An Architecture for the Simultaneous Execution of Hard Real-Time Threads

Jonathan Barre, Christine Rochange and Pascal Sainrat
Institut de Recherche en Informatique de Toulouse – Université de Toulouse
HiPEAC European Network of Excellence
Toulouse, France
{barre, rochange, sainrat}@irit.fr

***Abstract*— Simultaneous multithreading (SMT) processors might be good candidates to fulfill the ever increasing performance needs of embedded applications. However, off-the-shelves SMT architectures do not fit the timing predictability requirements of hard real-time systems: to schedule critical threads so that they are guaranteed to meet their deadlines, it is necessary to estimate their Worst-Case Execution Times which is not possible when simultaneous threads might interfere. In this paper, we propose an SMT architecture designed to enforce isolation between hard real-time threads so that their worst-case execution time can be safely estimated. We report experimental results that show that this architecture still provides a high level of performance and we give an insight into how the thread isolation feature could be controlled by a real-time task scheduler.**

I. Introduction

Embedded systems tend to integrate more and more advanced functionalities and to provide more and more guarantees for safety. This results in an explosion of the software size and complexity. In the same time, increasing the number of computing nodes might raise difficulties, particularly for interconnections. This is why standards like Automotive Open System Architecture (AutoSAR) and Integrated Modular Avionics (IMA) promote the integration of several tasks on a single computing node.

In this context, embedded system designers start considering high-performance processors, like those now found in desktop computers, to support multitasking in embedded software. Simultaneous multithreading (SMT) cores [19] might be good candidates, especially when tasks of different criticality levels are to be executed on the same node.

However, thread parallelism is 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 (WCET) computation cannot handle properly the effects of thread interferences. We argue that existing techniques could be used if the hardware was designed for predictability.

In this paper, we propose an SMT architecture able to execute several hard real-time threads in parallel with less critical threads. The issue is that the timing behavior of a hard real-time thread must be predictable through static analysis so that it can be scheduled properly to meet its deadlines. The timing predictability is achieved by implementing specific policies for the distribution and scheduling of the processor internal resources.

The paper is organized as follows. In Section 2, we give some background information on WCET computation techniques and we review the predictability of resource distribution strategies in an SMT core. We also discuss related work. Section 3 describes the architecture that we propose to execute real-time threads with a predictable Worst-Case Execution Time. Experimental results are provided and discussed in Section 4. We give concluding remarks the paper in Section 5.

II. Simultaneous Multithreading and Timing Predictability

A. Computing Worst-Case Execution Times

Since it is often impossible to measure every possible execution path in a piece of code or to determine which path exhibits the longest execution time, the Worst-Case Execution Times (WCETs) of highly-critical tasks are usually determined using static analysis techniques. Typically, three steps are necessary. The first one analyses the flow control (loop bounds, infeasible paths, etc.). The second step determines the execution times (or costs) of small code parts (e.g. basic blocks) when executed on the target hardware. Finally, flow information and partial execution times are combined to produce an upper bound of the worst-case execution time of the task (the most popular method to do this is the Implicit Path Enumeration Technique [16]).

In this paper, we focus on the second step and we present an architecture that makes it possible to compute the worst-case costs[1] of the basic blocks of a critical thread that might be executed concurrently to some unknown (either critical or not) threads.

Our method to estimate the cost of a basic block is based on parameterized execution graphs [18] and considers any

[1] The cost of a basic block is the time between the end of the block (when its last instruction leaves the pipeline) and the end of the preceding block (i.e. the cost is the part of the execution time of the block that does not overlap with the execution time of the previous block

possible context (pipeline state) for the execution of the block. The times at which the instructions progress through the pipeline are computed as a function of the release times of the resources by previous instructions. These times are then used to derive an upper bound of the block cost. This technique is able to model most of single-threaded architectures but cannot take into account the effects of thread interferences that would occur in an off-the-shelves SMT core. The first reason is that the threads that might be co-scheduled with the critical thread under analysis are generally unknown. The second reason is that, even if those co-scheduled threads were known, the complexity of the flow controls would make it impossible to determine which basic blocks are executed simultaneously. This is why we need an architecture that would run critical threads in isolation with concurrent threads.

At this point, we mention that we do not consider instruction and data caches, neither branch predictors in this paper. These components are naturally concerned by thread interferences. We believe that clever solutions to provide thread isolation at this level while keeping a high performance level are still to be designed. A trivial solution would be to statically partition the caches and the branch prediction tables between the threads. When estimating WCETs, caches and branch predictors are taken into account in two stages. One step consists in estimating the different possible costs for a block (considering that each access to one cache might hit or miss and that a branch might be well predicted or mispredicted). The other step analyses the flow control to determine how many times each of these costs can be encountered in the execution of the threads. This second step has been the subject of many papers [1][3][4][10] but is clearly out of the scope of this paper. This is why we will ignore caches and branch predictors in this work.

B. *Thread Interferences in Standard SMT Architectures*

Simultaneous multithreading (SMT) processors execute several threads concurrently to improve the usage of hardware resources (mainly functional units) [19]. Common resources (instruction queues, functional units, but also instruction and data caches and branch predictor tables) are shared between concurrent threads. As mentioned above, we focus on the pipeline resources.

Some of these resources (instruction queues and buffers) are referred to as *storage* resources because they keep instructions for a while, generally for several cycles. On the contrary, *bandwidth* resources (e.g. functional units or commit stage) are typically reallocated at each cycle [5]. In a recent paper [2], we showed that the most deterministic policy to share an instruction queue is to partition it statically, giving each thread an equal amount of entries which it cannot exceed and which cannot be accessed by the other threads. Then the behavior of a thread with regards to statically-partitioned resources does not depend on surrounding threads. We also noticed that, among the possible scheduling policies to control access to shared resources, only the strict Round-Robin scheme (Strict Round Robin does not reallocate a slot to another thread when the selected thread does not need the resource, as would do an optimized Round Robin strategy) and the Parallel strategy (which statically partition the bandwidth between threads) ensure full determinism for the accesses to the resources.

C. *Related Work*

In [15], Kato, Kobayashi and Yamasaki introduce the notion of Multi-Case Execution Time (MCET), computed from the different WCETs of the considered thread when it runs along with different concurrent threads. However, 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: the number of possible interleavings with other threads might explode. Nevertheless, considering a predictable architecture like the one we propose in this paper would probably help in getting valid results.

As far as we know, the only attempt to analyze a standard SMT processor was done by Crowley and Baer [11]. They analyze the combined control flow graphs of concurrent threads and they express all the possible thread interleavings within the ILP (Integer Linear Programming) formulation of WCET computation [16]. Unfortunately, the size of the problem 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 motivates our work on predictable SMT architectures. Controlling the interleaving of threads at runtime should make it analyzable when estimating the worst-case execution times of tasks with strict deadlines.

In [12], appropriate scheduling strategies are used to give the priority to critical threads while still allowing other threads to progress. However, the proposed schemes only tend to increase the performance of the critical threads but they do not enforce timing predictability. As a consequence, they do not fit the requirements of hard real-time applications. Carzola *et al.* [5][6][7] have proposed some hardware schemes to guarantee a quality of service for a set of threads. Again, this solution mainly targets soft real-time tasks and is not appropriate for strict real-time.

The CarCore processor [20] features simultaneous multithreading with two specialized pipelines: one for data processing instructions and one for memory accesses. Instructions from 4 active threads can process in parallel in the pipelines (instructions of a given thread are executed in the program order). CarCore implements a priority-based instruction scheduling policy that makes it possible to ensure timing predictability for one thread. As mentioned above, we have proposed a dynamically-scheduled architecture also designed to insure isolation to one critical thread [2]. In this paper, we extend this architecture to allow the simultaneous execution of several critical threads.

III. A PREDICTABLE SMT ARCHITECTURE FOR SEVERAL CRITICAL TASKS

Our objective is to design a multithreaded (SMT) architecture that makes it possible to analyze the Worst-Case Execution Time of several hard real-time tasks running together. As a first step, we proposed in [2] an SMT processor where one hard real-time thread can run along with other noncritical threads with a predictable worst-case execution time. The hard real-time thread executes with no interference with the other threads to exhibit an analyzable timing behavior. In the same time, performance is preserved for the other threads so that SMT benefits are not cancelled by our modifications to the architecture. In this paper, we improve this architecture to make it able to run more than one critical threads in parallel, so that the worst-case timing behavior of each of these tasks only depends on the number of co-scheduled critical tasks (but not on the tasks themselves). As in [2], we allow noncritical tasks to run along in the remaining slots.

A. *Basic Pipeline Structure*

To illustrate our approach, we consider the basic SMT pipeline structure shown in Fig. 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 stage), after what they enter the decode queue (DQ). Once renamed (RN stage), they are allocated into the reorder buffer (ROB). The EX (execution) stage selects in the ROB some instructions with ready inputs and targeting an available functional unit, and issues them to their functional units. Instructions belonging to 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.

B. *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 [2], only static partitioning can make the timing behavior of a real-time thread analyzable. This is illustrated in Figure 1 where a capacity of four active threads is assumed: the fetch and decode queues and the reorder buffer are statically-distributed in four partitions, one for each thread

C. *Thread Scheduling*

At each cycle, the fetch stage fetches instructions from a single thread, which is selected according to a Strict Round-Robin (S-RR) policy. The other stages select instructions according to a policy that we call *Bandwidth Partitioning* (BP). It consists in allocating one part of the bandwidth to each critical thread.

Whenever a given resource is able to handle n instructions per cycle, the Bandwidth Partitioning policy guarantees a bandwidth b to each critical thread with:

– b equals n / nc instructions per cycle if $n \geq nc$ (where nc is the number of co-scheduled critical threads)
– b equals one instruction every p cycles, with $p = nc / n$ otherwise

Numerical examples are given in Table I.

Whenever a critical thread does not use all the bandwidth allocated to it, spare slots can be used by noncritical threads on a Round-Robin basis..

The *Bandwidth Partitioning* policy is complemented with a *Replay* scheme where the execution of an instruction belonging to a not-critical thread in a not-pipelined functional unit is aborted (i.e. the instruction is marked back "ready" in the reorder buffer) whenever the unit is required by a critical thread during its slot.

This guarantee on the available bandwidth for a critical thread is the key feature to make its timing behavior independent from co-scheduled threads and then predictable.

D. *Execution Modes*

Our proposal is a WCET-aware architecture (described above) featuring several execution modes, each one supporting a given number of critical threads. For example, a 4-threaded core might have four possible modes, respectively supporting zero, one, two or four critical threads. In the following, these modes will be referred to as m_0, m_1, m_2 and m_4 respectively

Switching between modes requires reconfiguring the resource partitioning schemes (for both storage and bandwidth distribution). This can only be done when the pipeline is empty (no thread is running).

The (worst-case) execution time of a thread might depend on the core execution mode when the thread is scheduled. We feel that a clever task scheduler could take the different mode-related WCETs of each task into account to sequence threads and could initiate mode switching when necessary.

IV. PERFORMANCE EVALUATION

A. *Simulation Methodology*

The results presented in this paper were obtained using the OTAWA[2] framework [9] developed in our research team for the purpose of analyzing execution times. OTAWA includes a cycle-level simulator built on SystemC that models generic single-threaded and SMT cores. It is driven by a functional simulator automatically generated from the PowerPC ISA description by our GLISS[3] tool. OTAWA also provides facilities to perform static WCET analysis on binary codes. In particular, it implements our method to estimate basic block costs [18] and the IPET method to compute task WCETs [16]. At this time, flow information must be provided by the user.

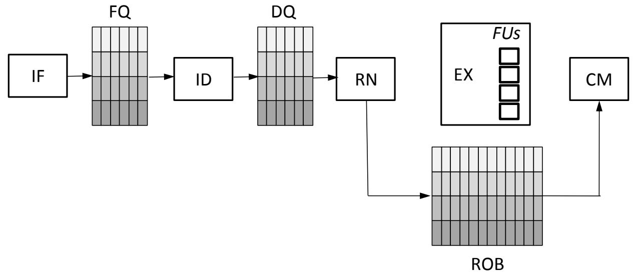

Figure 1. Basic pipeline

[2] http://www.otawa.fr
[3] http://www.irit.fr/Gliss

TABLE I. PARTITIONED RESOURCE BANDWIDTHS

Total bandwidth of the resource	***Number of critical threads*** 1	2	4
8 insts per cycle	8 insts per cycle	4 insts per cycle	2 insts per cycle
4 insts per cycle	4 insts per cycle	2 insts per cycle	1 inst. per cycle
2 insts per cycle	2 insts per cycle	1 inst. per cycle	1 inst. every other cycle
1 inst. per cycle	1 inst. per cycle	1 inst. every other cycle	1 inst. every 4 cycles
1 inst. every ℓ cycles	1 inst. every ℓ cycles	1 inst. every *2.*ℓ cycles	1 inst. every *4.*ℓ cycles

B. Processor Configuration

The main characteristics of the architecture considered in our experiments are listed in Table IV. All the functional units are pipelined. The caches are considered as perfect (i.e. all the accesses are hits) and we assume a perfect (oracle) branch predictor. Nevertheless, the instruction fetch is still aligned on cache line boundaries (we consider 8 instruction cache lines).

The architecture can handle up to four concurrent threads and can run in four different modes, corresponding to 0, 1, 2 or 4 critical threads. The associated storage and bandwidth partitions allocated to one critical thread (modes m_1, m_2 and m_4) are given in Table I. Note that the processor can change mode only when all the four thread slots are empty.

C. Benchmarks

A workload is composed of 4 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. All the results reported in this paper were obtained with four concurrent threads executing the same function (which would be likely to maximize interferences in a non-WCET-aware architecture).

The different functions used in the experiments reported here are listed in Table II (their source code comes from the SNU-RT suite[4], a collection of relatively simple tasks commonly executed on hard real-time embedded systems). All executables were compiled with `gcc` using `-O` directive.

TABLE II. BENCHMARKS

Function	*Comments*
fft1k	FFT (Fast Fourier Transform) for 1K array of complex numbers
fir	FIR filter with Gaussian number generation
lms	LMS adaptive signal enhancement
ludcmp	LU decomposition

D. Experimental Results

Table III shows the execution times observed on a standard (unpredictable) architecture with the characteristics given in Table IV and a Round-Robin based thread scheduling. The values reported in the first column are the execution times of one thread executed alone (i.e. the three other thread slots are free). The second column gives the execution time of the last terminated thread when four threads executing the same function are co-scheduled. Those times will used later as a reference to appreciate the performance sustained by our WCET-aware core.

The results obtained for execution modes m1, m2 and m4 are given in Table V and Table VI. Each value in Table 5 is the execution time of the critical thread that completes last while the values reported in Table 6 relate to the co-scheduled noncritical threads. Percentages indicate the increase in execution time compared to a thread executing alone on the standard core (i.e. mode m_0). In mode m_1, as expected, the critical thread executes as fast as if it was alone because it has priority to access every resource. Co-scheduled threads execute slower since they only use the resources that the real-time thread does not need (except for the fetch bandwidth which is fairly distributed among all threads, as in the standard core). It can be observed that the penalty for the noncritical threads is moderate: this is due to the fact that the resource requirements are small due to a limited instruction flow (at most 8 instructions fetched every 4 cycles). For the same reason, the performance degradation for two real-time threads executing in mode m_2 is small: they both execute nearly as fast as if they were alone because the amount of resources is sufficient.

TABLE III. EXECUTION TIMES ON A STANDARD CORE

Function	*One thread alone*	*Last of four threads*
fft1k	595 074	801 337
fir	14 664	16 771
lms	226 294	251 109
ludcmp	503 829	629 369

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

TABLE IV. CHARACTERISTICS OF THE ARCHITECTURE

Parameter	***Total***	***Allocated to each critical thread***		
		mode m_1	***mode m_2***	***mode m_4***
Thread slots	4			
Fetch bandwidth	8 insts/cycle	8 insts every 4 cycles		
Decode/rename/commit bandwidth	4 insts/cycle	4 insts/cycle	2 insts/cycle	1 inst/cycle
Fetch queue size	64 entries	64	32	16
Decode queue size	64 entries	64	32	16
Reorder buffer size	128 entries	128	64	32
Functional units (*latency*)				
integer ALU (*1 cycle*)	4	4	2	1
fp ALU (*3 cycles*)	2	2	1	1 every other cycle
multiplier (*6 cycles*)	2	2	1	1 every other cycle
divider (*15 cycles*)	1	1	1 every other cycle	1 every 4 cycles
memory (*1 cycle*)	2	2	1	1 every other cycle

The penalty for noncritical threads is greater than in mode m1 because half of the threads (i.e. two among four) are prioritized in mode m_2 (instead of one among four in mode m_1). In mode m_4, the amount of resources allocated to each critical thread is below its requirements and, for this reason the execution times are longer (by up to 40%). This penalty must be viewed as the price for predictability. Moreover, an SMT core executing in m_4 mode still outperforms a single-thread core with the same resources.

We have modeled the architecture proposed in this paper to estimate worst-case execution times using the approach described in [18]. It consists in representing the execution of a basic block as an execution graph that expresses the dependencies between instructions proceeding through the pipeline. Then, the time at which each node of the graph can be executed is computed as a function of the times at which resources are freed by previous instructions. The execution cost of the block can be upper-bounded by analyzing the respective execution times of the last node of the basic block and of the last node of the previous block. The reader should refer to the original paper for more details on the method. Naturally, the execution costs estimated this way are overestimated because pessimistic hypotheses about the context in which a basic block executes (i.e. the state of the pipeline that depends on the path executed before the block) are made. Actually, the cost of a block is evaluated considering any possible pipeline state while only a few of these states might be encountered in a real execution. Nevertheless, the method is simple and it helps in getting safe WCET estimates.

The WCETs that have been found for the four benchmarks considering our predictable architecture are given in Table VII. The percentages show the difference between the estimated worst-case execution times and the observed times (reported in Table V). The overestimation might seem large but is in the same range as what is often observed with WCET estimation tools for single-threaded processors (see the results of the 2006 WCET tool challenge [14]). Nevertheless, we will consider improving our model of the architecture as future work.

TABLE V. OBSERVED EXECUTION TIMES OF CRITICAL THREADS

	Execution mode		
	m_1	**m_2**	**m_4**
fft1k	595 074 -	609 008 *+ 2.3%*	858 898 *+44.3 %*
fir	14 664 -	14 672 *+0.1 %*	18 339 *+ 25.0%*
lms	226 294 -	226 301 *+0.0 %*	291 179 *+28.7 %*
ludcmp	503 829 -	503 829 *+ 0.0%*	706 542 *+ 40.2%*

TABLE VI. OBSERVED EXECUTION TIMES OF NONCRITICAL THREADS

	Execution mode		
	m_1	***m_2***	***m_4***
fft1k	766 110 *+28.7 %*	856 781 *+ 44.0%*	- -
fir	14 666 *+ 0.0%*	17 744 *+ 21.0%*	- -
lms	257 573 *+ 13.8%*	276 312 *+ 22.1%*	- -
ludcmp	602 885 *+ 19.7%*	694 233 *+ 37.8%*	- -

TABLE VII. ESTIMATED WORST-CASE EXECUTION TIMES

	Execution mode		
	m_1	***m_2***	***m_4***
fft1k	949 143 *+ 59.5%*	1 120 575 *+ 84.6%*	1 352 764 *+ 57.5%*
fir	22 479 *+ 53.1%*	24 238 *+ 65.2%*	30 442 *+66.0%*
lms	374 290 *+ 65.4%*	408 700 *+ 80.6%*	505 195 *+73.5%*
ludcmp	730 552 *+45.0 %*	766 827 *+ 52.2%*	905 786 *+ 28.2%*

E. Considering Execution Modes to Schedule Critical Tasks

In this section, we show how the multiple execution modes of our architecture (related to the number of critical threads guaranteed to exhibit a predictable timing behavior) could be used to schedule hard real-time tasks.

This is illustrated considering the PapaBench tasks set [17]. PapaBench is a benchmark that was designed to be used in academic research on the estimation of worst-case execution times. It is based on the Paparazzi software developed to control Unmanned Aerial Vehicles (UAV)[5]. In this work, we consider a subset of the PapaBench.v2 tasks listed in Table VIII. The values stand for their estimated worst-case execution times.

All these tasks are repeatedly executed at the same frequency, except for task t7 that must be launched at a five times higher frequency. Inter-task dependencies are shown in Fig. 2.

Since all the tasks have hard deadlines, they must be executed in a predictable execution mode (m_1, m_2 or m_4). If the cycle length is to short, as in Fig. 3, it is not possible to schedule all tasks in mode m_1. Mode m_2 should be selected at the beginning of the cycle since the parallelism within the task set is limited by the dependencies (note that the two free slots can be allocated to noncritical preemptable threads). However, the remaining time after task `t13` finishes is too short to execute the last tasks by pairs. A switch to mode m_4 makes it possible to schedule as many as four tasks in parallel with a guaranteed timing predictability.

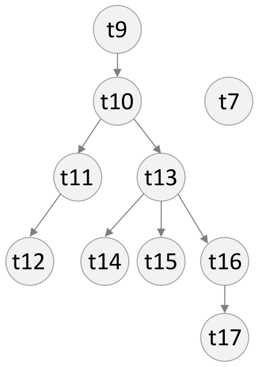

Figure 2. PapaBench inter-task dependencies

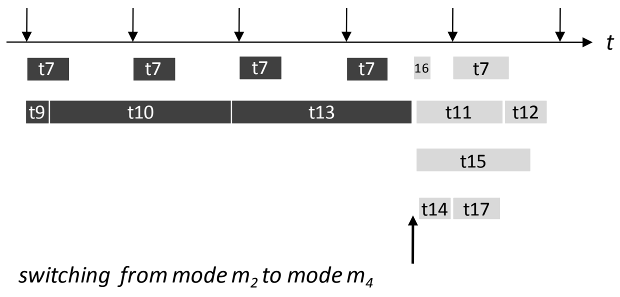

Figure 3. A possible schedule of the PapaBench tasks

[5] http://paparazzi.enac.fr

TABLE VIII. PAPABENCH TASKS

		WCET according to execution modes		
#	*task name*	m_1	m_2	m_4
t7	stabilization	217	223	297
t9	parse_gps-msg	107	127	222
t10	send_gps_pos	949	962	1160
t11	send_radIR	397	398	456
t12	send_takeOff	184	186	222
t13	navigation_update	886	941	1205
t14	course_run	139	148	171
t15	send_nav_values	513	513	599
t16	altitude_control	68	69	87
t17	climb_control	190	198	250

The simple example above shows how the predictable execution modes could be used to generate task schedules that meet deadlines. In this work, we have generated the schedule manually but we feel that execution modes could be efficiently exploited by an off-line real-time task scheduler. As mentioned before, similar approaches have been investigated to design real-time scheduling techniques for processors featuring Dynamic Voltage Scaling [13]. Note that we do not propose, in this paper, any task scheduling algorithm that would use our SMT core efficiently. This is left for future work.

V. 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 sets of tasks. However, thread timing interferences in standard cores make the estimation of worst-case execution times difficult, if not impossible. For this reason, current simultaneous multithreading processors cannot be used to execute applications with hard real-time requirements.

We believe that timing predictability can be achieved with specific hardware. In this paper, we have proposed a predictable SMT architecture that supports several critical threads. Thanks to the implementation of predictable policies for controlling the sharing of internal resources among threads, it is possible to compute an upper bound of the execution times of threads that must meet hard deadlines. The storage resources (instruction queues and buffers) are statically-partitioned and low-level thread scheduling is done using a Bandwidth Partitioning strategy that gives priority to the hard real-time threads for the access to one part of the resources.

The architecture supports four execution modes that respectively allow the predictable execution of zero, one, two or four critical threads (the remaining thread slots can be used for noncritical threads).

Experimental results show that two critical threads can be supported with a moderate performance penalty: the critical threads execute nearly as if they were alone in the pipeline and the other threads have their execution time increased by 20% to 40% (this is the price for predictability). Executing four threads in parallel naturally degrade their respective performance because of the strict resource allocation policy required to ensure determinism. However, the cost might be acceptable whenever thread parallelism is necessary.

From a simple example, we have given an insight into how a clever off-line real-time scheduler could take advantage of the various execution modes supported by the architecture. Depending on the respective execution times of the tasks to be scheduled, the scheduler might select the appropriate execution mode to ensure that the tasks will meet their deadlines.

As future work, we intend to address some issues that were ignored here, like strategies for sharing the instruction and data caches, as well as the branch predictor, so as to maintain full timing predictability for the critical threads. We will also investigate real-time scheduling algorithms that could exploit the multi-mode execution capabilities of the proposed architecture.

References

[1] M. Alt, C. Ferdinand, F. Martin, R. Wilhelm, “Cache Behavior Prediction by Abstract Interpretation”, Static Analysis Symposium, 1996.

[2] J. Barre, C. Rochange, P. Sainrat, “A Predictable Simultaneous Multithreading Scheme for Hard Real-Time”, 21st International Conference on Architecture of Computing Systems (ARCS’08), 2008 (to appear)

[3] I. Bate, R. Reutemann, “Efficient Integration of Bimodal Branch Prediction and Pipeline Analysis”, 11th IEEE Int’l Conference on Embedded and Real-Time Computing Systems and Applications, 2005.

[4] C. Burguière, C. Rochange, “On the Complexity of Modelling Dynamic Branch Predictors when Computing Worst-Case Execution Times”, ERCIM/DECOS Workshop on Dependable Embedded Systems, 2007.

[5] F. Cazorla, A. Ramirez, M. Valero, P. Knijnenburg, R. Sakellariou, E. Fernández, “QoS for High-Performance SMT Processors in Embedded Systems”, IEEE Micro, 24(4), 2004.

[6] F. Cazorla, P. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, M. Valero, “Predictable Performance in SMT Processors”, ACM Conf. on Computing Frontiers, 2004.

[7] F. Cazorla, P. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, M. Valero, “Architectural Support for Real-Time Task Scheduling in SMT Processors”, Int’l Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), 2005

[8] F. Cazorla, A. Ramirez, M. Valero, E. Fernández, “Dynamically Controlled Resource Allocation in SMT Processors”, 37th Int’l Symposium on Microarchitecture, 2004.

[9] H. Cassé, P. Sainrat, “OTAWA, a framework for experimenting WCET computations”, 3rd European Congress on Embedded Real-Time Software, 2006.

[10] A. Colin, I. Puaut, “Worst Case Execution Time Analysis for a Processor with Branch Prediction”, Real-Time Systems, 18(2), 2000.

[11] P. Crowley, J.-L. Baer, “Worst-Case Execution Time Estimation for Hardware-assisted Multithreaded Processors”, HPCA-9 Workshop on Network Processors, 2003.

[12] G. Dorai, D. Yeung, S. Choi, “Optimizing SMT Processors for High Single-Thread Performance”, Journal of Instruction-Level Parallelism, vol. 5, 2003.

[13] F. Gruian, “Hard real-time scheduling for low-energy using stochastic data and DVS processors”, International Symposium on Low Power Electronics and Design (ISPLED), 2001.

[14] J. Gustafsson, “The WCET Tool Challenge 2006”, 2nd International Symposium on Leveraging Applications of Formal Methods, 2006.

[15] S. Kato, H. Kobayashi, N. Yamasaki, “U-Link: Bounding Execution Time of Real-Time Tasks with Multi-Case Execution Time on SMT Processors”, 11th Int’l Conf. on Embedded and Real-Time Computing Systems and Applications, 2005.

[16] Y.-T. Li, S. Malik, “Performance Analysis of Embedded Software using Implicit Path Enumeration”, Workshop on Languages, Compilers, and Tools for Real-time Systems, 1995.

[17] F. Nemer, H. Cassé, P. Sainrat, J.-P. Bahsoun, M. De Michiel, “PapaBench : A Free Real-Time Benchmark”, 6th Workshop on Worst-Case Execution Time Analysis, 2006.

[18] C. Rochange, P. Sainrat, “A Context-Parameterized Model for Static Analysis of Execution Times”, Transactions on High-Performance Embedded Architectures and Compilers, 2(3), 2007.

[19] D. Tullsen, S. Eggers and H. Levy. “Simultaneous Multithreading: Maximizing On-Chip Parallelism”. In 22nd Int’l Symposium on Computer Architecture, 1995.

[20] S. Uhrig, S. Maier, and T. Ungerer, “Toward a processor core for real-time capable autonomic systems”, IEEE Int’l Symposium on Signal Processing and Information Technology, 2005.