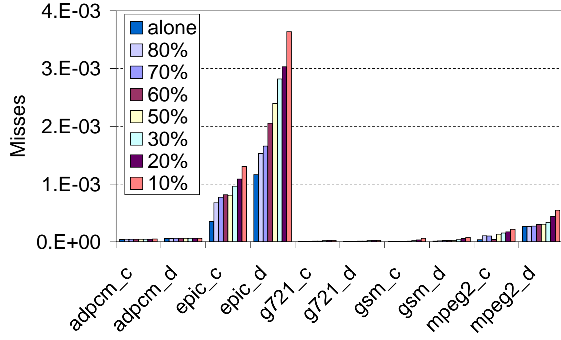
Regarding caches, we measured for all multi-media applications in all 80 workloads the average number of misses in each cache with respect to the number of committed instructions, as we vary the amount of resources given to it. The results are shown in figure 3. We observe that there is an



(a) Instruction cache



(b) Data cache

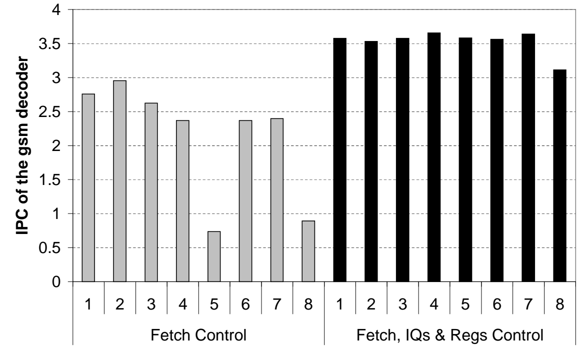


(c) L2 cache

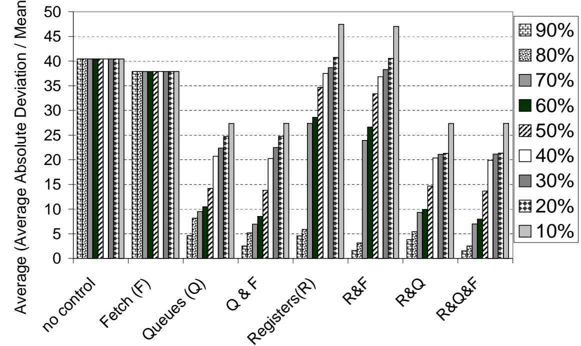
Figure 3: Cache interference introduced by the NCT as we vary the amount of resources given to the CT.

increase in cache miss rate caused by interference by another application in the workload. We observe as well that this interference is lower when the amount of resources given to the CT is higher and *vice versa*. This is caused by the fact that if the CT is allowed to use many resources, the NCT executes slower and hence uses caches less frequently and thus produces less interference.

The absolute number of misses per committed instruction is low: lower than 2×10^{-4} for the instruction cache, 7×10^{-3} for the data cache, and 4×10^{-3} for the L2 cache. We can draw two conclusions from these figures. First, in the icache there is almost no interference between the CT and the NCT. Second, the interference introduced in the caches by a non-critical thread in a workload is so small that we expect that this only slightly affects the execution time of multimedia applications. As a result, we do not need to control how caches are shared between the threads.



(a) IPC when we control the fetch (gray bars), and the fetch, the IQs and regs. (black bars).



(b) Average of the fraction (Standard Deviation / Mean).

Figure 4: Variability in CT's IPC in different workloads as we vary the resources under control.

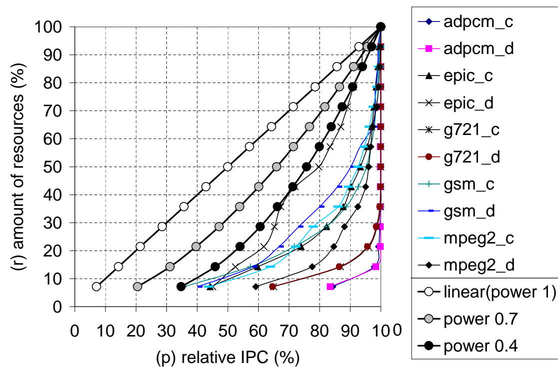
5.3.2 Other resources

We systematically measured the effect of controlling resources other than the caches. We looked at the following resource partitions. *Nothing* means that we do not control any resource inside the SMT. Resources are implicitly shared as determined by the default fetch policy. *Fetch* means that we prioritize the CT when fetching instructions from the instruction cache. *Queues* and *Registers* mean that we give a fixed amount of entries of that resource to the CT. Furthermore, we made all combinations of these resources [2].

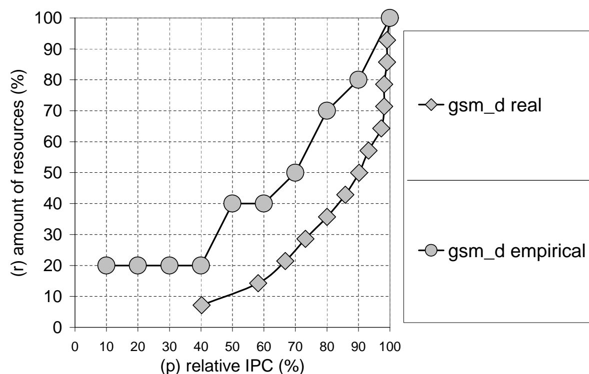
In Figure 4(a), we show for the *gsm* decoder benchmark its actual IPC values when it is executed with each of the eight SPEC CPU benchmarks shown in Section 4.2. These benchmarks are indicated as 1, 2, ..., 8 in Figure 4(a). IPC values are shown for two possible ways to partition resources: when we prioritize instruction fetch and when we partition the registers and the issue queues. From this figure, it is immediately clear that controlling the instruction fetch alone gives little control over the speed of the CT and the variability in IPC is large. On the other hand, controlling queues, registers and fetch does give much control over the speed of the CT and hence the variability is low.

In order to measure the sensitivity of the variability to resource partitioning more systematically, we proceed as follows. We used all pairs of media benchmarks as CT

[2]The issue bandwidth provides small variations in the results. For this reason we do not show its results.



(a) Different arguments for the Power function



(b) Empirical function

Figure 5: Different performance/resources functions

and spec2000 benchmarks as NCT. We measured execution times of the CT. For each CT from the MediaBench suite, we obtained 8 numbers, one for each spec benchmark. We computed the mean and the standard deviation of these numbers, and the fraction deviation/mean. In this way, we obtain a measure of the variability in execution time of a CT as we change the NCT, expressed relative to the average execution time. This allows us to average these values over all possible critical threads. This final average is the overall measure of variability used in this study. In Figure 4(b) we show these results. We can immediately observe that when we do not control resource allocation, we get a high variability of 40%. This can be interpreted as that in many cases the difference in execution time of a CT in an arbitrary context can be as high as 40% of the total execution time or even higher. If we only prioritize the instruction fetch of the CT, this variability is hardly reduced. The most important resources to control are the registers and the issue queue entries. The best results are obtained when we control everything: we give the CT priority in the fetch stage and reserve a certain amount of registers and issue queue entries for it. By controlling these resources we do not eliminate interference completely but we are able to reduce them to a great extent. These are the resources controlled by our Resource Allocator below.

Regarding the percentage of resources given to the CT, we show that as we decrease this amount the variability increases. For low percentages (10 or 20%) variability is even higher than when no control is carried out. This is mainly because when the CT uses few resources, the NCT executes more instructions, causing more interferences. In addition, every time the CT misses in the cache, it has not enough resources to hide this latency, even for L1 data cache misses. Hence, we conclude that if we give less than 20% of resources to the CT, its deadline could be compromised.

5.4 Static approach

In this section, we discuss our static approach to resource partitioning. In this approach, the job scheduler computes *a priori* the resource partitioning that is used throughout the entire period of the critical thread. In Figure 2, we have plotted the relation between the amount of resources dedicated to the CT and the performance it obtains. It clearly follows that, for all benchmarks considered in this paper, the relation between performance and amount of dedicated resources is super-linear. That is, if we dedicate $X\%$ of the resources, we obtain more than $X\%$ of IPC_{alone} and in some cases much more. Since the job scheduler knows the WCET of the critical thread and the period, P, in which it should execute, it knows the slow down the CT can suffer: the slow down factor $S = \frac{P}{WCET}$ discussed above. Hence, given this fraction S of the performance of the CT, it needs to compute a function $f(1/S) = Y$ to determine that the CT needs $Y\%$ of the resources to obtain this performance. We call such a function a *performance/resource function* or *p/r function*. These p/r functions can be considered to be approximations to the inverse of the curves shown in Figure 2. Hence, when the job scheduler assembles a workload with a certain application as critical thread, it computes the value of S and determines the corresponding value $Y = f(1/S)$ for a p/r function f. Then it instructs the resource allocator to reserve $Y\%$ of the resources for this critical thread. In the next subsection, we discuss performance/resource functions in more detail.

5.4.1 Performance/resource functions

Figure 5(a) shows the actual p/r relation of each thread and several p/r functions that approximate this actual p/r relation, which can be used by the job scheduler. These functions are plotted as circles in the figure. We show several functions that are given by $r = f(p) = p^{1/value}$ for *value* equal to 1 (linear), 0.7, and 0.4. For lower *values*, the amount of resources given to the CT is reduced and the actual p/r relation is better approximated. This may be positive since we allow the NCT to use more resources. However, this may also compromise the success rate.

For our experiments discussed in the next section, we use the p/r functions described above. We moreover use another p/r function that is more directly based on the graphs shown in Figure 2, called the *empirical* function. In order to construct this function, we empirically determine for each Multimedia Benchmark and for each possible value of p, the value of r that leads to a good success rate and achieves high NCT performance. We may consider this function as the upper bound of our static method. Figure 5(b) shows an example of how the function *empirical* is determined. The diamonds show the actual p/r relation for the gsm_d benchmark. The circles show the approximation we used.

Note that this approximation is slightly larger than the corresponding value in the curve in Figure 2 in order to take into account interference by the NCT, mainly for low percentages of p.

5.5 Dynamic approach

In this approach, the resource allocator dynamically determines the amount of resources given to the critical thread. The main advantage of this method is that it adapts to program execution phases, increasing overall performance. In the next section, we show that the dynamic approach provides better results than the static approach but it is only applicable if the application under consideration has a number of characteristics that we discuss in more detail below. The main advantage of the static approach is that it can be used always.

The mechanism is based 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 instant throughout its execution. In the present case, if we want to slow down an task with a factor S, it is sufficient to slow it down with a factor S at every instant. The dynamic approach that exploits this observation is a simplification of the method we proposed in [5][6], called *Predictable Performance* or *PP*. The resource allocator distinguishes two phases that are executed in alternate fashion. We briefly describe these phases below. For more information, please consult [5]. During the first phase, the *sample phase*, all resources under control are given to the CT and the NCT is temporarily stopped. As a result, we obtain an estimate of the current IPC_{alone} of the CT which we call the *local* IPC_{alone}. The sample phase starts with a warm up period of 50,000 cycles to remove pollution by the NCT from the shared resources. Next, we measure the IPC_{alone} of the critical thread in an *actual-sample* phase of 10,000 cycles.

During the second phase, the *tune phase*, the NCT is allowed to run as well. Our mechanism dynamically varies the amount of resources given to the CT to achieve an IPC that is equal to the local $IPC_{alone} \times 1/S$. The tune phase, which lasts 300,000 cycles, is divided in periods of 15,000 cycles during which the realized IPC of the CT is measured. If this measured value is lower than required, the CT is assigned more resources. If it is higher than required, resources are taken away from the CT and given to the NCT.

The main difference between the present dynamic approach and the *Predictable Performance* mechanism from [5][6] is that in *PP* the real value of IPC_{alone} of the CT is used to compute resource allocations in the tune phase. This value has to be provided by the OS, in contrast to the present approach that does not require this value. Moreover, the present approach controls fewer resources. In particular, we do not exercise control over the caches, in contrast to *PP* in which L2 cache miss rates of the critical thread are monitored and this information is used to dedicate part of the L2 cache exclusively to the critical thread.

The main difficulty in our dynamic method is to measure accurately the local IPC_{alone} of the CT, due to the pollution created by the NCT in the shared resources. As we have shown in [5][6], the main source of interaction among the CT and the NCT is the L2 cache. This pollution stays for a long time, up to 5 million cycles. We have analyzed the pollution caused by the NCT in this resource in detail. We found that for multi media applications the measured value of the IPC_{alone} is 1% lower than the real value. For SPEC benchmarks used in [5][6], it is 8% lower. The main reason for this is that media applications have a smaller working set than spec benchmarks. For this reason, an NCT does not interfere as much with media applications as with spec benchmarks. As a result, we can use a more simple resource partitioning algorithm for media applications than the algorithm from [5][6] that is geared toward general purpose applications.

We conclude that, if applications under consideration have a small working set in comparison with the L2 cache size, then they are unlikely to be affected by a NCT with a much larger working set. As a result, the IPC measured in the sample phase is closer to the actual IPC_{alone}, what allows us to leave the L2 miss rate out of consideration, thereby considerably simplifying the mechanism. If this condition is not satisfied, the dynamic approach cannot be applied and we have to resort to either the static approach discussed above or to the expensive *PP* mechanism described in [5][6].

To summarize, in the dynamic approach, the job scheduler provides the value $1/S$ to the resource allocator. Next, the resource allocator determines the IPC_{alone} of that instance of the task during a sample phase and reduces its IPC by a factor of $1/S$ during the subsequent tune phases. This implies that the CT can meet its deadline and that we minimize the amount of resources given to the CT, enabling high performance of the NCT.

6. SIMULATION RESULTS

In this section, we present the results of the static and dynamic approaches. Moreover, we show the results obtained using the *Predictable Performance* mechanism we presented in [5][6] and a fetch control like mechanism based on [2][10].

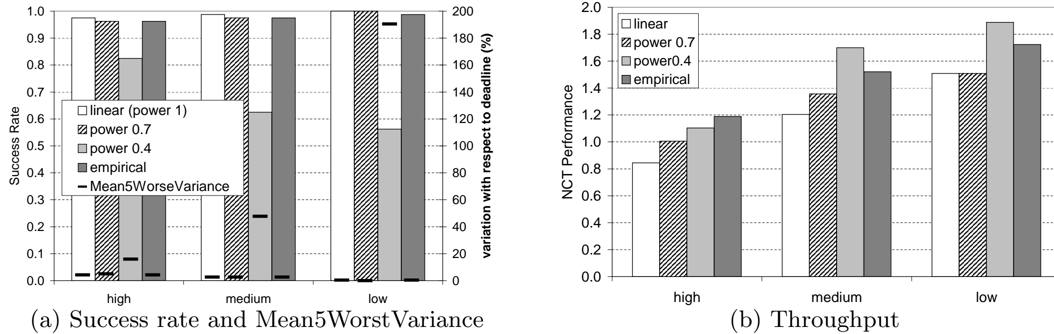
6.1 Static method

Figure 6 shows the success rate and the performance for the different p/r functions used in the static method. In figure 6(a), bars show the success rate and are measured on the left y-axis. Lines show the Mean5WorstVariance and are measured on the right y-axis.

In figure 6(a), we can see that the linear p/r function provides the best success rate. We also observe that when the p/r function is more aggressive, the success rate decreases. This is intuitively clear, since we reduce the amount of resources given to the CT. All functions, except the function $f(p) = p^{1/0.4}$, achieve a good success rate and Mean5Worst-Variance. As we move from high to low utilization scenarios, the success rate improves. On average, the success rate is 0.987, 0.979, and 0.671 for the linear, $f(p) = p^{1/0.7}$, and $f(p) = p^{1/0.4}$ functions, respectively. For the empirical function the success rate is 0.975. The Mean5WorstVariance is 1%, 2.6%, and 87% and 2.4%, respectively.

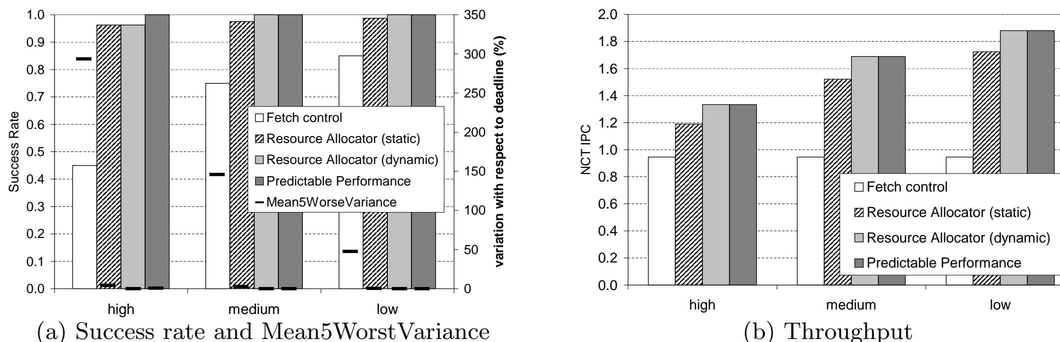
Figure 6(b) shows the average IPC of the NCT, averaged over all experiments. We see that the function $f(p) = p^{1/0.4}$ achieves the best performance results. However, this is at the cost of success rate. Hence, we conclude that this function is too aggressive. We observe that as we increase the aggressiveness of the p/r function, we obtain more performance for the NCT. This is because more aggressive p/r functions provide the CT with fewer resources.

We conclude that the *empirical* performance/resource function performs best. If this function is too difficult to ob-



(a) Success rate and Mean5WorstVariance (b) Throughput

Figure 6: Success Rate and throughput as we change the p/r relation



(a) Success rate and Mean5WorstVariance (b) Throughput

Figure 7: Success Rate and throughput of our approach and previous approaches

tain in some circumstances, the p/r function $f(p) = p^{1/0.7}$ performs only slightly worse. Recall that a p/r function is computed by the OS level job scheduler and hence is implemented in software. Hence we can easily use complex functions like the functions discussed above.

6.2 Dynamic method

In this section, we present the results of the dynamic method and moreover compare this mechanism with previous approaches. For this experiment, we have also considered the Predictable Performance mechanism proposed in [5][6] and a prioritization-aware fetch policy [2][10] which we call *fetch control* in this section. This mechanism always prioritizes the CT when fetching instructions. We also compare our results with the *empirical* p/r function for the static method.

Figure 7(a) shows the success rate for the different approaches. If we just control fetch, we see that we obtain a low success rate and a high Mean5WorstVariation, even for the low utilization scenario. The predictable performance approach obtains a success rate of 1, and hence a Mean5WorstVariation of 0. This is mainly due to the fact that this policy uses knowledge of the IPC_{alone} of each CT that allows it to compute dynamically how far the current IPC of the CT is from the target IPC. In this way, this mechanism converges to the target IPC. Our mechanism does not require this information but nevertheless achieves a success rate of 0.987. It also has a low Mean5WorstVariance of 1.63%. In the static method using the *empirical* p/r function, the SR is 0.975 and the Mean5WorstVariance is 2.4%.

Regarding throughput, we can see that our dynamic method achieves the same performance as Predictable Performance. Our static method achieves 9% less performance, but still much more performance than the fetch control method, up to 56% more in the low utilization scenario.

7. IMPLEMENTATION

Our two proposals require some hardware to control the amount of resources given to the CT and NCT. In this section, we present the hardware changes in our baseline architecture to provide such functionality. Finally, we show how the OS and the hardware collaborate to deal with time requirements.

7.1 Hardware to control resource allocation

The objective of this hardware is to ensure that the CT is allowed to use at least a given amount of each shared resource. The hardware requirements needed to implement this functionality are similar to the hardware requirements to implement the mechanism we proposed in [3]. The tasks done by this hardware are: track, compare, and stall.

Track: We need a *resource usage counter* for each resource under control, both for the CT and the NCT. Each counter tracks the number of slots that each thread has of that resource. Figure 8 shows the counters required for a 2-context SMT if we track the physical registers. Resource usage counters are incremented in the decode stage (indicated by *(1)* in Figure 8). Register usage counters are decremented when the instruction commits (2). We also control the oc-

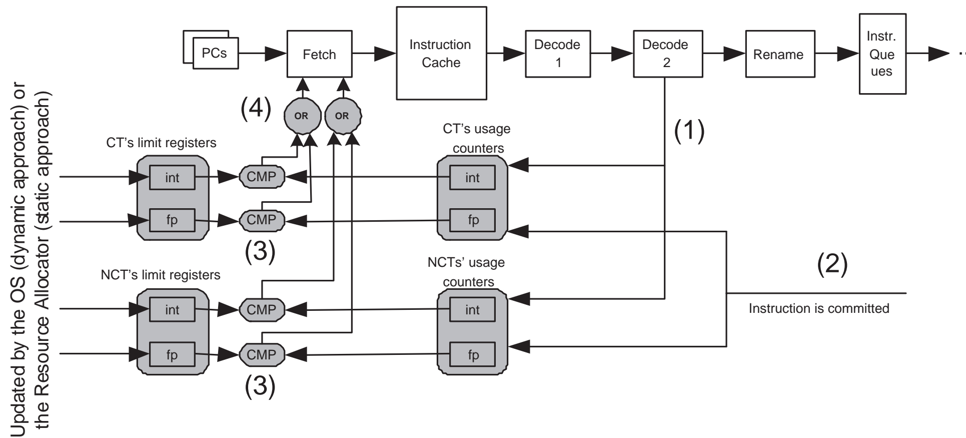


Figure 8: Hardware required to implement our mechanism

cupancy of the IQs. Hence, 3 queue usage counters are required, one for each queue. Queue usage counters are decremented when instructions are issued from the issue queues. All added registers are special purpose registers. They do not belong to the register file. The design of the register file is left unchanged with respect to the baseline architecture. 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 [1].

Compare: We also need two registers, *limit registers*, that contain the maximum number of entries that the CT and NCT are entitled to use. These registers are modified either by the OS in the dynamic approach and by the resource allocator in the static approach. In the example shown in Figure 8, we need 4 counters: one for the fp registers and one for the integer registers for both the CT and NCT. Every cycle we compare the resource usage counters of each thread with the limit registers (3). If a threads 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.

7.2 OS/hardware collaboration

For the static approach, the OS only has to update the values of the (special purpose) limit registers in order to accomplish with the tasks' deadlines. When the OS schedules a task for execution, it also sets the value of these registers [3].

For the dynamic approach, the OS sets the percentage of the IPC_{alone} of the CT that the hardware has to achieve. This requires the addition of one register. In addition to the hardware discussed in the previous section, we use a Finite State Machine (FSM) to dynamically change the resource allocation to converge to the target IPC. This FSM is quite simple and can be implemented with 4 counters and simple control logic. The FSM starts by giving all resources to the CT (sample phase). This is done by simply setting the limits registers of the CT to the maximum number of resources, and resetting the entries of the NCT. Next, at the end of the warm-up phase, we begin to compute the IPC of the CT. At the end of the actual-sample phase, we compute the local target IPC and set the resource allocation to converge to the local target IPC. At the end of each tune sub-phase, we vary the resource allocation so that the IPC of the CT converges to the target IPC.

[3]Note that, if the limit registers of the CT and the NCT are set to the maximum number of resources, no thread is stalled. That is, we would have a standard SMT processor guided by the fetch policy.

8. DISCUSSION

In this section, we briefly summarize the results presented above and discuss some other issues, namely, the cost and the applicability of the mechanisms discussed above.

Concerning the cost of the different mechanisms, the *fetch control* mechanism used in [10] only prioritizes the fetch of the critical thread and hence has lowest cost. Our static mechanism needs to keep track of how many resources are used by each thread and hence is more expensive. However, the cost for doing this is not high, as is shown in section 7. Since the required percentage that the critical thread should receive is provided by the job scheduler, the base resource allocator is enough to implement our static method. Our dynamic method is more complex. Apart from the resource allocator that is required to monitor that threads do not exceed their share of the resources, a mechanism in hardware is required to sample the IPC_{alone} of the critical thread during the sample phase and to periodically determine the IPC of this tread during the tune phase. Moreover, logic is required to suspend the NCT during the sample phase and to adjust resource allocation during the tune phase. Finally, Predictable Performance requires all this plus extra logic to monitor the L2 cache. Moreover, the logic required to determine resource allocation after the sample phase is more complex than the logic required by our dynamic mechanism proposed in the present paper.

Concerning the applicability of the various approaches, both *fetch control* and our static method can be used for all applications. Our dynamic method requires that the non-critical thread does not interfere too much with the critical thread in the L2 cache. Predictable Performance requires that all instances of an application have more or less the same IPC. Fortunately, it has been shown [7] that media

Table 2: Comparing approaches shown in this paper

Metric	fetch control	static	dynamic	PP
Success rate	-- (<75%)	++ (>95%)	++	++
Throughput NCT	--	+	++	++
Applicability	++	++	+	-
Cost	--	-	+	++

applications have these properties so that PP can be applied. Hence we can summarize all aspects in table 2. In this table, we use the following symbols: ++ (very high), + (high), - (low), and -- (very low).

Depending on the properties of applications that need to run on the system, the amount of hardware available to provide soft real-time functionality, and the required success rate, a designer of an embedded, real-time system can choose one of the alternatives discussed in this paper. If there is hardly any room to implement a real-time mechanism, *fetch control* can be used which has a poor success rate but costs next to nothing. If there is a modest amount of real estate available and the success rate must be reasonably high, our static method is best suited. If, on the other hand, the success rate must be 1 then Predictable Performance can be used, at the cost of a complex implementation. In situations in between, our dynamic method may be a good candidate.

9. CONCLUSIONS

In this paper, we have proposed two novel approaches to the problem of enabling SMT processors for soft-real time systems. The main problem of using SMT processors in real-time systems is that in an SMT processor threads share almost all hardware resources. This may may cause interference between threads which implies that the speed a thread obtains in one workload can be very different from the speed it has in another workload [4]. In contrast to previous approaches, our methods do not require any knowledge beyond information that is traditionally used by the OS level job scheduler, namely, Worst Case Execution Time and the Period of the time-critical thread. Neither do our methods require extensive profiling of candidate workloads like some other methods do [8][10]. Our methods are based on resource partitioning, in contrast to previous approaches that are IPC based, reserving a minimum fraction of all resources for the critical thread so that it can just reach its deadline. In this way, the non-critical threads also receive as many resources as possible so that their throughput is maximized at the same time. In the first method, the job scheduler determines the fraction of resources dedicated to the critical thread and this fraction is fixed during the entire period. In the second method, the SMT hardware extension of a resource allocator dynamically adjusts the amount of resources for the critical thread, thereby adapting to program phases which increases the throughput of the non-critical threads even more. We have compared our approaches to two previously published mechanisms, namely, *fetch control* [2][10] and *Predictable Performance* [5][6]. We have shown that we significantly outperform *fetch control* and are almost as good as *Predictable Performance* using a much less complex mechanism. On average, the critical thread meets its deadline in 98% of the cases considered. We have discussed the pros and cons of all 4 mechanisms to explore the design space of real-time enabled SMT processors in detail.

10. ACKNOWLEDGEMENTS

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 Jim Smith for his technical comments in the development of this paper. Authors would also thank to Oliverio J. Santana, Ayose Falcón, and Fernando Latorre for their work in the simulation tool and the reviewers for their helpful comments.

11. REFERENCES

[1] IA-32 intel architecture software developer's manual. volume 3: System programming guide.

[2] 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. of the 30th ISCA*, pages 350–361, 2003.

[3] F. Cazorla, E. Fernandez, A. Ramirez, and M. Valero. Dynamically controlled resource allocation in smt processors. *Proc. of the 37th Annual ACM/IEEE Intl. Symposium on Microarchitecture*, pages 171–182, 2004.

[4] F. Cazorla, P. Knijnenburg, E. Fernandez, R. Sakellariou, A. Ramirez, and M. Valero. Implicit vs. explicit resource allocation in SMT processors. *In Symp. on Digital System Design. Invited Paper*, 2004.

[5] F. Cazorla, P. Knijnenburg, E. Fernandez, R. Sakellariou, A. Ramirez, and M. Valero. Predictable performance in SMT processors. *ACM International Conference on Computing Frontiers*, pages 171–182, 2004.

[6] F. Cazorla, P. Knijnenburg, E. Fernandez, R. Sakellariou, A. Ramirez, and M. Valero. Qos for high-performance SMT processors in embedded systems. *IEEE micro. Special Issue on Embedded Systems*, 24(4):24–31, July/August 2004.

[7] C. J. Hughes, P. Kaul, S. V. Adve, R. Jain, C. Park, and J. Srinivasan. Variability in the execution of multimedia applications and implications for architecture. *Proc. of the 28th Annual ISCA*, pages 254–265, 2001.

[8] R. Jain, C. Hughes, and S. Adve. Soft real-time scheduling on simultaneous multithreaded processors. *Proc. of the 23th International Symposium on Real-Time Systems Symposium*, pages 134–145, 2002.

[9] M. Levy. Multithreaded technologies discolsed at MPF. *Microprocessor Report*, Nov 2003.

[10] A. Snavely, D. Tullsen, and G. Voelker. Symbiotic job scheduling with priorities for a simultaneous multithreaded processor. *ACM SIGMETRICS*, pages 234–244, June 2002.

[11] 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. of the 23th Annual ISCA*, Apr. 1996.

[12] D. W. Wall. Limits of instruction-level parallelism. *Proc. of the 4th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, April 1991.