QoS for High-Performance SMT Processors in Embedded Systems

ALTHOUGH SIMULTANEOUS MULTITHREADING PROCESSORS PROVIDE A GOOD COST-PERFORMANCE TRADEOFF, THEY EXHIBIT UNPREDICTABLE PERFORMANCE IN REAL-TIME APPLICATIONS. THE AUTHORS PRESENT A RESOURCE MANAGEMENT SCHEME THAT ELIMINATES A MAJOR CAUSE OF PERFORMANCE UNPREDICTABILITY IN SMTS, MAKING THEM SUITABLE FOR MANY TYPES OF EMBEDDED SYSTEMS.

Francisco J. Cazorla
Alex Ramirez
Mateo Valero
Polytechnic University of Catalonia

Peter M.W. Knijnenburg
Leiden University

Rizos Sakellariou
University of Manchester

Enrique Fernández
University of Las Palmas de Gran Canaria

Embedded systems have specific constraints and characteristics, such as real-time constraints, low-power requirements, and severe cost limitations that differentiate them from general-purpose systems. Processors for embedded systems typically are simple, with short pipelines and in-order execution. When they are used for real-time applications, they also lack unpredictable components, such as caches and branch predictors. These bare processors provide predictable performance, and hence they can guarantee worst-case execution times of real-time applications. However, embedded systems must host increasingly complex applications and have increasingly higher data throughput rates. To meet these growing demands, future embedded processors will resemble current high-performance processors. For example, the new Philips TriMedia already has a deep pipeline, L1 and L2 caches, and branch prediction.[1] But because of their unpredictable components, such processors have unpredictable execution times, so they are difficult to use in real-time applications.

Because embedded processors must be low in cost, obtaining as much performance as possible from each resource is desirable. Hence, a viable option is a simultaneous multithreading (SMT) processor, which shares many resources between several threads for a good cost-performance tradeoff.[2] An SMT design adapts a superscalar processor's front end to fetch from several threads, while the back end is shared. An instruction fetch policy decides from which threads to fetch instructions, thereby implicitly determining how internal processor resources, such as rename registers or instruction queue (IQ) entries, are allocated to threads. SMT processors have high throughput but, because of uncontrolled interference between threads, poor performance predictability—even worse than that of superscalar processors running only one thread. This poses problems for the suitability of high-performance SMT processors in real-time systems.

Other resource-sharing approaches include multiprocessors, which share only the higher

Published by the IEEE Computer Society

levels of the memory hierarchy. In multiprocessors, different threads are assigned to separate processors, so their performance is more predictable. However, since they share few resources between threads, their cost-performance ratio is worse than that of SMTs. Also, in a limited form of SMT, different threads share only a few resources, typically the instruction and data caches and the functional units, but each thread has its own instruction pipeline. An example is the Meta.[2] This solution has better performance predictability for the individual threads but underuses many resources, thus reducing total throughput. Hence, the cost-performance tradeoff is worse than that of a full-fledged SMT.

The key problem with using SMTs in embedded real-time systems is that in the traditional collaboration between the operating system (OS) and the SMT, the OS only assembles the workload, whereas the processor decides how to execute this workload. Hence, part of the OS's traditional responsibility has "disappeared" into the processor. Consequently, the OS cannot guarantee time constraints on the execution of a thread if that thread must run concurrently with other threads, even when the processor has sufficient resources to guarantee time constraints. To handle this situation, the SMT processor should be able to guarantee specific requirements set by the OS. This implies a tight interaction between the OS and the processor, so that the OS can exercise more control over how threads execute and how they share the processor's internal resources.

We propose a new collaboration in which the SMT processor provides "levers" with which the OS can fine-tune the processor's internal operation as needed to meet certain requirements. In particular, we propose a resource management mechanism to accomplish one such requirement: the ability to execute one thread in a workload at a given percentage of its full speed. This amounts to eliminating the SMT's performance unpredictability and thus simplifies the use of out-of-order, high-performance SMTs in embedded environments.

OS-SMT collaboration

Most approaches to thread prioritization focus on workload selection or provide limited control of performance predictability because a given thread's instructions per cycle (IPC) still depends on the workload the thread executes in.[3,4] In contrast, our new approach allows one thread to run at an arbitrary percentage of its full speed by dynamically allocating enough resources to accomplish this. In addition, the other threads in the workload receive many resources that the high-priority thread temporarily doesn't need and can reach significant speed.

The common characteristic of many current fetch policies is that they attempt to improve established metrics such as throughput or fairness.[5] There are several optimizations that improve on these basic policies, such as stalling, flushing, or reducing the fetch priority of threads experiencing L2 misses,[6-9] or reducing the effects of misspeculation by stalling on hard-to-predict branches.[10] However, the problem with all the fetch policies proposed so far is that the performance of a certain thread in a workload is unpredictable. For example, Figure 1 shows the IPC of the benchmark gzip when it runs alone (full speed) and when it runs with other threads using two different fetch policies, icount[11] and flush.[9] The icount policy first fetches from threads with the fewest instructions in the processor's front end. The flush policy stalls a thread upon an L2 miss and flushes the thread's instructions from the pipeline. Gzip's IPC varies considerably, depending on the fetch policy and characteristics of the other threads running in the workload. Thus, if we want to provide performance predictability on an SMT processor, current resource management approaches using instruction fetch policies are no longer adequate. Hence, we need a new paradigm for resource management in SMT processors.

In this article, we approach performance predictability as a quality of service requirement. The inspiration for this approach came from QoS in networks that gives processes guarantees about bandwidth, throughput, or other services. Analogously, SMT resources can be reserved for threads guaranteeing a required performance. In an SMT processor, each thread reaches a certain percentage of the speed it would achieve running alone on the machine. Hence, for a given workload consisting of N applications and a given instruction fetch

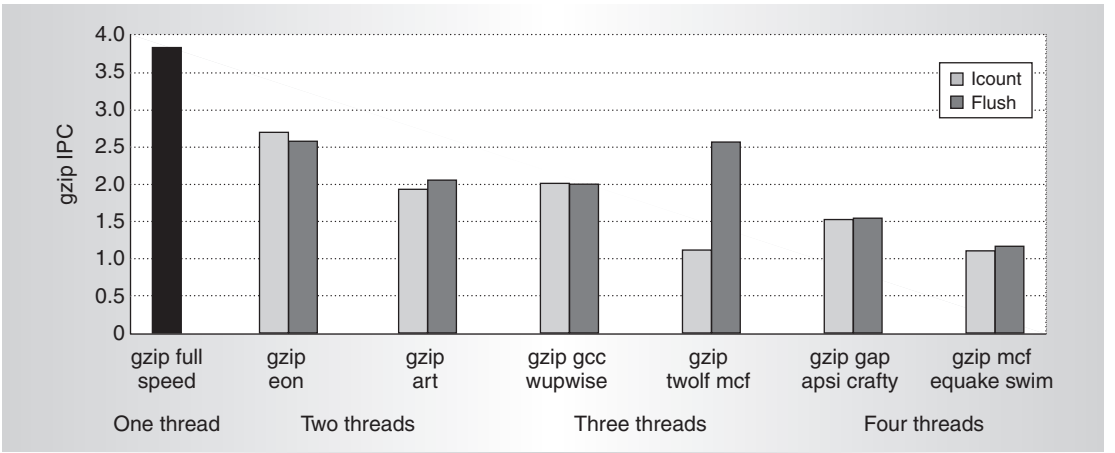

Figure 1. Instructions per cycle of gzip benchmark with different workloads and fetch policies.

policy, these fractions give rise to a point in an *N*-dimensional space that we call the QoS space.

For example, Figure 2a shows the QoS space for two threads, eon and twolf. In this figure, both the *x*- and *y*-axes span from 0 to 100 percent. We used two fetch policies: icount and flush++.[7] Theoretically, it is possible to reach any point in the shaded area below these points by judiciously inserting empty fetch cycles. Hence, we call this area the reachable part of the space for the given fetch policies. In Figure 2b, the dashed curve indicates points that intuitively we could reach using a fetch and resource allocation policy. Obviously, by assigning all fetch slots and resources to one thread, we reach 100 percent of its full speed. Conversely, it is impossible to reach 100 percent of each application's speed at the same time because the applications must share resources.

In Figure 2b, the QoS space representation also provides an easy way to visualize other metrics used in the literature. Points of equal weighted speedup[4] lie on the line perpendicular to the diagonal from bottom left to top right. The figure shows the point with maximum weighted speedup in the reachable part. Similarly, points of equal throughput lie on a single line whose slope is determined by the ratio of the full speed of each thread (in this case, –1.2/3.8 = –0.32). The figure indicates such a point with maximum throughput. Finally, points near the bottom left to top right diagonal indicate fairness, in the sense that each thread achieves the same proportion of its full speed. For all these cases, maximum values lie on the lines farthest from the origin.

Each point or area in the reachable part entails application execution properties: maximum throughput; fairness; real-time constraints; power requirements; a guarantee, say 70 percent, of a given thread's full speed; or any combination of these properties. In other words, each point or area in the space represents a solution to a QoS requirement. It is the OS's responsibility to select a workload and a QoS requirement, and it is the processor's responsibility to provide the levers that enable the OS to pose such requirements.

To implement the levers, we consider the SMT as having a collection of sharable resources and add mechanisms to control how the resources are shared. These mechanisms include prioritizing instruction fetch for particular threads, reserving parts of the resources such as instruction or load/store queue entries, prioritizing issue, and so forth. In our view, there should be a tight interaction between the OS and the processor. Figure 3 depicts this relationship. When levers are present, the OS, knowing the needs of applications, can exploit the levers to navigate through the QoS space. Parameterizing this solution makes it generally usable and enables it to provide opportunities for fine-tuning the machine for arbitrary workloads and QoS requirements. For example, we have shown elsewhere that directly controlling resource allocation can improve the total throughput and fairness of SMT processors.[6]

To satisfy a wide range of varying QoS requirements, it is essential that instruction fetch policies return points that maximize the

space's reachable part. This means that we must find policies that can sacrifice some IPC from one application for better IPC in another. Whether this tradeoff is acceptable depends on the circumstances. Rather than considering maximum throughput or fairness the ultimate objective, it is important that we provide as much flexibility as possible with a mechanism that dynamically adjusts resource allocation to achieve a given OS requirement. This mechanism reverses the procedure of current mechanisms that behave the same way throughout the execution of a program and define only a single point in the QoS space. The new mechanism is dynamic and attempts to converge to a point or area in the QoS space that represents a QoS requirement.

QoS through resource allocation

There are two ways to influence a thread's IPC. One is a static assignment of resources to threads. In contrast, we propose a novel dynamic mechanism that meets a QoS requirement that a specific job runs at a given percentage of its full speed—that is, of the IPC the job would have if it ran on the machine by itself. As an example, we show that we can achieve 70 percent of gzip's full speed as it runs in several workloads, while maximizing the throughput of the other threads in the workload.

Methodology

We assume a fairly standard, four-context SMT configuration. Our machine can fetch up to eight instructions from up to two threads each cycle. It has six integer, three floating-point, and four load/store functional units and 32-entry-deep integer, load/store, and floating-point IQs. There are 320 physical registers shared between all threads. Each thread has its own 256-entry reorder buffer. We use a two-level cache hierarchy with separate 32-Kbyte, four-way, set associative data and instruction caches and a unified 512-Kbyte, eight-way L2 cache. Latency from L1 to L2 is 10 cycles, and from L2 to memory 100 cycles. We use a trace-driven SMT simulator, based on the SMTsim simulator.[11] It consists of our own front end, which reads a trace file, and a modified version of SMTsim's back end.

We collected traces of the most representative 300-million-instruction segment of each

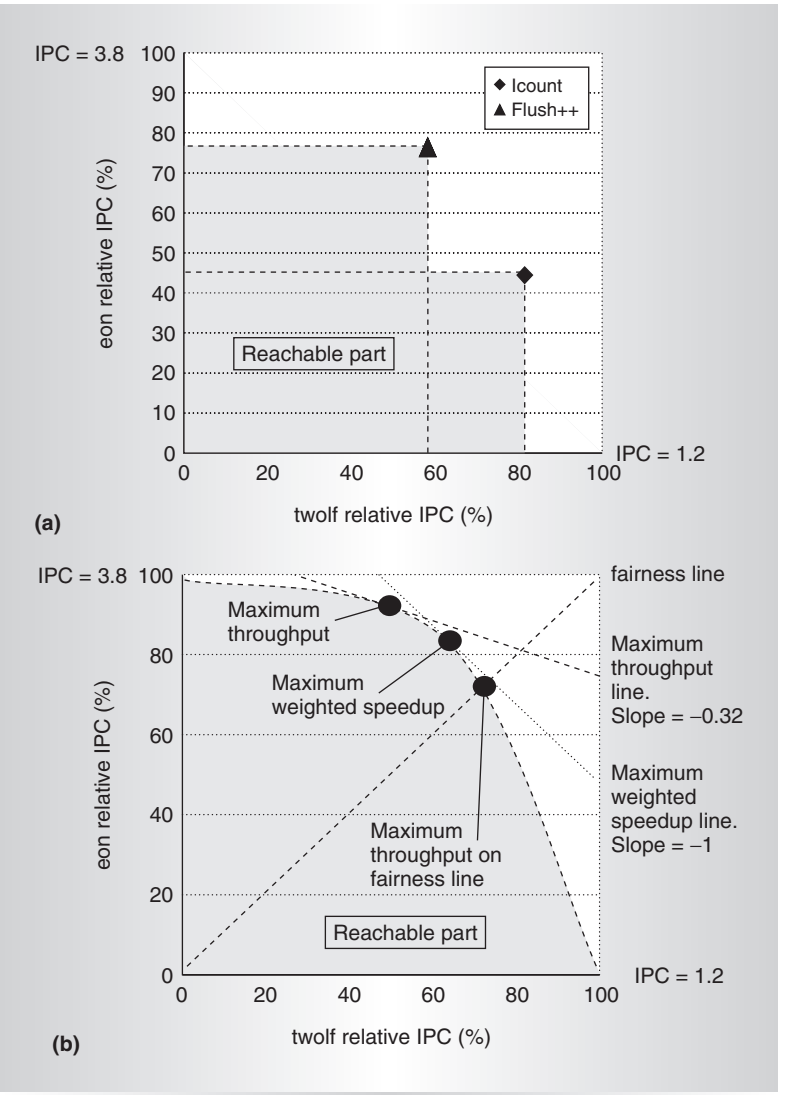

Figure 2. QoS space for three standard fetch policies (a); important QoS points and areas (b).

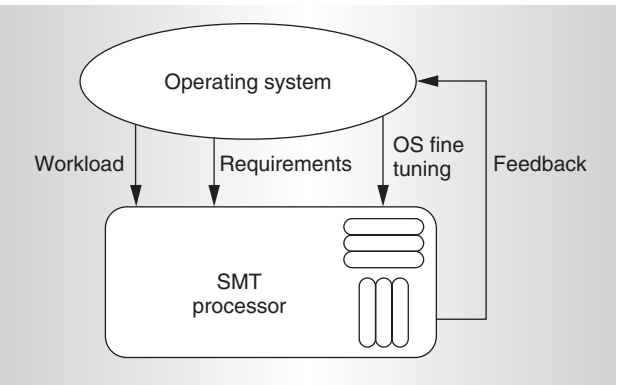

Figure 3. Interaction between OS and architecture to enforce QoS requirements.

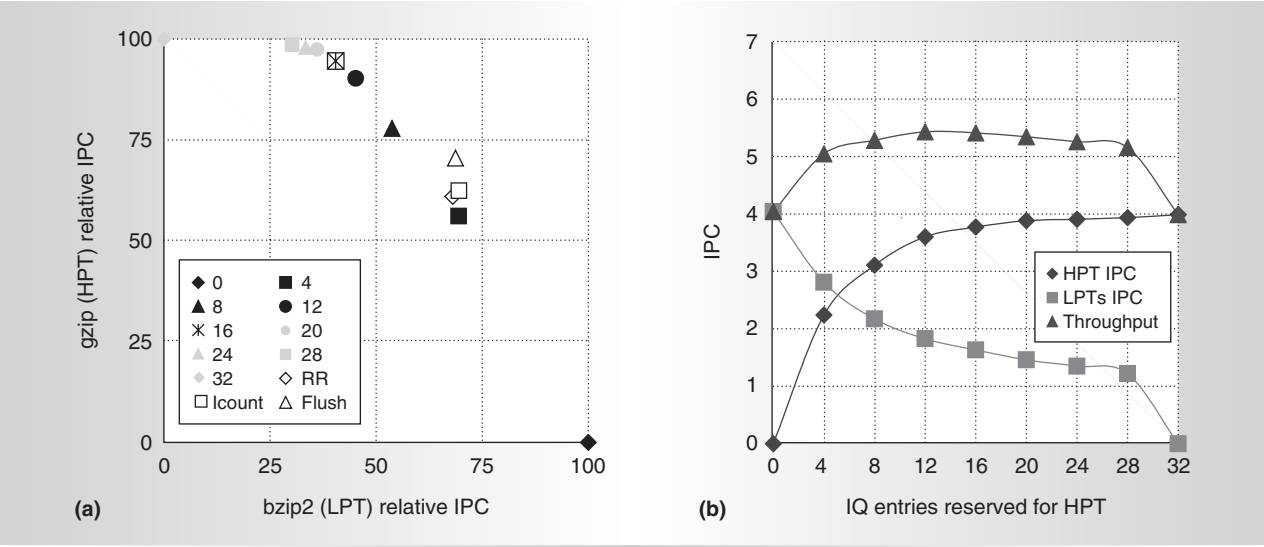

Figure 4. QoS space resulting from static resource allocation (a); IPC values and overall throughput for different resource allocations (b).

program, following an idea presented by Sherwood et al.[12] The workloads consisted of programs from the SPEC2000 integer and fp benchmark suites. We executed each program using the reference input set and compiled each program using the DEC Alpha AXP-21264 C/C++ and Fortran compilers using static libraries and a high level of optimization.

In our experiment, we considered workloads of two, three, and four threads, the same workloads we used in Figure 1. We considered two types of threads: memory-bounded (MB) and threads with high instruction-level parallelism (ILP). MB threads exhibit a high number of L2 misses and have a low full speed. ILP threads exhibit good memory behavior and have a high full speed. We focus on L2 cache behavior because it is the main source of unpredictability in SMT processors.[13] In particular, a MB thread tends to clog the pipeline after an L2 miss because many instructions are waiting for the miss to resolve. This can severely degrade other threads' performance because they cannot use resources occupied by the stalled MB thread. In contrast, instructions in ILP threads tend to issue and execute quickly, so they do not occupy resources for a long time. We always use the ILP thread gzip as the high-priority thread (HPT). The low-priority threads (LPTs) are either all ILP or all MB. They are denoted I*n* and M*n*, respectively, where *n* is the number of LPTs. For example, I1 denotes the workload consisting of gzip and eon, and M2 denotes the workload consisting of gzip, mcf, and twolf.

Static resource allocation

Next, we statically assign resources, namely instruction and load/store queue entries, to the HPT. Figure 4a shows the resulting QoS space for varying numbers of these resources (from 0 to the maximum value 32 in steps of 4), and Figure 4b shows the resulting IPC values. The HPT is gzip and the LPT is bzip2. Figure 4a also shows the points reached by the round-robin (RR), icount, and flush instruction fetch policies for comparison. The graph in Figure 4a is not symmetrical because we fetch and issue instructions from the HPT gzip first.

By controlling resource allocation, we can navigate through the QoS space and bias a workload's execution to a prioritized thread. Figure 4b shows that, in this case, total throughput remains almost unchanged. Only when we reserve no entries or all entries for the HPT does throughput degrade, because only one thread is running. For the other cases, both threads are ILP and don't occupy IQ entries for a long time. Hence, prioritizing the fetch and issue of the HPT delivers almost full speed quickly but, on the other hand, degrades the LPT quickly. This shows that if the OS could control the resource allocation of an SMT, it would be ready to handle QoS requirements

and real-time constraints. Next, we propose a mechanism by which the OS can accomplish one of such real-time constraints: executing a thread, at least, at a given IPC.

Dynamic resource allocation

Our dynamic allocation mechanism controls a designated HPT's execution speed by dynamically allocating resources to it. The mechanism ensures that the HPT runs at a target IPC that represents *x* percent of the IPC it achieves running alone on the machine. At the same time, the mechanism tries to maximize the throughput of the remaining LPTs as well.

Underlying our mechanism is the observation that in order to realize *x* percent of a given job's overall IPC, it is sufficient to realize *x* percent of the maximum possible IPC at every instant throughout the job's execution. We employ two alternating phases[13] in order to accomplish this objective:

- *Sampling.* The HPT receives all shared resources, and LPTs temporarily stop. As a result, we obtain an estimate of the HPT's full speed, called the local IPC. To counteract LPTs interference, this phase is divided into a warm-up phase of 50,000 cycles and an actual sampling phase of 10,000 cycles.
- *Tuning.* Our mechanism first determines a local target IPC, which is the local IPC computed in the last sampling period multiplied by the target percentage given by the OS. Next, we tune resource allocation every 15,000 cycles for a period of 1.2 million cycles. Thus, each tuning phase consists of 80 subphases of 15,000 cycles each. At the end of each subphase, the IPC of the HPT is computed. If this IPC value is lower than the target IPC given by the OS, the HPT receives more resources. Otherwise, the number of resources dedicated to the HPT decreases. The number of resources is architecture dependent and should be established experimentally for a particular processor.

The resources we consider are rename registers, instruction and load/store queue entries, and ways in the eight-way set associative L2 cache. Implementing our mechanism requires counters in the front end to track the number of IQ entries and registers used by each thread. This is no more complex than icount, which tracks IQ entries. We only add counters for the registers, which are incremented in the decode stage and decremented when instructions commit. In addition, reserving L2 cache ways requires small changes in the least recently used (LRU) cache replacement policy. We can compute the IPC in each tuning subphase with just a few instructions, and, if no cache access is required, this computation is not time consuming.

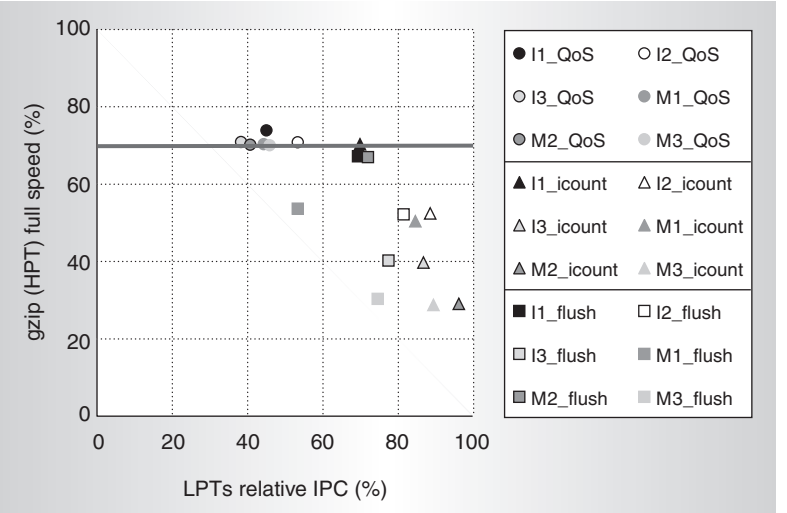

Figure 5. QoS space resulting from dynamic resource allocation.

Figure 5 shows by an example that our QoS mechanism works. In this example, HPT gzip must run at 70 percent of its full speed. The *y*-axis denotes the achieved percentage of gzip's full speed. The *x*-axis denotes the achieved average speed of the LPTs as a percentage of their full speed, the speed they'd obtain if they ran as a single workload using flush. The flush and icount policies are scattered through the QoS space and almost always reach percentages for gzip much lower than 70 percent. In contrast, our QoS mechanism always achieves 70 percent of gzip's full speed or slightly more. This shows that we can isolate gzip's execution from the other threads and hence enable real-time constraints on an SMT processor. In another article, we show that we can reach arbitrary percentages with the same accuracy.[13]

Figure 6 shows total IPC values, demonstrating that our mechanism is efficient and does not starve the LPTs. The LPTs' names and workload designations are listed on the *x*-axis. For this

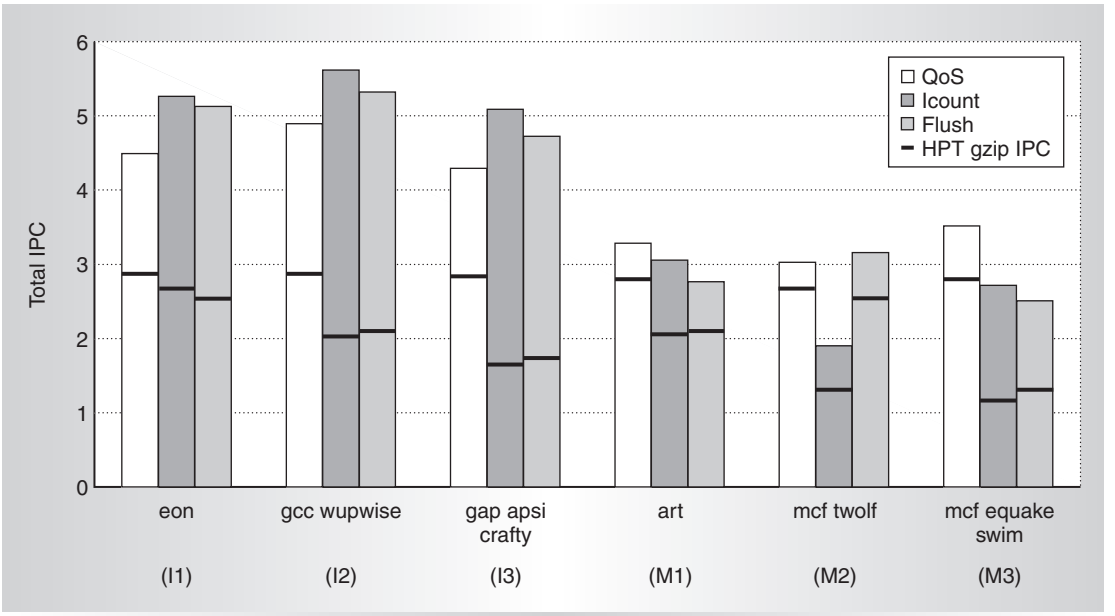

Figure 6. Effectiveness of our QoS mechanism: total throughput for various workloads, using our mechanism and using standard fetch policies icount and flush.

example, the total throughput of our QoS mechanism does not degrade, on average, compared with either icount or flush. This shows that our mechanism reserves resources only when the HPT requires them and leaves enough room for the LPTs to make significant progress. The LPTs' throughput is lower with our QoS mechanism than with the other policies because we must reserve many resources for the HPT to reach its high target percentage, even if it temporarily has no use for them. With icount and flush, the LPTs would have used these resources. Moreover, during the sampling periods, the LPTs stop, further degrading their throughput. Our primary goal, however, is to reach 70 percent of gzip's full speed, and to do so for workloads I1 through I3, we must pay a price. On the other hand, for workloads M1 and M3, our approach gives the LPTs (which are MB) less opportunity to clog the pipeline after a load that has a long latency due to an L2 cache miss. As a result, because the required speed of the HPT (which is ILP) is higher than it is in either icount or flush, the total throughput increases.

Our approach is a first step toward the use of high-performance SMT processors in future real-time systems. In future work, we extend the present approach to support several high priority threads. We will also incorporate other types of QoS requirements, like minimizing power consumption, in the present framework. MICRO

Acknowledgements

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 program (contract HPRI-CT-2001-00135). The authors would like to thank Oliverio J. Santana, Ayose Falcón, and Fernando Latorre for their work in the simulation tool.

References

1. T.R. Halfhill, "Philips Powers Up for Video," *Microprocessor Report*, no. 168, Nov. 3, 2003.
2. M. Levy, "Multithreaded Technologies Disclosed at MPF," *Microprocessor Report*, no. 168, Nov. 10, 2003.
3. R. Jain, C.J. Hughes, and S.V. Adve, "Soft Real-Time Scheduling on Simultaneous Multithreaded Processors," *Proc. 23rd Real-Time Systems Symp.* (RTSS-23), IEEE Press, 2002, pp. 134-145.
4. A. Snavely, D.M. Tullsen, and G. Voelker, "Symbiotic Job Scheduling with Priorities for a Simultaneous Multithreaded Processor,"

Proc. 9th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-9), ACM Press, 2000, pp. 234-244.
5. K. Luo, J. Gummaraju, and M. Franklin, "Balancing Throughput and Fairness in SMT Processors," *Proc. IEEE Int'l. Symp. Performance Analysis of Systems and Software* (ISPASS 01), IEEE Press, 2001, pp. 164-171.
6. F.J. Cazorla et al., "Approaching a Smart Sharing of Resources in SMT Processors," *Proc. Workshop Complexity-Effective Design* (WCED), June 2004; http://www.ece.rochester.edu/~albonesi/wced04/.
7. F.J. Cazorla et al., "Improving Memory Latency Aware Fetch Policies for SMT Processors," *Proc. 5th Int'l Symp. High-Performance Computing* (ISHPC-5), LNCS Press, 2003, pp. 70-85.
8. F.J. Cazorla et al., "DCache Warn: An I-Fetch Policy to Increase SMT Efficiency," *Proc. 18th Int'l Parallel and Distributed Processing Symp.* (IPDPS 04), IEEE CS Press, 2004, pp. 74-83.
9. D. Tullsen and J. Brown, "Handling Long-Latency Loads in a Simultaneous Multithreaded Processor," *Proc. 34th Int'l Symp. Microarchitecture* (Micro-34), IEEE CS Press, 2001, pp. 318-327.
10. P.M.W. Knijnenburg et al., "Branch classification for SMT fetch gating," *6th Workshop in Multi-Threaded Execution, Architecture and Compilation* (MTEAC-6), 2002, pp. 49-56.
11. D. Tullsen et al., "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor". *Proc. 23rd Int'l Symp. on Computer Architecture* (ISCA-23), 1996, pp. 191-202.
12. T. Sherwood, E. Perelman, and B. Calder, "Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications," *Proc. 10th Int'l Conf. Parallel Architectures and Compilation Techniques* (PACT 01), IEEE CS Press, 2001, pp. 3-14.
13. F.J. Cazorla et al., "Predictable Performance in SMT Processors," *Proc. 1st Conf. Computing Frontiers* (CF 04), ACM Press, 2004, pp. 433-443.

Francisco J. Cazorla is a doctoral candidate at the Polytechnic University of Catalonia (UPC), Spain. His research interests include instruction fetch policies for SMT architectures. Cazorla has BS and MS degrees in computer science from the University of Las Palmas de Gran Canaria, Spain.

Alex Ramirez is an assistant professor in the Computer Architecture Department at UPC. His research interests include profile-guided compiler optimization, code layout optimization, and design and implementation of superscalar and multithreaded processors. Ramirez has a PhD in computer science from UPC.

Mateo Valero is a professor in the Computer Architecture Department at UPC. His research interests include high-performance architectures. Valero has a PhD in telecomunications from UPC. He is an IEEE Fellow, an Intel Distinguished Research Fellow, and an ACM Fellow. Since 1994 he is a foundational member of the Royal Spanish Academy of Engineering.

Peter M.W. Knijnenburg is an assistant professor of computer science at the Leiden Institute of Advanced Computer Science, Leiden University, The Netherlands. His research interests include adaptive and iterative compilation, interaction of compilers and computer architectures, and computer architecture. He has a PhD in computer science from Utrecht University.

Rizos Sakellariou is a lecturer in computer science at the Department of Computer Science, University of Manchester, UK. His research interests include high-performance, parallel and distributed systems. He has a PhD in computer science from University of Manchester.

Enrique Fernández is a professor in the Computer Science and Systems Department at the University of Las Palmas de Gran Canaria (ULPGC). His research interests include high-performance architectures. He has an industrial engineering degree from the Polytechnic University of Las Palmas and a PhD in computer science from ULPGC.

Direct questions and comments about this article to Francisco J. Cazorla, Computer Architecture Dept., Polytechnic University of Catalonia, C/Jordi Girona 1-3, Edifici D-6, Campus Nord 08034 Barcelona, Spain, fcazorla@ac.upc.es.