A Predictable Simultaneous Multithreading Scheme for Hard Real-Time

Jonathan Barre, Christine Rochange, and Pascal Sainrat

Institut de Recherche en Informatique de Toulouse,
Université de Toulouse - CNRS, France
{barre,rochange,sainrat}@irit.fr
http://www.irit.fr/TRACES

Abstract. Simultaneous multithreading (SMT) processors might be good candidates to fulfill the ever increasing performance requirements of embedded applications. However, state-of-the-art SMT architectures do not exhibit enough timing predictability to allow a static analysis of Worst-Case Execution Times. In this paper, we analyze the predictability of various policies implemented in SMT cores to control the sharing of resources by concurrent threads. Then, we propose an SMT architecture designed to run one hard real-time thread so that its execution time is analyzable even when other (non critical) threads are executed concurrently. Experimental results show that this architecture still provides high mean and worst-case performance.

1 Introduction

These last years, the complexity of embedded software has grown exponentially. As an example, it is now expected that next-generation upper class cars will include as much as 1GB binary code [1]. The explosion of the software size is due to the implementation of more and more functionalities, either to improve safety (e.g. anti-lock braking system), to augment the conveniences for the users (e.g. automatic control of windscreen wipers) or to address environmental issues (e.g. control of gas emissions).

In the same time, it is not desirable to increase the number of computing nodes at the same pace because this might raise difficulties concerning interconnections among others. Then one solution is to integrate several tasks in a single computing node, as supported by the Automotive Open System Architecture (AUTOSAR) and Integrated Modular Avionics (IMA) initiatives.

To support multitasking, it seems unavoidable that high-performance processors, like those now found in desktop computers, will more and more often be used in embedded systems. We feel that simultaneous multithreading (SMT) cores [16] might be good candidates, especially when tasks of different criticality levels are to be executed on the same node. However, high performance processors are generally incompatible with the predictability requirements of hard real-time applications. Actually, the state of the art in the domain of Worst-Case Execution Time computation cannot handle properly some mechanisms

U. Brinkschulte et al. (Eds.): ARCS 2008, LNCS 4934, pp. 161–172, 2008.

designed to improve instruction parallelism, e.g. speculative execution. Thread parallelism adds new difficulties and we feel that it cannot be safely analyzed by static WCET estimation techniques unless the hardware is designed for predictability. This is what this paper deals with.

We propose a WCET-aware architecture for a processor implementing simultaneous multithreading. This architecture is aimed at being able to execute *one* thread with hard real-time constraints in parallel with less critical threads. The issue is that the timing behavior of the hard real-time thread must be predictable through static analysis so that it can be proved that it will always meet its deadlines. The solution resides in carefully selecting/designing the instruction distribution and scheduling policies that control the sharing of internal resources between concurrent threads.

We are aware that some applications would require that *several* threads with hard real-time constraints be executed on the same processing node. Our architecture provides timing predictability for a *single* critical thread, but it can be considered as a first step towards designing an architecture that can handle several critical threads.

The paper is organized as follows. In Section 2, we give some background information on simultaneous multithreading. In particular, we review the resource distribution and scheduling strategies proposed in the literature and implemented in commercialized SMT processors. Section 3 describes the architecture that we propose to execute a real-time thread with a predictable Worst-Case Execution Time, in parallel with non real-time threads. Experimental results are provided and discussed in Section 4. Section 5 reviews related work and we conclude the paper in Section 6.

2 Simultaneous Multithreading and Timing Predictability

Simultaneous multithreading (SMT) processors execute several threads concurrently to improve the usage of hardware resources (mainly functional units) [16]. Concurrent threads share common resources: instruction queues, functional units, but also instruction and data caches and branch predictor tables. In this paper, we focus on the pipeline resources (we leave the issues related to caches and branch prediction for future work).

Two kinds of pipeline resources should be distinguished: *storage* resources (instruction queues and buffers) keep instructions for a while, generally for several cycles, while *bandwidth* resources (e.g. functional units or commit stage) are typically reallocated at each cycle [14]. The sharing of storage resources is controlled both in terms of space and time: there are several possible policies to distribute the resource entries among the active threads (*distribution* policies) and to select the instructions that will leave the resource at each cycle (*scheduling* policies). Space sharing does not make sense for bandwidth resources since they are reallocated on a cycle basis. Depending on their distribution and scheduling schemes, the sharing of resources can be a major source of indeterminism to the timing behavior of a thread.

In this section, we describe the most common distribution and scheduling policies (either in research or in industrial projects) and we highlight the predictability problems that may arise when considering a hard real-time thread executing along with arbitrary threads on an SMT processor.

2.1 Resource Distribution Policies

The most flexible scheme for distributing the entries of a resource (e.g. an instruction queue) among the threads is the *dynamic distribution* policy under which any instruction from any thread can compete for any free entry.

This policy was the first considered in academic research. The dynamic distribution policy was implicitly retained to maximize the resource usage (which is the primary objective of simultaneous multithreading), ensuring that any resource would be used if any thread needed it. However, since there is no limit on the number of resource entries that can be held by a thread, some threads might undergo starvation when the all the resource entries are used by other threads. When starvation may happen depends on the respective behaviors of the concurrent threads. Considering a real-time thread, it cannot be guaranteed that one of its instructions that would require an entry in a dynamically-distributed resource get it immediately and the time it might have to wait before being admitted by the resource cannot be bounded (unless very pessimistically). Hence, with dynamic distribution, the worst-case execution time of a thread cannot be estimated due to the high variability of the delays to gain access to shared resources. This policy is therefore not suitable for hard real-time applications.

The *dynamic distribution with threshold* policy was designed to minimize the risks of starvation. Now each thread cannot retain more slots in a resource than a given threshold (usually a percentage of the resource capacity, greater than one over the number of threads). A single thread cannot monopolize all the entries. This policy is used to control the instruction schedulers in the Pentium 4 Hyper-Threading architecture [13].

In the context of hard-real time applications, dynamic distribution with threshold still has an unpredictable nature: it cannot be determined whether a thread will be allowed to hold as many resource entries as the threshold since some entries might be used by some other threads. Then it might be delayed to get an entry in a storage resource even if it has not reached its threshold and this delay cannot be upper bounded. Hence, it is not possible to derive an accurate WCET estimation for a real-time thread.

The distribution of resources can also be *static*: each resource is partitioned and each thread has a private access to one partition. This completely prevents starvation and ensures a fair access to the common resources to all threads. The performance may not be optimal in this case since some threads may be slowed down due to a lack of resource while other threads might underuse their partition. Static partitioning is widely used to share instruction queues among threads in SMT implementations [9][13], except in the out-of-order parts of the pipeline (like the issue queues of the Power5 and the schedulers queues of the

Pentium 4). This is probably because a partitioned queue is easier to implement than a dynamically shared queue.

Static resource distribution is naturally the most adequate to hard real-time system since it is fully deterministic. Actually, each thread is granted a given amount of resources which it cannot exceed and which cannot be used by another thread. Then the behavior of this thread with regards to statically-partitioned resources does not depend on surrounding threads.

2.2 Scheduling Policies

Besides to their distribution policy, shared resources are also controlled by a scheduling policy that arbitrates between threads to select the instructions that can leave the resource and go forward in the pipeline. Possible schemes are overviewed in the next section.

The most common scheme is the very simple *Round-Robin* (RR) policy. It gives each thread its opportunity in turn in a circular way, regardless of the behavior of the other threads. One should distinguish the *optimized round-robin* policy (O-RR) that skips the turn of a thread that has not any ready instruction (or an empty thread slot) from the *strict round robin* (S-RR) algorithm that might select an idle thread. As RR is completely oblivious of what happens in the processor, it is reputed to exhibit moderate performance, in contrast with context-aware policies.

Considering hard real-time applications and WCET calculation, S-RR is a very predictable policy. It is very easy to determine when the instructions of a real-time thread will be selected, since it is fixed that the thread can be selected every $1/n$ cycles if the processor is designed to handle up to n active threads. The O-RR algorithm slightly impairs predictability but the delay between two successive active turns for a thread can still be upper-bounded. The access from almost all the statically-shared queues to downstream resources is controlled by an O-RR policy in the Intel Pentium 4 [13] and by an S-RR strategy in the IBM Power5 [9].

The *icount* policy is another possible strategy that is widely considered for scheduling the fetch queue in academic projects [2][3][5]. It is based on dynamic thread priorities which are re-evaluated at each cycle to reflect the number of instructions of each thread present in the pre-issue pipeline stages (the highest priorities are assigned to the threads that have the fewest instructions in these stages). Instructions of several threads can be fetched simultaneously, with the constraint that the thread with the highest priority is satisfied first. Several variants of the *icount* strategy have been proposed in the literature: they use some counters related to each thread as a basis for updating thread priorities. For example, the *brcount* policy [17] considers the number of branches in the pre-issue stages while the *drca* policy is based on the resource usage [5]). Unfortunately, priority-based thread scheduling makes the timing behavior of one thread dependent on the other active threads. Hence, *icount* and parented policies cannot be used in the context of a hard real-time thread requiring WCET calculation.

Another policy, that we call the *parallel* scheme, equally partitions the bandwidth among all the active threads by selecting the same number of instructions for each thread. This means that all threads can have instructions progressing in the pipeline at every cycle. As far as we know, this policy in only used in the IBM Power5 to select instructions for commit [9]. Each core of this processor is a 2-thread SMT that can commit a group of instructions from each thread at each cycle. This policy is fully deterministic as the progression of one thread is independent from the other threads.

3 A Predictable SMT Architecture

Our objective is to design a multithreaded (SMT) architecture that makes it possible to analyze the Worst-Case Execution Time of real-time tasks. As a first step, we propose an SMT processor where *one* hard real-time thread can run along with other non critical threads. The hard real-time thread should execute with no interference with the other threads to exhibit an analyzable timing behavior. In the same time, performance should be preserved for the other threads so that SMT benefits are not cancelled by our modifications in the architecture.

3.1 Basic Pipeline Structure

To illustrate our approach, we consider the basic SMT pipeline structure shown in Figure 1. In the fetch stage (IF), instructions of one thread are read from the instruction cache and stored in the fetch queue (FQ). There, they wait to be selected for decoding (ID), after what they enter the decode queue (DQ). After renaming (RN), they are allocated into the reorder buffer (ROB). The EX (execution) stage selects instructions with ready inputs and an available functional unit from the ROB and issues them to functional units. Instructions from the same thread might be executed *out-of-order* to improve instruction-level parallelism. Terminated instructions are finally read from the ROB by the CM (commit) stage and leave the pipeline.

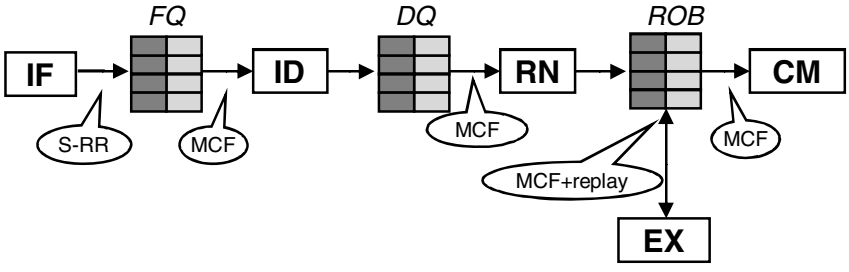

Fig. 1. A time-predictable SMT pipeline

3.2 Resource Distribution Policy

To achieve timing predictability, we statically partition each storage resource (i.e. instruction and decode queues and the reorder buffer). As discussed in the previous section, only static partitioning can make the timing behavior of a real-time thread analyzable. This is illustrated in Figure 1 where a capacity of two active threads is assumed: the fetch and decode queues and the reorder buffer are statically-distributed in two partitions, one for each thread.

3.3 Thread Scheduling

At each cycle, the IF stage fetches a sequence of instructions from a single thread:, the thread to be fetched is selected according to an S-RR policy.

As said previously, the fetch queue is statically partitioned. We must now specify the algorithm implemented to select the instructions to be decoded among those stored in the fetch queue. To have the hard real-time thread (also referred to as *hrt-t* in the following) processed in a way that does not depend on concurrent threads, we consider a partial fixed-priority scheme, where instructions from *hrt-t* are selected in the first place. If there are less *hrt-t* instructions in the fetch queue than the decode bandwidth, instructions from the non critical threads are selected on an O-RR basis. We will refer to this scheme as the *Most-Critical-First* (MCF) scheduling policy.

After decoding, instructions enter the partitioned decode queue. They are selected for renaming using the predictable MCF strategy described above. Once renamed, instructions are stored in the statically-partitioned reorder buffer where they wait for their input operands. When an instruction is ready, it becomes eligible for being issued to a functional unit. Besides to the intra-thread instruction scheduling policy (e.g. instructions are selected on an oldest-first basis), the thread scheduling policy has to arbitrate between the threads that have some ready instructions. As far as the hard real-time thread (*hrt-t*) is concerned, the delay for one of its ready instructions to be selected must not depend on concurrent threads. To achieve this, we complement the most-critical-first scheduling with a *Replay* scheme. When an *hrt-t* instruction is ready for execution, three situations can be observed:

- if the required functional unit is free, the instruction can be issued immediately;
- if the required functional unit is already allocated to an instruction that also belongs to the hard real-time thread, the execution is delayed. However, this is statically analyzable since it only depends on the thread own behavior.
- if the required functional unit is used by an instruction from a non critical thread, this instruction is squashed from the functional unit and switched back to the ready state in the reorder buffer. This means that it will have to be issued again later on. The functional unit is then immediately allocated to the *hrt-t* thread. Due to the MCF scheduling, this situation can only happen when a non critical thread executes an operation with a latency greater than one cycle in a non pipelined functional unit (in our example core, this only concerns divisions and loads/stores).

Finally, terminated instructions are selected for commit using a Most-Critical-First scheme that ensures timing predictability for the hard real-time thread.

4 Performance Evaluation

4.1 Simulation Methodology

We carried out the experiments reported in this paper using a cycle-level simulator developed as part of the OTAWA framework [6]. OTAWA is dedicated to WCET calculation and includes several utilities among which a cycle-level simulator built on SystemC. The simulator models generic processor architectures where each pipeline stage is seen as a box that reads instructions from an input queue, processes them by applying a set of predefined actions, and writes them to an output queue. Each queue is parameterized with a distribution policy and a scheduling policy. This timing simulator is driven by a functional simulator automatically generated from the PowerPC ISA description by our GLISS tool[1].

4.2 Processor Configuration

For the experiments, we have derived our predictable SMT architecture from a 4-way superscalar core, i.e. each stage is 4-instruction wide. The number of simultaneous threads has been fixed to 2 or 4 threads.

We have considered a perfect (oracle) branch predictor and a perfect (always hit) data cache. While we did not model the instruction cache (and in particular the related inter-thread interferences), we have considered a random 1% instruction miss rate, with a 100-cycle miss latency.

Table 1 gives the main characteristics of the pipeline. The queues are statically partitioned into 2 or 4 (number of threads). In the experiments, we consider two architectures: (a) the *baseline* core implements the O-RR scheduling policy for all the resources except for the issue to the functional units where instructions are selected on an oldest-first basis, independently of the thread they belong to; (b) the *WCET-aware* architecture is the one we designed to be time-predictable and implements our MCF policy together with the Replay scheme.

4.3 Benchmarks

A workload is composed of two or four threads compiled into a single binary code (this is because our functional simulator can only handle a single address space). The source code of each thread is embedded in a function call and the corresponding program counter is initialized at the entry point of this function. To maximize the possible inter-thread interferences, we have considered concurrent threads executing the same function. The different functions used in the experiments reported here are listed in Table 2 (their source code comes from the SNU-RT suite[2], a collection of relatively simple tasks commonly executed on hard real-time embedded systems).

[1] `http://www.irit.fr/Gliss`

[2] `http://archi.snu.ac.kr/realtime/benchmark`

Table 1. Baseline configuration

Parameter		Value
Pipeline width		4
Fetch queue size		16
Decode queue size		16
Reorder buffer size		16
Functional unit latencies	MEM (pipelined)	2
	ALU1	1
	ALU2	1
	FALU (pipelined)	3
	MUL (pipelined)	6
	DIV (pipelined)	15

Table 2. Benchmarks functions

Function	Comments
`fft1k`	Fast Fourier Transform
`fir`	FIR filter with Gaussian number generation
`ludcmp`	LU decomposition
`lms`	LMS adaptative signal enhancement

All executables were compiled using `-O` directive that removes most of useless accesses to memory while keeping the code algorithmic structure (which allows performing the flow analysis on the source code).

4.4 Experimental Results

To serve as a reference, we have simulated each thread in parallel with dummy threads, i.e. threads that do not execute any instruction nor allocate any dynamically-shared resource and, therefore, do not interfere with the main thread. All the storage resources were considered as partitioned. Raw results (execution time of the main thread, expressed in cycles, measured on the baseline, unpredictable architecture) are given in Table 3. The execution time of the reference thread is always longer with 4 threads than with 2 threads because the overall capacity of the resources is the same in both cases. Then, in the 4-thread configuration, each thread has smaller private partitions.

Table 3. Reference execution times

	2 threads	4 threads
`fft1k`	1 239 147	2 384 030
`fir`	25 022	47 897
`ludcmp`	4 165	8 205
`fft1k`	390 815	752 858

Table 4 shows how the same threads execute on the baseline (unpredictable) SMT architecture when they are concurrent to other (identical) threads. The reported execution times are those of the last completed threads: all the other threads have a lower execution time. In each case, we indicate the increase in the execution time over the reference measurements reported in Table 3.

Table 4. Execution times in the unpredictable SMT core

	2 threads	**4 threads**
`ff1k`	1 345 598 *+8.6%*	3 009 002 *+26.2%*
`fir`	26 548 *+6.1%*	60 691 *+26.7%*
`ludcmp`	5 068 *+21.6%*	10 686 *+30.2%*
`lms`	420 161 *+7.5%*	951 210 *+26.3%*

Table 5. Percentages of instructions delayed by concurrent threads

		delays (# cycles)					
		0	**1-5**	**6-10**	**11-20**	**21-30**	**>30**
`fft1k`	***2 threads***	93.93%	4.18%	1.25%	0.58%	0.04%	0.02%
	4 threads	94.50%	5.29%	0.09%	0.11%	0.00%	0.00%
`fir`	***2 threads***	92.93%	5.88%	0.76%	0.38%	0.02%	0.03%
	4 threads	96.66%	3.16%	0.09%	0.09%	0.00%	0.00%
`ludcmp`	***2 threads***	89.32%	9.83%	0.19%	0.43%	0.04%	0.19%
	4 threads	98.88%	1.01%	0.08%	0.04%	0.00%	0.00%
`lms`	***2 threads***	92.16%	7.08%	0.47%	0.24%	0.03%	0.03%
	4 threads	96.91%	2.89%	0.09%	0.11%	0.00%	0.00%

As expected, the global execution time for two or four "real" threads is higher than the execution time of one thread executed in parallel with dummy threads. This is due to threads competing for accessing dynamically-shared resources. Table 5 gives an insight into how often and how long instructions from one thread are delayed due to instructions from concurrent threads. The numbers indicate the percentage of instructions that have been delayed by n cycles to get a resource (physical register or functional unit). It appears that about 10% of the instructions are delayed by a concurrent thread in a 2-thread core. Note that some instructions undergo severe delays (some delays as long as up to 90 cycles have been observed).

The results achieved with our predictable core (with the *Most-Critical-First* scheduling policy and the *Replay* scheme) are shown in Table 6. Again, the execution times are those of the last completed thread. They do not concern the hard real time thread that exhibits, *in all cases*, the same execution time as

Table 6. Execution times in the predictable SMT core

	2 threads	**4 threads**
`ff1k`	1 867 441 *+38.8%*	3 503 297 *+16.4%*
`fir`	37 401 *+40.9%*	68 023 *+12.1%*
`ludcmp`	5 302 *+4.6%*	12 012 *+12.4%*
`lms`	520 369 *+23.8%*	1 001 526 *+5.3%*

the one given in Table 3. The performance loss (in terms of execution time increase) against the baseline SMT architecture is indicated in each case.

Naturally, the scheduling schemes that we implemented to insure timing predictability for one critical thread do degrade performance. However, the loss is moderate: 27% on a mean for a 2-thread core and 11.6% for a 4-thread core. The reason why the loss is higher for the 2-thread core is that, in that case, a quarter (instead of a half) of the statically-partitioned resources is dedicated to the real-time thread. Then roughly 25% (instead of 50%) of the instructions are prioritized at the expense of the other 75% (50%). This lets more opportunities for non-critical instructions to execute normally.

5 Related Work

Crowley and Baer [7] consider analyzing an SMT processor using a widely used approach for WCET computation (namely the IPET technique [11]). They express all the possible thread interleavings within the ILP (Integer Linear Programming) formulation of WCET computation. Naturally, the size of the generated ILP specification grows exponentially with the size of the threads and, unless considering very simple tasks (with few flow control), the problem cannot be solved in a reasonable time. This assessment is at the root of our work on predictable SMT architecture: we believe that the interleaving of threads must be controlled at runtime (i.e. by the hardware) to make it efficient and deterministic so that it can be taken into account when estimating the worst-case execution times of tasks with strict deadlines. Other works aim to make the timing behavior of threads more predictable through the use of appropriate thread scheduling strategies at the system level. Lo *et al.* [12] explore algorithms used to schedule real-time tasks on an SMT architecture. However, they do not take into account the possible interferences between active threads inside the pipeline and their effects on the worst-case execution times. In [10], Kato *et al.* introduce the notion of Multi-Case Execution Time (MCET): it is computed from the different WCETs of the considered thread when it runs along with different concurrent threads. We feel that the number of different possible WCET values for a thread

might be considerable as soon as each thread exhibits a large number of possible execution paths (then the number of possible interleavings with other threads might be huge). This is why we believe that these works require a predictable architecture (like the one we propose in this paper) to get valid results.

In [8], the goal is to preserve as much as possible the performance of specified prioritized threads, while still allowing other threads to progress. This goal is close to ours except that it does not insure timing predictability and thus is not appropriate in a strict real-time context. Some architectural schemes have been proposed by Carzola *et al.* [2][3][4] to guarantee a quality of service for a set of threads. This solution mainly targets soft real-time tasks and is out of the scope of our work. The CarCore processor [18] features simultaneous multithreading with two specialized pipelines: one for data processing instructions and one for memory accesses. Instructions from four active threads can process in parallel in the pipelines. This architecture can support *one* critical thread through a priority-based instruction scheduling policy. Contrary to our architecture, the CarCore processor cannot execute instructions out of order.

6 Conclusion

Higher and higher performance requirements, related to an ever increasing demand for new functionalities, will make it unavoidable to use advanced processing cores in embedded systems in the near future. Multithreaded cores might be good candidates to execute several tasks of different criticality in parallel. However, current simultaneous multithreading processors do not exhibit enough timing predictability to fit certification requirements of critical applications. Actually we feel that the thread interleaving cannot be handled by usual static analysis techniques. This is why we think that the solution does reside in the design of specific hardware.

We have proposed a predictable SMT architecture where the policies implemented to control the sharing of internal resources among threads are designed to allow a predictable execution of a single hard real-time thread concurrently with less critical other threads. All the storage resources (instruction queues and buffers) are statically-partitioned and low-level thread scheduling is done using a *Most-Critical-First* strategy that gives priority to the hard real-time thread. This is complemented by the *Replay* scheme that ensures that an instruction of the critical thread cannot be delayed by an instruction of a non-critical thread for accessing a functional unit.

Experimental results show that, while the hard real-time thread executes as fast as if it was alone in the pipeline, the performance loss for the other threads is moderate. It is less than 12% for a 4-thread predictable SMT core.

As future work, we intend to address some issues that were ignored in this preliminary study, like the strategy for sharing the instruction and data caches, as well as the branch predictor, so as to maintain full timing predictability for the critical thread. We will also investigate solutions that would allow the concurrent execution of several critical threads.

References

1. Broy, M., et al.: Engineering Automotive Software. Proceedings of the IEEE 95(2) (2007)
2. Cazorla, F., et al.: QoS for High-Performance SMT Processors in Embedded Systems. IEEE Micro 24(4) (2004)
3. Cazorla, F., et al.: Predictable Performance in SMT Processors. In: ACM Conf. on Computing Frontiers (2004)
4. Cazorla, F., et al.: Architectural Support for Real-Time Task Scheduling in SMT Processors. In: Int'l Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES) (2005)
5. Cazorla, F., et al.: Dynamically Controlled Resource Allocation in SMT Processors. In: 37th Int'l Symposium on Microarchitecture (2004)
6. Cassé, H., Sainrat, P.: OTAWA, a Framework for Experimenting WCET Computations. In: 3rd European Congress on Embedded Real-Time Software (2006)
7. Crowley, P., Baer, J.-L.: Worst-Case Execution Time Estimation for Hardware-assisted Multithreaded Processors. In: HPCA-9 Workshop on Network Processors (2003)
8. Dorai, D., Yeung, D., Choi, S.: Optimizing SMT Processors for High Single-Thread Performance. Journal of Instruction-Level Parallelism 5 (2003)
9. Kalla, R., Sinharoy, B., Tendler, J.: IBM Power5 Chip: A Dual-Core Multithreaded Processor. IEEE Micro 24(2) (2004)
10. Kato, S., Kobayashi, H., Yamasaki, N.: U-Link: Bounding Execution Time of Real-Time Tasks with Multi-Case Execution Time on SMT Processors. In: 11th Int'l Conf. on Embedded and Real-Time Computing Systems and Applications (2005)
11. Li, Y.-T. S., Malik, S.: Performance Analysis of Embedded Software using Implicit Path Enumeration. In: Workshop on Languages, Compilers, and Tools for Real-time Systems (1995)
12. Lo, S.-W., Lam, K.-Y., Kuo, T.-W.: Real-time Task Scheduling for SMT Systems. In: 11th Int'l Conf. on Embedded and Real-Time Computing Systems and Applications (2005)
13. Marr, D., et al.: Hyper-Threading Technology Architecture and Microarchitecture. Intel Technology Journal 6(1) (2002)
14. Raasch, S., Reinhardt, S.: The Impact of Resource Partitioning on SMT Processors. In: 12th Int'l Conf. on Parallel Architectures and Compilation Techniques (2003)
15. Tuck, N., Tullsen, D.: Initial Observations of the Simultaneous Multithreading Pentium 4 processor. In: 12th Int'l Conf. Parallel Architectures and Compilation Techniques (2003)
16. Tullsen, D., Eggers, S., Levy., H.: Simultaneous Multithreading: Maximizing On-Chip Parallelism. In: 22nd Int'l Symposium on Computer Architecture (1995)
17. Tullsen, D.,et al.: Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In: 23rd Int'l Symposium on Computer Architecture (1996)
18. Uhrig, S., Maier, S., Ungerer, T.: Toward a Processor Core for Real-time Capable Autonomic Systems. In: IEEE Int'l Symposium on Signal Processing and Information Technology (2005)