Exploiting Spare Resources of In-order SMT Processors Executing Hard Real-time Threads

Jörg Mische, Sascha Uhrig, Florian Kluge, and Theo Ungerer
University of Augsburg, Germany
Members of the HiPEAC European Network of Excellence
{*mische, uhrig, kluge, ungerer*}*@informatik.uni-augsburg.de*

***Abstract*— We developed an SMT processor that allows a static WCET analysis of several hard real-time threads and uses the remaining resources for soft or non real-time threads. The analysis is possible, because one *Dominant Meta Thread (DMT)* is executed as if it were the unique thread on the processor and thus single-threaded WCET techniques can be applied. To provide more than one hard real-time thread the execution time of the Dominant Meta Thread is distributed by time sharing whereby the length of the time slices and periods can be adjusted at runtime.**

Our technique, called *Dominant Time Sharing (DTS)*, can be used to minimize the number of control units in embedded hard real-time systems and hence reduces the overall energy consumption and material demand.

In contrast to many other studies we are able to handle multicycle memory latencies while preserving analyzability. The proposed technique can easily be extended to access other external resources like coprocessors or reconfigurable arrays.

I. Introduction

In recent years, lots of techniques to improve the performance of general purpose processors were invented and now belong to the standard architectures of high-end processors: dynamic branch prediction, cache hierarchies, out-of-order execution, simultaneous multithreading etc. These high-performance techniques are based on speculative execution or dynamic assignment of resources at runtime. They lower the average execution time while increasing the worst case execution time (WCET). In hard real-time systems deadlines must be met even in the worst case, therefore only a lower WCET is of interest, the average case is not important. Consequently hardly any of these high-performance techniques is used in current hard real-time systems.

Simultaneous multithreading (SMT) processors are commonly build by feeding a wide superscalar out-of-order pipeline with instructions of several threads. Out-of-order execution permits the threads to share processor resources, resulting in collisions and interference between theoretically independent threads. Therefore the execution time of a thread depends on its co-schedule, the threads that are executed in parallel. For example, concurrent memory accesses disturb the timing behavior of a hard real-time thread. This makes a WCET analysis nearly impossible.

The advantage of SMT processors is the high utilization of processor resources, involving a high throughput. With higher throughput less cycles are needed for a certain job and, hence the clock rate can be reduced, which consequently reduces energy consumption. Such an improved energy efficiency is especially important for embedded systems, the main area of hard real-time applications.

To preserve this benefit while allowing hard real-time applications, we guarantee the unlimited usage of all resources to one thread, the so-called *Dominant Meta Thread (DMT)*. Further threads can only use spare resources and possibly have to drop out when requesting a resource for more than one cycle.

By privileging the DMT, it behaves like a single thread that is executed without concurrent threads in a single-threaded way. In conjunction with some modifications of the pipeline for in-order execution, the total isolation makes the DMT deterministic and thus enables a static WCET analysis to meet hard real-time requirements. Furthermore the privileged execution guarantees the fastest possible execution time of a single thread.

The additional, noncritical threads should use the remaining resources as intense as possible to achieve a high throughput. They can be used to execute non real-time or soft real-time tasks. By interfering threads with different real-time requirements, the computing power (and energy consumption) that the hard real-time thread demands for the worst case, but in the average case does not utilize, can efficiently be used.

Multiple hard real-time threads are not scheduled in the same cycle, but in consecutive cycles, using time slicing to partition the execution time of the DMT. This paper makes the following contributions:

1) An SMT architecture that allows a static WCET analysis.
2) A scheduling algorithm that executes multiple hard real-time threads concurrently on an SMT processor.
3) An issue policy that uses free resources for non critical threads without interfering the hard real-time threads.
4) Solutions to handle multicycle memory accesses.

In the next section, we present an overview of related work. The isolation of the DMT is presented in section III. Section IV describes the time sharing of the hard real-time threads. Its integration into the CarCore architecture used for evaluation is given in section V. Section VI discusses our evaluation results while VII concludes the paper.

II. Related Work

The research on real-time capable SMT processors firstly focused on a foreground thread that is nearly not affected by other background threads [1], [2]. Preserving its single-thread performance guarantees predictability and makes the foreground thread real-time capable. The first work on real-time scheduling of multiple threads [3] only covers soft real-time. Recently there is some effort to control the IPC of one real-time thread [4]. This assured IPC can even be spread over several threads [5]. These studies have in common, that they cannot fulfill *hard* real-time requirements, i.e. they cannot guarantee that a thread never misses a deadline. This is due to the fact that they use out-of-order SMT processors, where the dynamic resource sharing inside the core leads to resource conflicts and interactions between the thread slots. Hence only soft real-time is possible.

Barre et al. [6] avoid the interference by privileging one hard real-time capable thread and partitioning the resources of an out-of-order SMT pipeline equally to the thread slots. Out-of-order execution marginally boosts one single thread, while in-order superscalar pipelines increase total throughput [7]. Thus the missing performance boost of a single thread is only marginal in comparison to the reduced design complexity [8]. Therefore we avoid any dynamic features by using an in-order architecture, but we do not restrict the resource coverage of one thread by partitioning.

The first hard real-time capable multithreaded designs mainly used scalar architectures: The academic *Komodo* processor [9] provides several hard real-time threads by time sharing within a constant period of 100 cycles, while the Ubicom IP3023 microprocessor [10] uses a fixed table that assigns every cycle of a 64 cycle period to a specific thread. But both approaches use very special instruction sets (Java, respectively a small specialized instruction set for networking) and assume scratchpad memories. Only *jamuth* [11], a commercial derivate of the Komodo processor, supports slower memories for soft real-time threads.

The *Real-time Virtual Multiprocessor (RVMP)* [12] allows a static WCET analysis by partitioning threads in the time and in the space dimension (parallel usage of resources). Unfortunately its schedule is precalculated (i.e. static) and the processor cannot deal with memory latencies.

Another approach, called *Virtual Simple Architecture (VISA)* [13] guarantees the execution time of a simple hypothetical processor, but executes the threads on a high-performance, speculative processor. If the performance of the speculative processor is too low (because of incorrect speculations), the processor falls back to the simple architecture to meet the deadline. The problem of this architecture is, that between the detection of an upcoming deadline miss and the final deadline there must be enough time to execute the rest of the thread in the simple mode. Hence, if the deadline is of wide scope, VISA can be used to further increase the scope, but if there is none, it cannot improve utilization. By contrast, our approach improves the non real-time performance even with tight timing bounds.

III. Dominant Meta Thread Isolation

Instead of basing our SMT processor on an out-of-order pipeline, we use a superscalar in-order pipeline as starting point. As program execution on superscalar in-order processors is deterministic, static WCET analysis and therefore hard real-time applications are possible. Furthermore they provide high theoretical performance, as more than one instruction can be executed per cycle. But in practice, the performance can only be used to a small degree, because data dependencies and latencies extremely reduce the actual Instructions Per Cycle (IPC) rate (Tab. I).

To improve the utilization of the processor resources, while preserving the hard real-time capabilities, we provide additional noncritical threads, that run in parallel to the hard real-time capable DMT and are scheduled with fixed priorities. The additional threads must not alter the DMT's run-time behavior under any circumstances, as this would affect the WCET analysis, but unused resources should be covered to a high percentage.

A. Multithreading

Advancing a single-threaded in-order processor to multithreading requires some architectural modifications: some resources must be duplicated (program counter, instruction windows, register set), early pipeline stages must deal with multiple threads (fetch stage, issue stage) and pipeline stalls must be avoided (interruptible microcode sequences, split phase load) [14].

The DMT should be executed as if it is executed on a single-threaded processor, absolutely isolated from the noncritical threads, which requires suitable fetch and issue policies. In both stages the DMT must be absolutely preferred, i.e. the DMT must cause the same memory fetch accesses and the same assignment of instructions to functional units, no matter if there are additional threads or not. By contrast the back-end of an in-order pipeline is already partitioned (instructions in different functional units do not interfere), therefore no further pipeline modifications are necessary.

B. Memory Latency

The so far described architecture widely isolates the DMT, as long as memory accesses take only one cycle or can be pipelined. But in the embedded area, such memory is too expensive (see section V-B) and thus blocking multiple cycle latencies must be assumed. Hence, if both a noncritical and a hard real-time thread issue a load instruction in consecutive cycles, the memory controller is busy when the second load from the hard real-time thread arrives and delays its execution until the first noncritical memory access is finished.

This additional retardation only occurs if a noncritical thread issues a memory instruction in a closely preceding cycle. Consequently it is not deterministic, but for a WCET analysis the worst case - always a maximum collision - must be assumed. This can easily be handled by doubling the memory latency in the analysis, but increases the WCET compared to single-threaded execution.

To minimize this effect, the issue stage of our architecture announces a memory access of the DMT as early as possible to the memory controller and directs it not to start any memory operation until the dominant memory access arrives at the memory controller. This reduces the additional latency penalty by the number of pipeline stages between issue stage and memory controller. If the number is bigger than the memory latency the impact can be hidden completely, otherwise an incrementation of the memory latency for WCET analysis cannot be avoided, but the extent can be reduced. We call this technique *Dominant Memory Access Announcing (DMAA)*.

IV. Dominant Time Sharing

If multiple hard real-time threads are desired, they cannot be executed together with the DMT, as the former could use all resources concurrently and none are left for another hard real-time thread. But the DMT can be broken into multiple threads which are dominant alternately. Therefore the total execution time is divided into small intervals of time, called *Rounds*. Within every round, each hard real-time thread is the dominant thread for a certain number of cycles (its *Quantum*), for the rest of the round it is suspended. Applying this technique guarantees every hard real-time thread a certain fraction of the total computing power.

To put it another way, the processor is virtualized into multiple virtual processors with individual clock rates and every hard real-time thread is executed as DMT on its individual virtual processor. The virtual clock rate can be adjusted by the quantum of each thread:

$$virtualclockrate_i = \frac{quantum_i}{roundlength} \cdot realclockrate \qquad (1)$$

As context switching is done in hardware with zero overhead, schedulability is given if the sum of the virtual clock rates is not greater than the real clock rate of the system. High throughput is still guaranteed, as the spare resources are not divided into virtual processors. Instead they are distributed as a whole to the noncritical threads.

A. Slight Adaption of Quantum

If a hard real-time thread starts a multicycle memory access in the last cycle of its time slice, the next hard real-time thread cannot immediately start a memory access in the next cycle, as the memory controller is busy. Thus, the time slice of the first thread is effectively prolonged by the duration of the memory access, while the time slice of the second thread is shortened. Aggregation of this small effect (notably with short rounds) can result in a big gap between theoretical assured and really used execution time culminating in a deadline miss.

To avoid this, the quantum of the thread and the round are temporarily increased, if the last instruction within a time slice is a memory operation. The quanta of the subsequent threads in the round keep unchanged, except they also execute a memory operation in the last cycles of their particular time slice. To compensate the extension of the round, in the next round the quanta of the affected threads are (temporarily) decreased by the access time. Accordingly the length of the successive round is shortened by the number of memory access cycles multiplied by the number of temporarily prolonged threads.

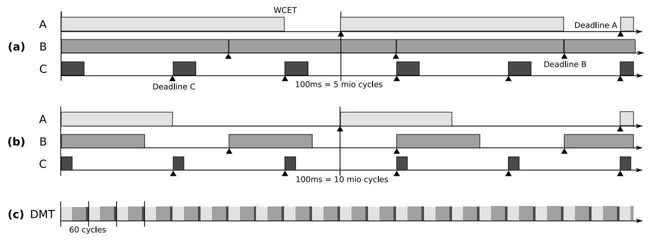

Fig. 1. Example of a periodic task set

If no further last-cycle memory accesses occur in the second round, each thread got exactly two times its quantum and both rounds together lasted for exactly double the time of an unchanged round. But even when assuming that in each round, every last instruction of a quantum is a memory access, the maximum degradation of one thread does not exceed the duration of one memory access multiplied by the number of threads. During the complete lifetime of the task-set an aggregation is impossible.

B. Connection to Traditional HRT Scheduling

Typically, a hard real-time system consists of a task-set of a fixed number of periodic threads, each with a period and a deadline. After each period a new instance of the thread is released. If we assume that the deadline is equal to the period (which is a common simplification), a thread must terminate before the end of a period. A WCET analysis provides the maximum time, the thread needs for execution.

To meet the deadlines in our model, it is sufficient to calculate a theoretical clock rate that suffices to meet the deadline:

$$virtualclockrate_i = \frac{WCET_i}{period_i} \cdot realclockrate \qquad (2)$$

With this relation and equation (1) a formula for the quantum is easily derived:

$$quantum_i = \frac{WCET_i}{period_i} \cdot roundlength \qquad (3)$$

As the quanta are integer values, the length of a round must be big enough to ensure that all quanta have reasonable values. On the other hand, one round should be as small as possible to assure fair scheduling.

C. Example

We assume a task set of three threads A, B and C with periods (and deadlines) of 100ms, 60ms and 40ms and WCETs of 4 000 000, 3 000 000 and 400 000 cycles. Fig. 1a shows the task set assuming a separate 50 MHz processor for each thread. As thread B needs the complete execution time, it is not possible to schedule all three threads together on one processor at 50 MHz.

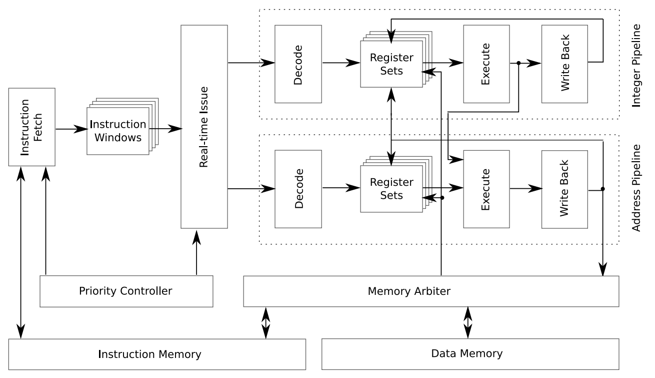

Fig. 2. CarCore Processor

To determine the minimum cycle rate to schedule all threads on one processor, we just calculate the virtual clock rate V_i for each thread ($V_A = \frac{4000000}{100ms} = 40MHz$, $V_B = 50MHz$, $V_C = 10MHz$) and sum them up (100 MHz in this example). Fig. 1b shows the execution on 3 separate processors at 100 MHz.

Therefore A gets 40%, B 50% and C 10% of one round. As a memory access takes three cycles, the cycle quantum of a thread can temporarily be reduced by three cycles. Hence, the minimum cycle quantum has to be three cycles. In order to guarantee still some execution cycles per round we choose six cycles as absolute minimum quantum for the thread with the least fraction of the round, thread C. Accordingly a round is 60 cycles long and A gets 24 cycles and B 30 cycles. The result is shown in Fig. 1c.

V. Architectural Details

The above described scheduling technique was implemented in the CarCore, a superscalar in-order processor with load-store architecture that allows simultaneous multithreading. It supports the rich and intricate Infineon TriCore instruction set [15]. We developed a cycle-accurate SystemC model of the CarCore architecture and we are currently working on a FPGA prototype.

A. General Architecture

There are two pipelines, one for integer arithmetics and one for address calculation and memory access. Similar to the original TriCore, each pipeline consists of five stages, where the first two (*Fetch* and *Real-time Issue*) are shared and the last three (*Decode*, *Execute* and *Write Back*) are independent in each pipeline. The issue policy is also inherited from TriCore: the instructions are issued in-order and two instructions from one thread can be issued in parallel, if an address instruction directly follows an integer instruction. Otherwise our SMT extension fills the pipeline with instructions from two distinct threads.

As the TriCore instruction set provides 16-bit and 32-bit instructions, the fetch stage can fetch 2 to 4 instructions per cycle by its 64-bit port to instruction memory. Each thread slot has its own *Instruction Window*, where the instructions are buffered between fetch and real-time issue. The real-time issue stage predecodes the instructions and assigns them to the appropriate pipeline, depending on the priority of the thread slot.

There is no branch prediction, because a dynamic branch prediction complicates WCET analysis and even a static one (like in TriCore) wastes cycles in the case of a misprediction that without prediction can be used for other threads.

B. Memory Model

To keep the pipeline utilized, 2 instruction or 64 bits must be fetched per cycles, therefore the instruction memory is 64KB software-managed scratchpad with 1 cycle latency. Slower memory would stall the pipeline and negate the efforts to efficiently share resources. But scratchpad memory is expensive and power consuming, therefore we chose 32MB SD-RAM for data memory (3 cycles latency for a 32 bit random access). These assumptions are comparable to commercially available TriCore boards and will be implemented on the FPGA prototype of the CarCore.

C. Thread Scheduling

The scheduling is split into two parts: Within the pipeline (namely the fetch and the issue stage) the scheduling is done by a straightforward fixed priority scheme to minimize the critical path, while the more difficult calculation of the priorities for each thread slot is done in a separate hardware module called *Priority Controller* that feeds fetch and issue stage with these priorities.

Each thread is assigned to a separate thread slot, but not all thread slots are active at the same time. The priority controller assigns the same highest priority to every hard real-time thread, but only one is active at a certain cycle, the other hard real-time threads are suspended. The noncritical threads are steadily active, but they have lower priorities.

To achieve the time sharing, each hard real-time thread has a register for its quantum. At the beginning of a round an internal counter is set to the quantum of the first hard real-time thread and this thread is activated while the other hard real-time threads are suspended. At each cycle the counter is decreased by one and when it reaches zero, the first thread is suspended while the second one is activated and the counter is set to its quantum. This continues until the last hard real-time thread was active for the number of cycles according to its quantum and the round is complete. Immediately the next round begins with activating the first hard real-time thread.

TABLE I
Benchmark Characteristics (*EEMBC, †Mälardalen)

Name	Description	IPC	int	adr	mem
a a2time*	Angle to Time Conversion	0.483	24	24	29
c canrdr*	Remote CAN Request	0.387	7	30	43
i aifirf*	Finite Impulse Response	0.439	8	35	39
r rspeed*	Road Speed Calculation	0.459	8	37	37
x crc†	Cyclic Redundancy Check	0.591	42	16	7
f fft1†	Fast Fourier Transform	0.472	20	26	31
m mm†	Matrix Multiplication	0.577	15	42	41

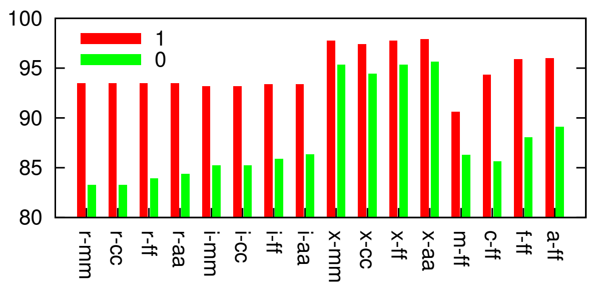

Fig. 3. Relative DMT performance (%) depending on the number of preceding instructions that are considered for announcing

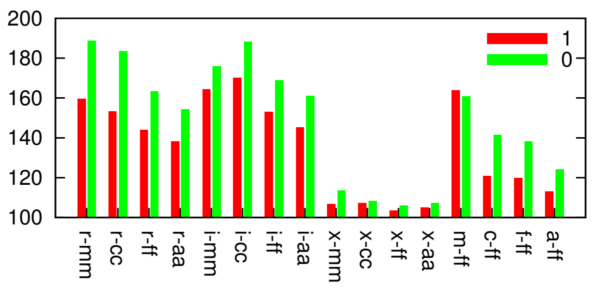

Fig. 5. Relative performance (%) of the first noncritical thread depending on the announce policy

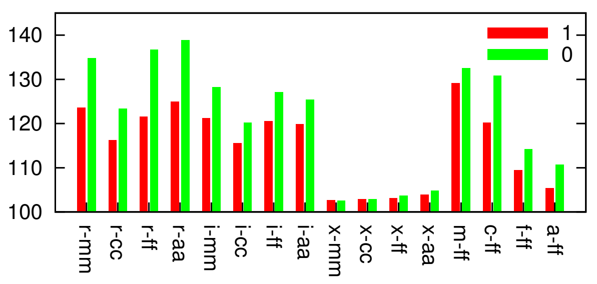

Fig. 4. Overall performance (%) depending on the number of preceding instructions that are considered for announcing

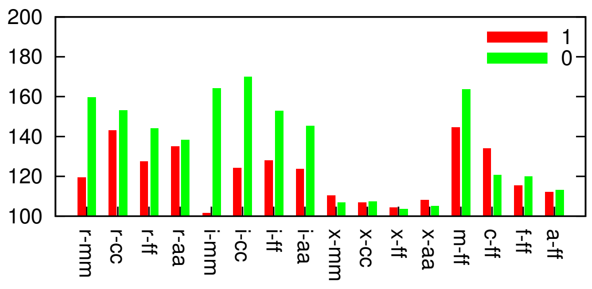

Fig. 6. Relative performance (%) of the second noncritical thread depending on the announce policy

VI. Evaluation

Benchmarks from the EEMBC automotive benchmark suite [16] and the Mälardalen WCET group [17] were compiled with the HighTec GNU C/C++-Compiler [18] and executed on a cycle-accurate SystemC model of the CarCore processor with 8 thread slots. For every benchmark the single threaded IPC and the percentage distribution of integer instructions (int), address instructions (adr) and memory latencies (mem) is given in Tab. I.

Each letter of the task-set names corresponds to a benchmark (Tab. I), the position matches the thread slot and the minus separates hard real-time from noncritical threads, e.g. *mr-iiiiii* stands for 2 real-time threads with the benchmarks *mm* and *rspeed* and 6 noncritical threads executing *aifirf*.

A. Dominant Memory Access Announcing (DMAA)

Because of the 3 cycle memory latency, memory access of a noncritical thread (NA) can block the memory controller and therefore delay the DMT when a DMT memory access (DA) is issued in one of the following two cycles. But if the memory controller knows about an upcoming DA two cycles earlier it can deny NAs in these two cycles and idle until the DA arrives, whereby NAs are completely hidden from the DMT and it is executed as if there were no other threads. In the CarCore architecture this is possible, as the issue stage can already recognize memory instructions and a memory instruction is not passed to the memory controller until the execute stage (located two stages later in the pipeline) calculated the address.

Fig. 3 shows the performance variation of task-sets with one DMT and seven noncritical thread against the number of preceding instructions that are considered for announcing. The performance of the different task-sets is measured in instructions per cycle (IPC) and presented relatively to their performance when announcing two instructions ahead (which for the DMT is the same as in single threaded mode). 1 means that a NA is only canceled if it is the directly preceding instruction of a DA and 0 stands for no announcing at all.

As expected, the performance of the DMT is lower, when there is less or no announcing, because the execution time increases. Surprisingly the choice of the noncritical threads has only marginal effect on this degradation. More important is the frequency of memory accesses of the DMT itself, but the relation is not linear (*r/rspeed* has more memory accesses than *m/mm* and *f/ffft1*, but less degradation).

By contrast, the noncritical threads (Fig. 5 shows the thread the highest noncritical priority, Fig. 6 the second highest) benefit on less announcing, as less memory accesses are delayed. Again, the variation depends more on the memory characteristic of the thread itself than on its co-scheduled threads.

Less announcing implies less retarded memory accesses and therefore a higher overall throughput, as Fig. 4 indicates. To sum up, there is a trade-off between DMT performance and overall throughput that can be used to adapt the scheduling to the demanded performance.

B. Utilization

Fig. 7,8,9 show the performance of different task-sets. Within one task-set, the time is shared in equal portions to every hard real-time thread, if there are 2 each gets 50% of time, 33.3% if they are 3, etc. The length of a round is 120 cycles. The performance of a thread is measured in IPCs and normalized to its single threaded IPC.

The evaluation shows, that the hard real-time threads reach exactly the expected performance and the performance of the noncritical threads depends only loosely on the task-set. While the first noncritical thread achieves more than 50% of its single threaded performance, the relative performance of the further noncritical threads declines exponentially.

By permuting the priorities of the noncritical threads in a round robin fashion, the execution time could be distributed equally between the noncritical threads, but the overall performance is not affected and therefore is not shown in a separate figure.

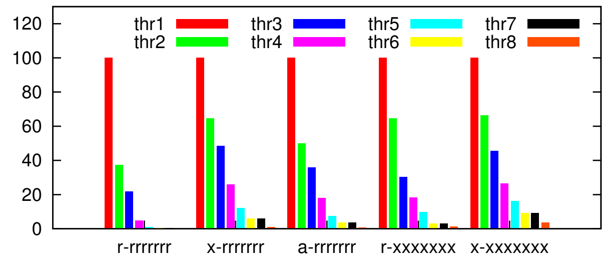

Fig. 7. Task-sets with one DMT and normalized IPCs (%) of individual thread slots

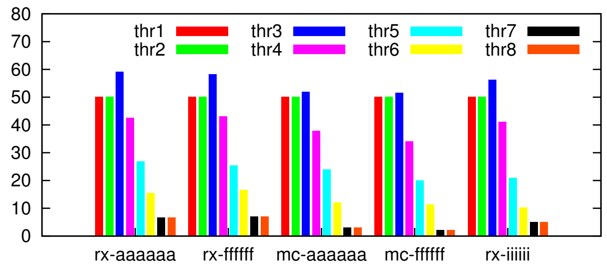

Fig. 8. Task-sets with two DMTs and normalized IPCs (%) of individual thread slots

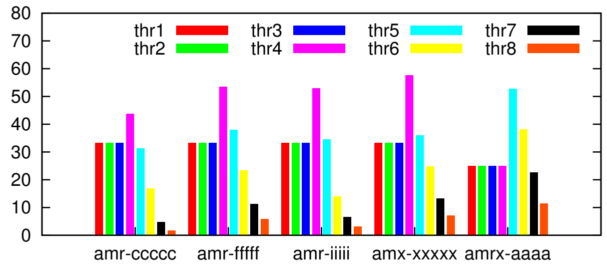

Fig. 9. Task-sets with three or more DMTs and normalized IPCs (%) of individual thread slots

VII. Conclusion

In this paper we present an SMT processor that is able to execute multiple hard real-time threads concurrently to multiple noncritical threads. The hard real-time threads are scheduled by time sharing and merged to one *Dominant Meta Thread (DMT)* that is executed as if there were no other threads, allowing a static WCET analysis. Noncritical threads are scheduled with fixed priorities concurrently to the DMT, but with lower priority to guarantee that they cannot delay the hard real-time threads. As the evaluation shows, the isolation works perfectly while three additional noncritical threads achieve about 50%, 35% and 20% of their single threaded performance, doubling the overall throughput.

References

[1] S. E. Raasch and S. K. Reinhardt, "Applications of Thread Prioritization in SMT Processors," in *Proc. of the 1999 Workshop on Multithreaded Execution, Architecture, and Compilation*, Jan. 1999.

[2] G. K. Dorai and D. Yeung, "Transparent Threads: Resource Sharing in SMT Processors for High Single-thread Performance," in *Proc. PACT'02*, Sept. 2002, pp. 30–41.

[3] R. Jain, C. J. Hughes, and S. V. Adve, "Soft Real-Time Scheduling on Simultaneous Multithreaded Processors," in *Proc. RTSS'02*, Dec. 2002, pp. 134–145.

[4] F. J. Cazorla, P. M. Knijnenburg, R. Sakellariou, E. Fernández, A. Ramirez, and M. Valero, "Predictable Performance in SMT Processors," in *Proc. of the 1st Conference on Computing Frontiers*, 2004, pp. 433–443.

[5] N. Yamasaki, I. Magaki, and T. Itou, "Prioritized SMT Architecture with IPC Control Method for Real-Time Processing," in *Proc. RTAS'07*, Dec. 2007, pp. 12–21.

[6] J. Barre, C. Rochange, and P. Sainrat, "A Predictable Simultaneous Multithreading Scheme for Hard Real-Time," in *Architecture of Computing Systems, LNCS 4934*, Feb. 2008, pp. 161–172.

[7] S. Hily and A. Seznec, "Out-Of-Order Execution May Not Be Cost-Effective on Processors Featuring Simultaneous Multithreading," in *Proc. of the 5th Int. Symposium on High-Performance Computer Architecture*, Jan. 1999, pp. 64–67.

[8] B. I. Moon, H. Yoon, I. Yun, and S. Kang, "An In-Order SMT Architecture with Static Resource Partitoning for Consumer Applications," *Parallel and Distributed Computing: Applications and Technologies, LNCS 3320*, pp. 539–544, 2004.

[9] J. Kreuzinger, A. Schulz, M. Pfeffer, T. Ungerer, U. Brinkschulte, and C. Krakowski, "Real-time Scheduling on Multithreaded Processors," in *Proc. of the 7th Int. Conf. on Real-Time Computing Systems and Applications (RTCSA'00)*, Dec. 2000, pp. 155–159.

[10] *The Ubicom IP3023 Wireless Network Processor*, Ubicom, Inc., Apr. 2003, white paper.

[11] S. Uhrig and J. Wiese, "jamuth – An IP Processor Core for Embedded Java Real-Time Systems," in *Proc. of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems*, 2007.

[12] A. El-Haj-Mahmoud, A. S. AL-Zawawi, A. Anantaraman, and E. Rotenberg, "Virtual Multiprocessor: An Analyzable, High-Performance Architecture for Real-Time Computing," in *Proc. CASES'05*, 2005, pp. 213–224.

[13] A. Anantaraman, K. Seth, K. Patil, E. Rotenberg, and F. Mueller, "Virtual simple architecture (VISA): exceeding the complexity limit in safe real-time systems," in *Proc. ISCA'03*, 2003, pp. 350–361.

[14] S. Uhrig, S. Maier, and T. Ungerer, "Toward a Processor Core for Real-time Capable Autonomic Systems," in *Proc. of the 5th IEEE Int. Symposium on Signal Processing and Information Technology*, Dec. 2005, pp. 19–22.

[15] *TriCore 1 User's Manual*, Infineon Technologies AG, Jan. 2008, v1.3.8.

[16] EEMBC, "AutoBench 1.1 Software Benchmark Data Book," http://www.eembc.com/TechLit/Datasheets/autobench_db.pdf.

[17] Mälardalen WCET research group, "WCET Bechmarks," http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

[18] HighTec EDV-Systeme GmbH, "Website," http://www.hightec-rt.com/.