Implicit vs. Explicit Resource Allocation in SMT Processors

Francisco J. Cazorla[1], Peter M.W. Knijnenburg[2], Rizos Sakellariou[3], Enrique Fernandez[4], Alex Ramirez[1], Mateo Valero[1]

[1]DAC, UPC, Spain, {fcazorla, aramirez, mateo}@ac.upc.es
[2]LIACS, Leiden University, the Netherlands, peterk@liacs.nl
[3]University of Manchester, UK, rizos@cs.man.ac.uk
[4]University of Las Palmas Gran Canaria, Spain, efernandez@dis.ulpgc.es

Abstract

In a Simultaneous Multithreaded (SMT) architecture, the front end of a superscalar is adapted in order to be able to fetch from several threads while the back end is shared among the threads.

In this paper, we describe different resource sharing models in SMT processors. We show that explicit resource allocation can improve SMT performance. In addition, it enables SMTs to solve other QoS requirements, not realizable before.

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. Therefore, many hardware resources are used to exploit this limited amount of ILP, 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 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 [1]. In Simultaneous Multithreaded (SMT) architectures, first introduced in [10][22][23], several threads are running together, sharing resources at the micro-architectural level. This allows an SMT to increase throughput with a moderate area overhead over a superscalar processor [2][12][17]. In an SMT, the front end of a superscalar is adapted in order to be able to fetch from several threads while the back end is shared among the threads. A *fetch policy*, e.g., *icount* [21], decides how instructions are fetched from the threads, thereby implicitly determining the way internal processor resources, like rename registers or IQ entries, are allocated to the threads. The common characteristic of many existing fetch policies is that they attempt to increase throughput and/or fairness [16] by stalling or flushing threads experiencing L2 misses [3][7][15][20], or reduce the effects of misspeculation by stalling on hard-to-predict branches [13]. Current trends in processor architecture show that many future microprocessors will have some form of SMT. Examples of SMTs include the Pentium4 [17] and the Power5 [12], and embedded processors like the META [14].

However, in current SMT organizations, threads not only share resources but compete for them. This competition can hurt performance since threads may be allocated resources that they cannot actually use for a long time, while other threads that could use these resources are starved for them. As an example, consider that a thread experiences an L2 cache miss. This miss may take hundreds of cycles to resolve. During this time, the missing load instruction and all instructions dependent on this load cannot make any progress. Nevertheless, they occupy IQ entries and hold physical registers. In current SMT organizations, during the time that the missing load is serviced, the offending thread can bring more instructions into the pipeline. Hence, the offending thread occupies more and more resources that cannot be used by other threads, degrading the situation fast. It is clear that this situation degrades SMT performance.

The reason that this can happen is that in current SMT organizations, there is no explicit resource allocation. It is the instruction fetch policy that decides from which threads instructions are brought into the processor. In this sense, the fetch policy acts like a

Proceedings of the EUROMICRO Systems on Digital System Design (DSD'04)
0-7695-2203-3/04 $ 20.00 IEEE

IEEE COMPUTER SOCIETY

hardware scheduler that works independently from the software scheduler inside the Operating System. This means that the OS level scheduler can have negative interference with the hardware scheduler inside the processor. Once instructions are inside, they are assigned resources blindly. The example above shows that this does not always lead to the best results. Thus, it is clear that in order to exploit the available resources in an SMT processor maximally, implicit resource allocation by means of the instruction fetch policy is not adequate and that a more explicit resource allocation scheme is required.

In this paper, we describe different resource sharing approaches in SMT processors. We show that if explicit resource allocation is used, the performance of SMT processors may improve. In particular, we show two different mechanisms that use explicit resource allocation, namely, a mechanism that dynamically distributes resources over threads in order to maximize throughput and fairness, and predictable performance for a thread in a workload, formulated as a notion of Quality of Service for SMT processors.

This paper is structured as follows. Section 2 discusses briefly our experimental setup. Section 3 discusses a resource allocation policy to improve throughput and fairness. Section 4 discusses how to obtain predictable performance for a designated thread by means of resource allocation.These last two sections also include discussions of related work and comparisons of these papers with ours. We draw some conclusions in Section 5.

2. Experimental Setup

We use a standard 4-context SMT configuration. There are 6 integer, 3 FP, and 4 load/store functional units and 32-entry integer, load/store, and FP IQs. There are 320 physical registers shared by all threads. Each thread has its own 256-entry reorder buffer. We use separate 64K, 2-way data and instruction caches and a unified 512KB 8-way L2 cache. The latency from L2 to L1 is 10 cycles, and from memory to L2 100 cycles. We use an improved version of the SMTSIM simulator [21]. We run 300 million most representative instructions for each benchmark. We consider workloads of 2, 3, and 4 threads that are of two different types: threads that exhibit a high number of L2 misses of over 1% of the dynamic load instructions, called *Memory Bounded* (MEM) threads. These threads have a low full speed. Secondly, threads that exhibit good memory behavior and have a high full speed, called *ILP threads.* We consider workloads in which all threads are ILP, all threads are MEM, or where there is a MIX of these types of threads.

3. Increasing Throughput and Fairness

In this section, we explain a mechanism that uses explicit resource allocation to improve throughput and fairness in SMT processors. By fairness we mean that the slowdown a thread experiences due to the fact that it is executed in a workload on an SMT, is the same for each thread in that workload [16].

3.1. Related Work

Current SMT processor proposals use either static resource allocation or fully flexible resource distribution. The static sharing model [9][17][18] evenly splits critical resources (registers and instruction queues) among all threads, ensuring that no single thread monopolizes a critical resource, causing other threads to wait for resources. However, this method lacks flexibility and can cause many resources to remain idle when one thread has no need for them, while the other threads could benefit from additional resources.

An alternative to static partitioning of resources is to have a common pool of resources that is shared among all threads. In this environment, the fetch policy determines how resources are shared, as it decides which threads enter the processor and which are left out.

Round Robin [21] is the most basic form of fetch and simply fetches instructions from all threads alternatively, disregarding the resource use of each thread.

Icount [21] prioritizes threads with few instructions in the pre-issue stages and presents good results for threads with high ILP. However, an SMT has difficulties with threads with high L2 miss rate. When this situation happens, *icount* does not realize that a thread can be blocked and does not make progress for many cycles. As a result, shared resources can be monopolized for a long time. This situation is more acute for fetch mechanisms that fetch instructions from up to two threads each cycle. If we use high performance fetch mechanisms like [8], that provides as high performance as previous ones, this situation is reduced.

Stall [20] is built on top of *icount* to avoid the problems caused by threads with a high cache miss rate. It detects that a thread has a pending L2 miss and prevents the thread from fetching further instructions to avoid resource abuse. However, L2 miss detection already may be too late to prevent a thread from occupying most of the available resources. Furthermore, it is possible that the resources allocated to a thread are not

IEEE COMPUTER SOCIETY

required by any other thread, and so the thread could very well continue fetching instead of stalling, producing resource under-use.

Flush [20] is an extension of *stall* that tries to correct the case in which an L2 miss is detected too late by deallocating all the resources of the offending thread, making them available to the other executing threads. However, it is still possible that the missing thread is being punished without reason, as the deallocated resources may not be used (or fully used) by the other threads. Furthermore, by flushing all instructions from the missing thread, a vast amount of extra fetch and power is required to redo the work for that thread.

Flush++ [3] based on the idea that *stall* performs better than *flush* for workloads that do not put a high pressure on resources, that is, workloads with few threads that have high L2 miss rate. Conversely, *flush* performs better when a workload has threads that often miss in the L2 cache, and hence the pressure on the resources is high. *Flush++* combines *flush* and *stall*: it uses cache behavior of threads to switch among *flush* and *stall* in order to provide better performance.

Data Gating [7] is a policy that attempts to reduce the effects of loads missing in the L1 data cache by stalling threads on each L1 data miss. However, not all L1 misses cause an L2 miss. We have measured that for memory bounded threads less than 50% of L1 misses cause an L2 miss. Thus, to stall a thread every time it experiences an L1 miss may be too severe.

Predictive Data Gating [7] and *DC-PRED* [15] work like *stall*, that is, they prevent a thread from fetching instructions as soon as a cache miss is predicted. By using a miss predictor, they avoid detecting the cache miss too late, but they introduce yet another level of speculation in the processor and may still be saving resources that no other thread will use. Furthermore, cache misses prove to be very hard to predict accurately [24], reducing the advantage of these techniques.

DWarn [4] adapts its behavior to the number of running threads. If there are few running threads, it stalls delinquent threads with L1 misses. When the number of threads increases, it reduces resource under-utilization by only reducing the fetch priority of delinquent threads, instead of stalling or flushing them.

3.2. Explicit Resource Allocation

Current fetch policies do not exercise direct control over how resources are distributed among threads. They use only indirect indicators of potential resource abuse by a given thread, for example, after L2 cache misses. Because no direct control over resources is exercised, it is still possible that a thread allocates most of the processor resources, causing other threads to stall. Also, to make things worse, it is a common situation that the thread which was allocated most of the resources will not release them for a long period of time. There are fetch policies in the literature [7][15][20] that try to detect this situation in order to prevent it by stalling the thread before it is too late, or even to correct the situation by squashing the offending thread to make its resources available to other threads, with varying degrees of success. The main problem of these policies is that in their attempt to prevent resource monopolization they may introduce resource under-use, because they are preventing a thread from using a set of resources that no other thread requires.

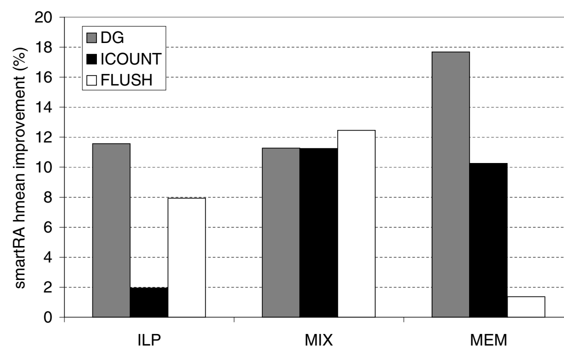


Figure 1. Hmean improvement of smartRA over various I-fetch policies

In [5], we show that the performance of an SMT processor can significantly be improved if a direct control on resource allocation is exercised. At any given time, threads must be forced to use a limited amount of resources. Otherwise, they could monopolize shared resources. In order to control the amount of resources given to each thread, we introduce the concept of *resource allocation policy*. A resource allocation policy controls the fetch slots, as instruction fetch policies do, but in addition it exercises a direct control over *all* shared resources. This direct control allows a better use of resources, reducing resource under-utilization. The main idea behind a smart resource allocation policy is that each program has very different resource demands. Even more, a given program has different resource demands during its execution. We show that the better we identify these demands and adapt resource allocation to them, the higher the performance of the SMT processor gets [5].

Fairness results using the Harmonic Mean as a metric, shown in Figure 1, show that our smart Resource Allocation (smartRA) mechanism improves all other

IEEE COMPUTER SOCIETY

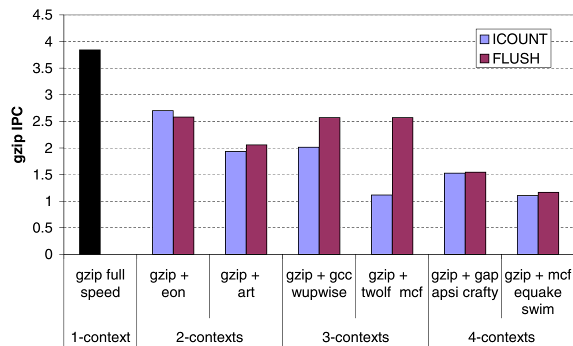


Figure 2. IPC of `gzip` for different contexts and different fetch policies

policies for all workload types. This indicates that smartRA is more fair than the other policies. This is caused by the fact that previously proposed policies favor ILP threads at the cost of degrading MEM threads. SmartRA proceeds the other way around by helping threads in *slow* phases as they do not harm the performance of threads in ILP phases. As a result, smartRA improves *flush* by 7.3%, DG by 13.5% and *icount* by 7.8%, on average in fairness.

From the fetch policies, *flush* achieves the best results. On average, smartRA improves *flush* by 2% in throughput and 7% in fairness. In addition to this, smartRA has another advantage over *flush*: it does not require to squash instructions in the pipeline. The amount of instructions that needs be refetched by *flush* is significant: up to 87% for the workloads considered in our work [5].

4. Predictable Performance

Approaches currently employed for resource sharing in SMTs may disagree with targets 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 *co-scheduling* phase, that is, the construction of the workload.

A problem with all the fetch policies with implicit resource allocation is that it is unpredictable what the performance of a certain thread in a workload actually is. Figure 2 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* [21] and *flush* [20]. As we can see, its IPC varies much, depending on the fetch policy as well as characteristics of the other threads running in the context. For instance, in a 3 thread context, its IPC can be higher than in a 2 thread context. This is caused by the fact that management of resources (IQ entries, registers, FUs) is not explicit. Currently, there is no fetch policy that can enforce that resources are allocated to a particular thread in such a way that this thread would perform similarly regardless its context.

4.1. Related work

To the best of our knowledge, only few papers have identified the need for a closer interaction between SMT co-scheduling and resource sharing algorithms. However, these papers are concerned with job prioritization which is a different problem than the one addressed in the present paper which is concerned with guaranteeing that particular jobs achieve a required throughput. In [11], the authors focus on the exploration of co-scheduling using representative resource sharing policies. It is recognized that in SMTs there is a problem of predicting the performance of a job, for instance to guarantee a certain performance requirement. It is suggested that, when necessary, this problem is bypassed by statically assigning a fixed number of resources. Besides the fact that such a static allocation may be under-utilizing the system, it is not clear what the appropriate static allocation would be if we had to achieve a certain percentage of the maximum speed. In Figure 3(a), we show the IPC of the `gzip` benchmark when it is statically assigned a percentage of the hardware resources. We see that, for the same partition of resources, `gzip` achieves different IPC values, with a variation of up to 36% depending on the workload.

In [19], several OS level job schedulers are proposed to enforce priorities. The *SOS* policy runs jobs alone on the machine to determine their full speed, runs several job mixes in order to determine the best mix that exhibits symbiosis, and finally runs jobs alone in order to meet priorities. This approach may under-utilize the machine resources since jobs need to be run alone several frames. Next, they propose an extension to the *icount* fetch policy 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 they are able to prioritize threads, running times of jobs are still hard to predict, rendering this approach unsuited for real-time constraints. For example, in Figure 3(b) we show the IPC of `gzip` for different handicaps and workloads. We observe that running `gzip` with the same handicap value leads to different IPC values, with a variation of

IEEE COMPUTER SOCIETY

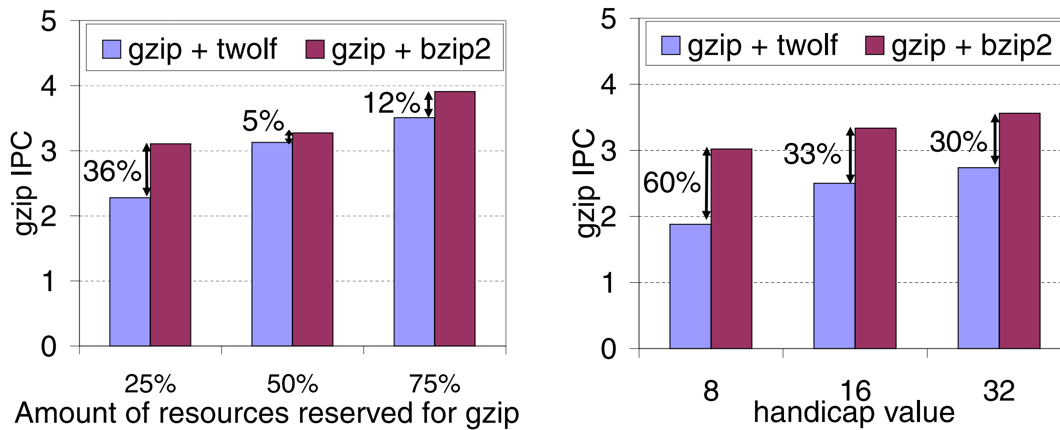
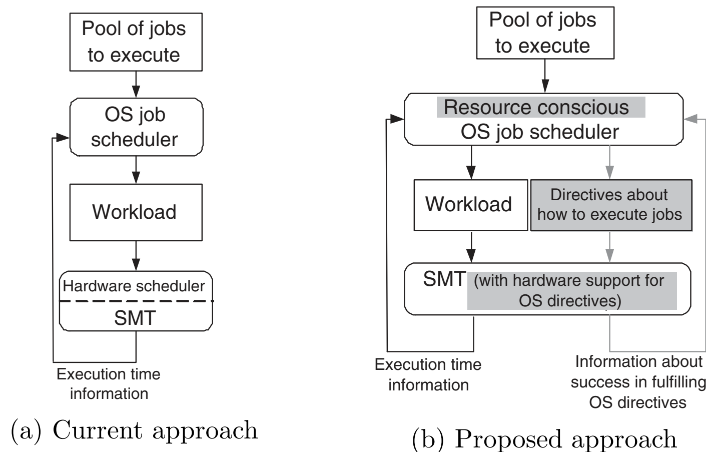


Figure 3. IPC of `gzip` for different: (a) static resource divisions; (b) handicap values



(a) Current approach

(b) Proposed approach

Figure 4. Two views of OS/SMT collaboration

up to 60% depending on the workload. Hence, these policies achieve some type of prioritization of threads. However, they are still far from the QoS requirement of this section, namely, guaranteeing a certain throughput for particular jobs in the workload.

4.2. Our approach

As opposed to traditional monolithic approaches where the OS has no control over the resource sharing of an SMT architecture (see Figure 4(a)), our approach is based on a resource sharing policy that is continuously adapted as it needs to take into account OS requirements (see Figure 4(b)).

To tackle this challenge, we propose a generic approach to resource sharing for SMTs formulated 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 fine tune 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 propose to use 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

IEEE COMPUTER SOCIETY

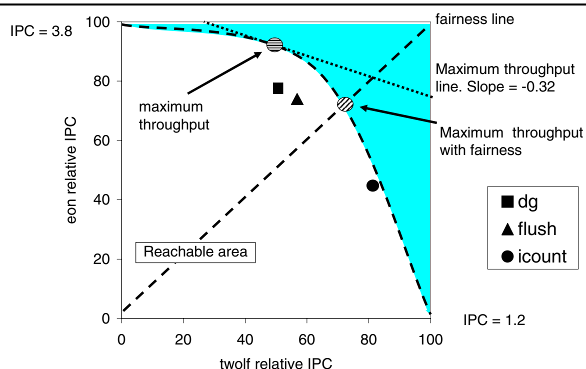


Figure 5. QoS space for three fetch policies and important QoS points and areas

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*. For example, Figure 5 shows the QoS space for two threads, `eon` and `twolf`, as could be obtained for the Pentium4 [17] or the Power5 [12]. In this figure, both x- and y-axis span from 0 to 100%. We have used three fetch policies: *icount* [21], *flush* [20], and data gating (*dg*) [7] 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. Hence, this shaded area is called the *reachable part* of the space for the given fetch policies. Figure 5 also shows a more general picture in which the dashed curve indicates points that intuitively could be reached using some fetch and resource allocation policy. Obviously, by assigning all fetch slots and resources to one thread, we reach 100% of its full speed. Conversely, it is impossible to reach 100% of the speed of each application at the same time since they have to share resources.

In Figure 5 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 weighted speedup, as defined in [19], lie on the same line that is perpendicular to the bottom-left top-right diagonal. 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%, 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. Note that the notion of QoS space may be applicable to other quantities, not only percentage of IPC.

To implement such levers, we consider the SMT as having a collection of sharable resources and add mechanisms to control how these resources are actually shared. These mechanisms include prioritizing instruction fetch for particular threads, reserving parts of the resources like instruction or load/store queue entries, prioritizing issue, etc. The OS, knowing the needs of applications, can exploit these levers to navigate through the QoS space. This solution should be parameterized and maximize the reachable part of the space so that it is generally usable and provides opportunities for fine tuning the machine for arbitrary workloads and QoS requirements.

In order to satisfy a wide range of varying QoS requirements, it is essential to have policies that return points that maximize the reachable part of the space. This means that we need to find policies that can sacrifice some IPC from one application for a better IPC for another. Whether this trade-off is acceptable depends on the particular circumstances. What is important is that, instead of considering maximum throughput or fairness as the ultimate objective, we want to provide as much flexibility as possible.

4.3. Explicit resource allocation

In [6] we move a step further than existing work and we propose a novel approach for a *dynamic* interaction between the OS and the processor which allows the former to pass specific requests onto the latter. In particular, we focus on the following challenge: given a workload of N applications[1] and a Predictable Performance Thread (PPT) in this workload, find a resource sharing policy to accomplish with these to objectives.

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

IEEE COMPUTER SOCIETY

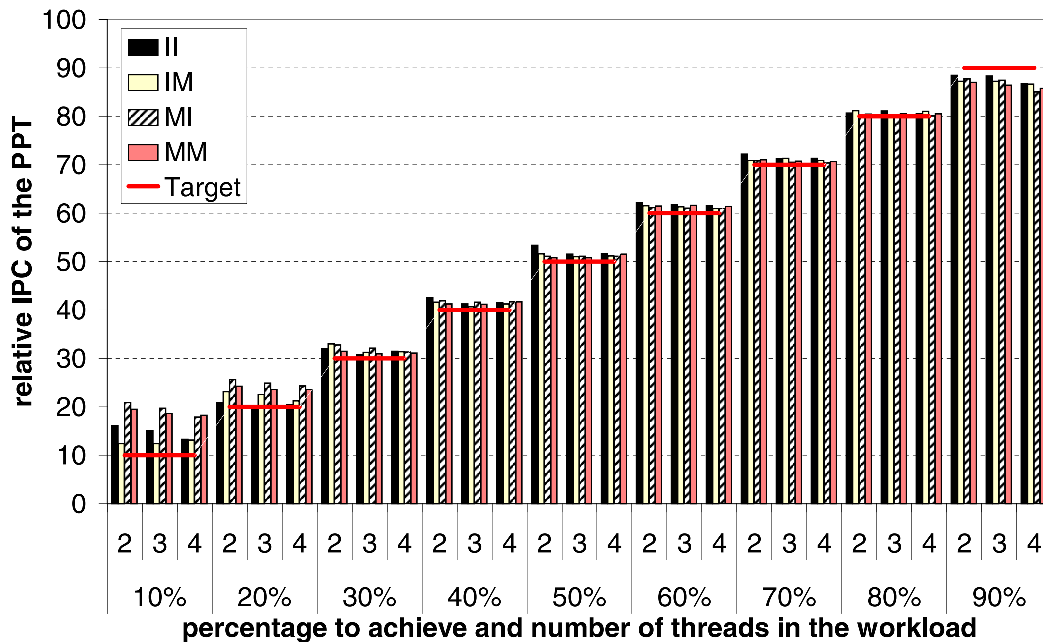


Figure 6. Realized IPC values for the PPT. The x-axis shows the target percentage of full speed of the PPT and size of the workloads. The four different bars represent the four different types of workload

- Ensure that PPT runs at (at least) a given target IPC that represents $X\%$ of the IPC it would get if it were executed alone on the machine.
- Maximize the throughput for the remaining Unpredictable Performance Threads (UPTs) in the workload.

In [6] we show that it is possible to solve this challenge. The basis of our mechanism rests on the observation that in order to realize $X\%$ of the overall IPC for a given job, it is sufficient to realize $X\%$ of the maximum possible IPC *at every instant* through the execution of that job. We employ two phases, that are executed in alternate fashion.

- During the *sample phase*, all shared resources are given to the PPT and UPTs are temporarily stopped. As a result, we obtain an estimate of the full speed of the PPT during this phase, called the *local IPC*. In order to counteract thread interference, this phase is divided in a *warmup phase* of 50,000 cycles and an *actual sample phase* of 10,000 cycles.
- During the *tune phase*, our mechanism dynamically varies the amount of resources given to the PPT in order to achieve a *target IPC* that is given by the local IPC computed in the last sample period times the required percentage given by the OS. The resources considered in this paper are rename registers, instruction and load/store queue entries, and ways in the 8-way set associative L2 cache. We tune resource allocation every 15,000 cycles for a period of 1.2 million cycles.

In Figure 6 we show for target percentages ranging from 10 to 90% the percentage of the full speed of the PPT that we achieve. We have used four different types of workload. We have considered the cases where the PPT is a high speed ILP thread or a slow memory bounded thread. The UPTs can also be of this nature. We have considered workloads of 2, 3, and 4 threads. The bars in the figure give averages over these number of threads. We see that we reach the required percentage exactly for target percentages of 20 to 80%. Only for the border cases, 10% and 90%, we are somewhat off target.

In [6] we have also shown that the throughput of the UPTs remains significant. In fact, for a number of target percentages of the PPT and a number of workloads, throughput of the UPTs exceeds their throughput under *icount* or *flush*. Even for high target percentages of the PPT, the throughput of the UPTs is not destroyed. This shows that in our mechanism the UPTs can use whatever spare resource that is left by the PPT.

IEEE COMPUTER SOCIETY

5. Conclusions

Current SMTs implicitly share resources among threads, which means that the OS is blind to this assignment of resources. However, to provide QoS for SMT processors, explicit resource allocation is required. We have presented two examples of QoS requirements where the OS can benefit from explicit resource allocation. Our results show that explicit resource allocation can improve the performance of SMT processors and moreover enable them to solve QoS requirements not realizable before.

Acknowledgments

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

References

[1] D.C. Bossen, J.M. Tendler, and K. Reick. Power4 system design for high reliability. *IEEE Micro*, 22(2):16–24, 2002.

[2] J. Burns and J-L. Gaudiot. Quantifying the SMT layout overhead- does SMT pull its weight? In *Proc. HPCA-6*, pages 109–120, 2000.

[3] F. J. Cazorla, E. Fernandez, A. Ramirez, and M. Valero. Improving memory latency aware fetch policies for SMT processors. In *Proc. ISHPC-5*, pages 70–85, October 2003.

[4] F. J. Cazorla, E. Fernandez, A. Ramirez, and M. Valero. DCache Warn: an I-Fetch policy to increase SMT efficiency. In *Proc. IPDPS*, 2004.

[5] F.J. Cazorla, E. Fernandez, A. Ramirez, and M. Valero. Approaching a smart sharing of resources in SMT processors. In *Proc. WCED*, 2004.

[6] F.J. Cazorla, P.M.W. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero. Predictable performance in SMT processors. In *Proc. Computing Frontiers*, 2004.

[7] A. El-Moursy and D.H. Albonesi. Front-end policies for improved issue efficiency in SMT processors. In *Proc. HPCA-9*, pages 31–42, 2003.

[8] Ayose Falcon, Alex Ramirez, and Mateo Valero. A low complexity, high-performance fetch unit for simultaneous multithreading processors. *Proceedings of the 12th Intl. Conference on High Performance Computer Architecture*, pages 244–253, February 2004.

[9] R. Goncalves, E. Ayguade, M. Valero, and P. O. A. Navaux. Performance evaluation of decoding and dispatching stages in simultaneous multithreaded architectures. In *Proc. Computer Architecture and High Performance Computing*, 2001.

[10] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *Proc. ISCA-19*, pages 136–145, 1992.

[11] R. Jain, C.J. Hughes, and S.V. Adve. Soft real-time scheduling on simultaneous multithreaded processors. In *Proc. RTSS-23*, pages 134–145, 2002.

[12] R. Kalla, B. Sinharoy, and J. Tendler. IBM Power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, 2004.

[13] P.M.W. Knijnenburg, A. Ramirez, J. Larriba, and M. Valero. Branch classification for SMT fetch gating. In *MTEAC-6*, pages 49–56, 2002.

[14] M. Levy. Multithreaded technologies disclosed at MPF. *Microprocessor Report*, November 2003.

[15] C. Limousin, J. Sebot, A. Vartanian, and N. Drach-Temam. Improving 3D geometry transformations on a simultaneous multithreaded SIMD processor. In *Proc. ICS-15*, pages 236–245, 2001.

[16] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. In *Proc. ISPASS*, pages 164–171, 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 Journal*, 6(1), February 2002.

[18] S. E. Raasch and S. K. Reinhardt. The impact of resource partitioning on SMT processors. In *Proc. PACT-12*, pages 15–25, 2003.

[19] A. Snavely, D.M. Tullsen, and G. Voelker. Symbiotic job scheduling with priorities for a simultaneous multithreaded processor. In *Proc. ASPLOS-9*, pages 234–244, 2000.

[20] D. Tullsen and J. Brown. Handling long-latency loads in a simultaneous multithreaded processor. In *Proc. MICRO-34*, pages 318–327, 2001.

[21] 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. In *Proc. ISCA-23*, pages 191–202, 1996.

[22] D.M. Tullsen, S. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proc. ISCA*, pages 392–403, 1995.

[23] W. Yamamoto and M. Nemirovsky. Increasing superscalar performance through multistreaming. In *Proc. PACT-4*, pages 49–58, 1995.

[24] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. Speculation techniques for improving load related instruction scheduling. In *Proc. ISCA-26*, May 1999.

IEEE COMPUTER SOCIETY