Feasibility of QoS for SMT

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

[1] Universidad Politécnica de Cataluña
{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] Universidad de Las Palmas de Gran Canaria
efernandez@dis.ulpgc.es

Abstract. Since embedded systems require ever more compute power, SMT processors are viable candidates for future high performance embedded processors. However, SMTs exhibit unpredictable performance due to uncontrolled interaction of threads. Hence, the SMT hardware needs to be adapted in order to meet (soft) real time constraints. We show by a simple policy that the OS can exercise control over the execution of a thread which is required for real time constraints.

1 Introduction

To deal with real time constraints, current embedded processors are usually simple in-order processors with no speculation capabilities. However, embedded systems are required to host more and more complex applications and have higher and higher data throughput rates. Therefore, future embedded processors will resemble current high performance processors. Simultaneous Multithreaded (SMT) architectures [5][6] are viable candidates for future high performance embedded processors, because of their good cost/performance trade-off [2]. In an SMT, several threads are running together, sharing resources at the microarchitectural level, in order to increase throughput. A *fetch policy* decides from which threads instructions are fetched, thereby implicitly determining the way processor resources, like rename registers or IQ entries, are allocated to the threads. However, with current policies the performance of a thread in a workload is unpredictable. This poses problems for the suitability of SMT processors in the context of (soft) real-time systems.

The key issue is that in the traditional collaboration between OS and SMT, the OS only assembles the workload while it is the processor that decides how to execute this workload, implicitly by means of its fetch policy. Hence, part of the traditional responsibility of the OS has "disappeared" into the processor. One consequence is that the OS may not be able to guarantee time constraints even though the processor has sufficient resources to do so. To deal with this situation, the OS should be able to exercise more control over how threads are executed and how they share the processor's internal resources.

In this paper, we discuss our philosophy behind a novel collaboration between OS and SMT in which the SMT processor provides 'levers' through which the OS can fine tune the internal operation of the processor to achieve certain requirements. We want to reserve resources inside the SMT processor in order to guarantee certain requirements for executing a workload. We show the feasibility of this approach by a simple parameterized mechanism that assigns fetch slots and instruction and load queue entries to a High Priority Thread. This, in turn, is a first step toward enabling the OS to execute a thread at a given percentage of its full speed and thus enabling the use of out-of-order, high performance SMT processor in embedded environments.

This paper is structured as follows. In Section 2 we describe our novel approach to the collaboration between OS and SMT. In Section 3 we discuss a simple mechanism to enable such collaboration. We discuss related work in Section 4. Finally, Section 5 is devoted to conclusions and future directions.

2 QoS: A Novel Collaboration Between OS and SMT

In this paper, we approach OS/SMT collaboration as *Quality of Service (QoS)* management. This approach has been inspired by QoS in networks. In an SMT resources can be reserved for threads guaranteeing a required performance. We observe that on an SMT processor, each thread reaches a certain percentage of the speed it would achieve when running alone on the machine. Hence, for a given workload consisting of N applications and a given instruction fetch policy, these fractions give rise to a point in an N-dimensional space, called the *QoS space*. For example, Figure 1(a) shows the QoS space for two threads, `eon` and `twolf`, as could be obtained for the Pentium4 or the Power5. In this figure, both x- and y-axis span from 0 to 100%. We have used two fetch policies: *icount* [5] and *flush++* [1]. Theoretically it is possible to reach any point in the shaded area below these points by judiciously inserting empty fetch cycles. Hence, this shaded area is called the *reachable part* of the space for the given fetch policies. In Figure 1(b), 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, that is, the speed it would reach when run alone. Conversely, it is impossible to reach 100% of the speed of each application at the same time since they have to share resources.

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 a QoS requirement and it is the responsibility of the processor to provide the levers to enable the OS to pose such requirements. To implement such levers, we add mechanisms to control how these

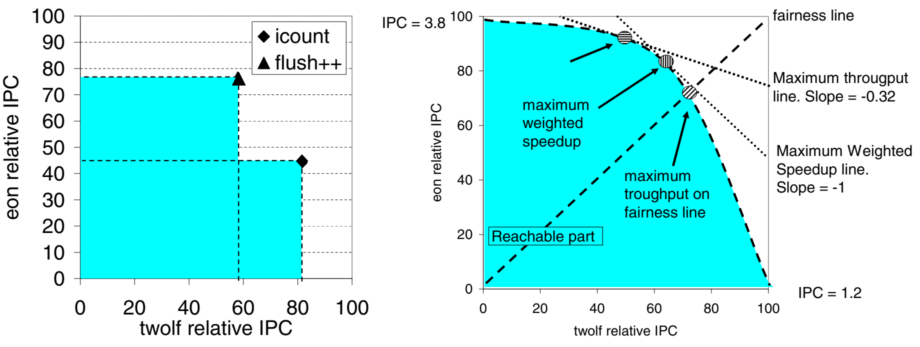

Fig. 1. (a) QoS space for three fetch policies; (b) important QoS points and areas.

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.

In this paper, we present a first step toward this goal by studying the behavior of threads when a certain number of resources is reserved for a High Priority Thread (HPT). We show that such a simple mechanism is already capable of influencing the relative speed of threads considerably and hence can cover the QoS space to a great extent.

3 QoS by Resource Allocation

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 a separate 32K, 4-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 provided by Tullsen [5]. We run 300 million most representative instructions for each benchmark. We consider workloads of 2 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* (MB) 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 4 workloads in which the High Priority Thread (HPT) is ILP or MB, and the Low Priority Thread (LPT) is ILP or MB. The workloads are: `gzip` and `bzip2` (ILP-ILP), `gzip` and `twolf` (ILP-MB), `twolf` and `bzip2` (MB-ILP), and `twolf` and `vpr` (MB-MB).

3.1 Static Resource Allocation

We statically reserve 0, 4, 8, . . . , 32 entries in the IQ and LSQ for the HPT. The remaining entries are devoted to the LPT. Moreover, we prioritize the instruction fetch and issue for the HPT: in each cycle, we first fetch/issue instructions from the HPT. If there are fetch opportunities left, then instructions from the LPT are fetched/issued. There are more resources in an SMT processor that are shared between threads, most notably the rename registers and the L1 and L2 caches. We have restricted attention to IQ and LSQ entries because these shared resources most directly determine which instructions from which thread are executed.

3.2 Results

We show the resulting QoS space for varying numbers of assigned resources in Figures 2(a) through 5(a). We also show the points obtained from the *round robin* (RR), *icount* (IC), and *flush* fetch policies for comparison. We immediately observe that our parameterized mechanism is capable of covering a large part of the reachable space by tuning its parameter. In contrast, the points reached by the three standard fetch policies show no coherent picture. For the ILP-ILP and MB-MB workloads, they reach points that are quite close together. For the other workloads, there is considerable difference and their relative position in the space changes. This shows that standard fetch policies provide little control over the execution of threads.

We show the resulting IPC values in Figure 2(b) through 5(b). We also show IPC obtained from the standard policies *icount* and *flush* for comparison.

ILP-ILP. Both threads have a high throughput and do not occupy IQs for a long time. As a result, reserving a number of these entries for the HPT and moreover prioritizing its fetch, quickly produces a situation in which the HPT dominates the processor and its speed comes close to its full speed. The total throughput is about the same as for *icount* (except the cases of 0 and 32) which means that what we take from one thread can successfully be used by the other.

ILP-MB. When the LPT thread misses in the L2, it tends to occupy resources for a long time which has an adverse effect on the speed of the other thread. Therefore, reserving resources for the HPT that is ILP causes its speed to sharply increase. As a result, the total throughput can be larger than for *icount*. *flush* needs to re-fetch and re-issue all flushed instructions, degrading its performance. The speed of the LPT does not degrade fast since it suffers many L2 misses and thus cannot use many resources.

MB-ILP. This case is the opposite to the previous one. Given that since the total throughput in comes largely from the LPT that is ILP, when it is denied many resources, its speed degrades fast and total throughput is degraded as well.

MB-MB. Throughput shows a flat curve that is about the same as for the *icount* and *flush* fetch policies as was the case for the ILP-ILP workload. Resources taken from one thread can effectively used by the other thread.

We conclude that by controlling resource allocation we can navigate through the QoS space and bias the execution of a workload to a prioritized thread. At the same time, we still reach considerable throughput for the LPT. Hence, our proposal to provide QoS in an SMT by means of resource allocation is a feasible approach.

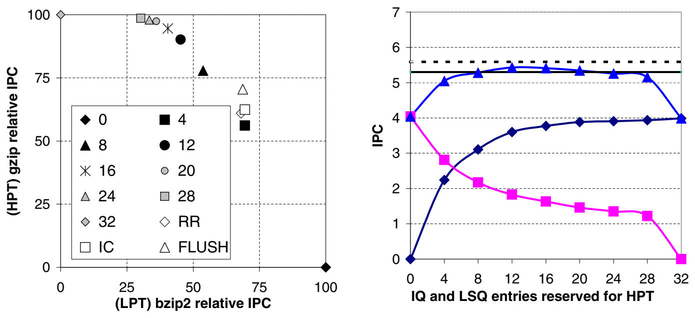

Fig. 2. (a) QoS space for ILP-ILP workload; (b) IPC values and overall throughput.

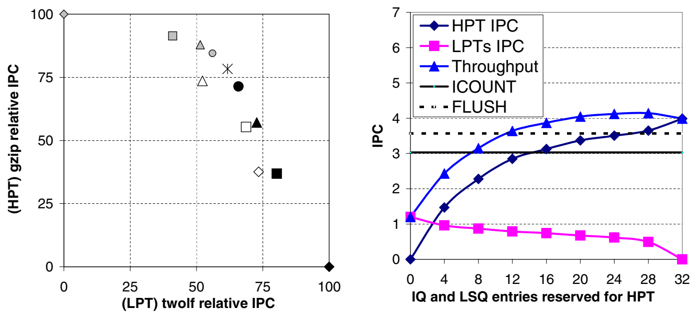

Fig. 3. (a) QoS space for ILP-MB workload; (b) IPC values and overall throughput.

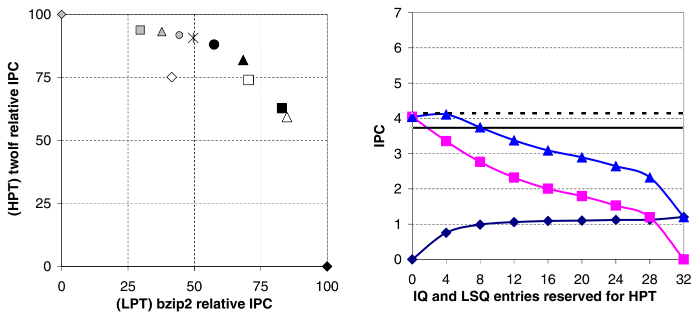

Fig. 4. (a) QoS space for MB-ILP workload; (b) IPC values and overall throughput.

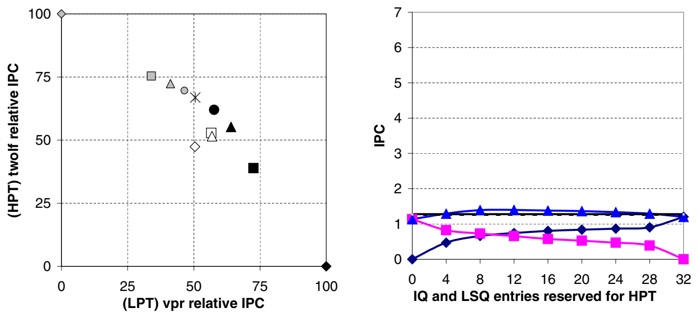

Fig. 5. (a) QoS space for MB-MB workload; (b) IPC values and overall throughput.

4 Related Work

To the best of our knowledge, there is not much work on real time constraints for SMTs. In [4] the authors consider a way of mapping OS-level priorities onto a modified *icount* policy that fetches depending on the priority of the threads. However, this approach exploits the fetch policy, obtaining very limited control, in contrast to our proposal. In [3] explicit static resource allocation is also studied and the authors conclude that resource allocation has little effect on throughput. However, they fail to recognize that the relative speed of threads does change significantly, which is precisely the property that we exploit to provide QoS.

5 Conclusions

In this paper, we have approached the collaboration between OS and SMT as Quality of Service management, where the SMT processor provides 'levers' through which the OS can fine tune the internal operation of the processor in order to meet certain QoS requirements, expressed as points or areas in the QoS space. We have shown, by evaluating a simple mechanism, that it is possible to influence to a great extend the execution of a thread in a workload, so that the OS can reach a large part of the QoS space. Hence, this mechanism is a first step toward enabling high-performance SMT processors to deal with real-time constraints and rendering them suitable for many types of embedded systems.

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 in the simulation tool.

References

1. F. J. Cazorla, E. Fernandez, A. Ramirez, and M. Valero. Improving memory latency aware fetch policies for SMT processors. In *Proc. ISHPC*, pages 70–85, 2003.
2. M. Levy. Multithreaded technologies disclosed at MPF. *Microprocessor Report*, November 2003.
3. S. E. Raasch and S. K. Reinhardt. The impact of resource partitioning on SMT processors. In *Proc. PACT*, pages 15–25, 2003.
4. A. Snavely, D.M. Tullsen, and G. Voelker. Symbiotic job scheduling with priorities for a simultaneous multithreaded processor. In *Proc. ASPLOS*, pages 234–244, 2000.
5. 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*, pages 191–202, 1996.
6. W. Yamamoto and M. Nemirovsky. Increasing superscalar performance through multistreaming. In *Proc. PACT*, pages 49–58, 1995.