DOI: 10.15439/2021F126
ISSN 2300-5963 ACSIS, Vol. 25

PRET-ization of uRISC Core

Martin Košťál
Faculty of Electrical Engineering
Czech Technical University in Prague
Email: kostama7@fel.cvut.cz

Michal Sojka
Czech Institute of Informatics, Robotics and Cybernetics
Czech Technical University in Prague
Email: michal.sojka@cvut.cz

***Abstract*—Modern safety-critical embedded systems have to be time-deterministic to guarantee safety. One source of time-nondeterminism are interrupts. This paper shows how to mitigate their influence in the system on a commercially available processor IP (Codasip uRISC) can be modified to exhibit time-determinism in real-time workloads and isolate interrupts. We extend the processor with fine-grained multithreading and isolated interrupt handling to localize time-nondeterminism caused by interrupts. We show a comparison between original and extended processors on a selection of TACleBench benchmarks. For interrupt-driven workloads, ideal interrupt isolation is achieved. The proposed modification can be used on other in-order single-issue processors.**

I. Introduction

Nowadays, Commercial-Of-The-Shelf (COTS) edge computing platforms are used in real-time applications. Their designers have to design them carefully to tolerate the time-nondeterminism of such platforms. Moreover, COTS platforms are optimized for average performance, so real-time applications require significant over-provisioning to meet timing requirements, especially in the worst case.

The Worst-Case Execution Time (WCET) bounds are given by two factors: the program and its inputs and the underlying processing architecture [1]. In this paper, we focus on processing architecture.

Modern processor architectures use many techniques to increase performance, but these are usually not time-deterministic. To name a few: instruction-level parallelism in superscalar architectures, branch prediction and speculative execution, out-of-order execution, caching and complex memory hierarchy. The problem with all these techniques is that timing depends on a complex micro-architectural state, which is often held secret from users. As a result, the execution time of a piece of code is subject to variance known as jitter.

Many researchers investigate the possibility of having a completely time-deterministic computing architecture. They propose either to modify existing architectures by replacing time-nondeterministic components with their deterministic counterparts. For example lockable [2] and partitionable caches [3], scratchpad memories [4] or time-predictable branch predictor [5]. Another approach is to develop a time-deterministic CPU, commonly referred to as PREcision Timed (PRET) machine [6], from scratch. Examples of such processors are FlexPRET [7] and Patmos [8].

This work was supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 832011 (THERMAC).

One source of time-nondeterminism in real-time applications running in COTS processors is interrupt handling. Interrupts are essential to I/O communication [9], [10] and their arrival time is often unpredictable [11].

In this paper, we focus on interrupt isolation so that the execution time of software threads not requiring the interrupts for their function is not affected. We demonstrate how a COTS processor Intellectual Property (IP) can be modified to provide multithreading with interrupt isolation to decrease execution time jitter of real-time tasks. While FlexPRET [7] implements a similar feature in a completely new architecture, we show how such a feature can be implemented by modifying an existing architecture (Codasip uRISC). Finally, we evaluate the proposed modification in terms of additional FPGA resource cost and jitter reduction, which is completely eliminated for interrupt independent tasks.

Section II describes background information, namely the PRET machine and Codasip uRISC processor. In Section III we introduce our design of PRET-like uRISC core, and in Section IV, we describe exact modifications of the core. Evaluation of our modifications based on the CPU simulator is provided in Section V, and we conclude in Section VI.

II. Background

This section describes the background information that serves as a basis for our work.

A. *PRET machine*

According to Lee et al. [12], [6] an abstract PRET machine is a machine where "Repeatable timing is more important and more achievable than predictable timing".

The authors argue that software control over timing is orders of magnitude coarser than hardware controlled timing. The software approach leads to unnecessary over-provisioning and does not allow for software and hardware independent safety certifications because the software always has to be tailored to a specific target computing platform. There are three distinguishing features that a PRET machine has:

- timing instructions
- hardware threads
- isolated interrupts

Timing instructions set and clear deadline for a task. At the beginning of task execution, a deadline is set and at the end of execution, the deadline is cleared. If the deadline is not cleared before its due time, an exception occurs, which can deal with the missed deadline.

Hardware threads eliminate pipeline bubbles caused by branching. The thread switching is implemented as fine-grained multithreading, which interleaves instructions from all threads in a round-robin fashion.

Lastly, interrupts isolation allows to assign interrupts to hardware threads. Interrupts are handled as streams of sporadic events.

B. uRISC

Codasip uRISC processor is a pipelined core, written in the CodAl architecture description language [13]. It is used mainly for technology demonstrations by Codasip and has gained popularity in academia [14], [15], [16]. The uRISC instruction set architecture (ISA) supports 46 instructions with an effective single-cycle latency. The processor architecture is 32bit wide, has 32 registers in a register file. Its pipeline has fetch, decode, execute, and writeback stages. As a modified Harvard architecture, it has separate interfaces to memory for instructions and data. The memories, as well as peripherals, are connected through the AHB3 lite bus.

III. Design

This section describes the design of our two extensions of the uRISC processor: 1) fine-grained multithreading and 2) thread independent interrupt handling.

The fine-grained multithreading helps reduce execution time jitter by effectively eliminating pipeline stalls (pipeline bubbles), which are induced by the elimination of pipeline hazards. It can be shown that fine-grained multithreading minimizes pipeline stalls by eliminating control hazards. Control hazards are eliminated because consecutive instructions do not depend on the result of previous n instructions. For a core with a four-stage pipeline, such as the uRISC, this condition is fulfilled by dispatching instructions from one thread every fourth cycle.

Issuing instructions from a thread every n^{th} cycle increases the minimal, average and worst-case execution time of tasks, but the execution time jitter is eliminated. In order to utilize the pipeline optimally, at least n threads have to be executed on a core. As shown in [12] n-thread fine-grained multithreading achieves minimal execution time jitter for real-time tasks without performance loss.

Interrupts are a source of uncertainty for the execution time of any task. It is not only the delay caused by interrupt service routine, but the changes to the state of the components such as caches or branch target buffer also affect execution time. We assume that real-world sources of interrupts can be modelled as sporadic events with a non-zero minimal time between two consecutive events. However, there could be more sources of interrupts, which can arrive at the same time. Traditional processors can deal with simultaneous interrupts in many ways. One is to use interrupt masking, which forbids servicing interrupts simultaneously; the other is nesting, which allows a higher-priority interrupt to preempt lower priority interrupt.

Isolation of interrupts removes the need for interrupt masking and nesting by assigning a single interrupt controller to a single thread. This way, tasks requiring interrupts for their functionality (e.g. I/O services) do not affect the execution of other tasks and, vice versa, are not affected by the execution of other tasks in other threads.

This work does not implement deadline instructions proposed in [12] because the same functionality can be implemented with a timer interrupt without changing the ISA.

A drawback of our design is the lack of atomic instructions in the uRISC ISA, limiting the options of inter-thread data access. We assume that every thread executes a separate program, and the programs do not communicate with each other. We plan to add atomic instructions later.

This design can be used for other in-order single-issue cores, which have one-cycle latency of all instructions. For cores with branch prediction and operand forwarding, the number of dependent instructions could be smaller than the number of pipeline stages but is never greater.

IV. Implementation

This section describes the specific implementation of PRET uRISC core and test platform, which is used for evaluation.

A. PRET extensions

The implementation of the fine-grained multithreading extends the number of simultaneously executed threads to four. Instructions from multiple threads are dispatched in a round-robin fashion, and the pipeline never contains two instructions from the same thread.

To support four threads, changes are made to the pipeline as shown in Fig. 1 by grey elements. A program counter is quadrupled, and the register file size is increased four times so that each thread has its own program counter and a portion of the register file. The whole pipeline keeps track of the thread associated with each instruction, further referenced as thread id. The fetch stage implements thread switching logic. Instructions are fetched based on a periodic round-robin schedule. Once the instruction is fetched, it is associated with a thread id. The decode stage is modified to decode operand register addresses based on the thread id so that every instruction accesses only a portion of the register file associated with its thread. It is not possible to address registers in the register file which are associated with other threads. Execute stage requires no modification; only passing of thread id is implemented. The writeback stage is modified to decode a return register address based on the thread id.

The uRISC has one interrupt signal and a fixed address for the interrupt service routine (ISR). We extend the interrupt signal to four signals, one for each thread.

The original uRISC core has four instructions related to interrupts: interrupt enable, disable, call and jump. Implementation of these instructions is modified so that interrupt enable only enables interrupts in the thread it is executed in. Likewise, interrupt disable instruction disables interrupts only for the thread it is executed in. Interrupt enable register is quadrupled.

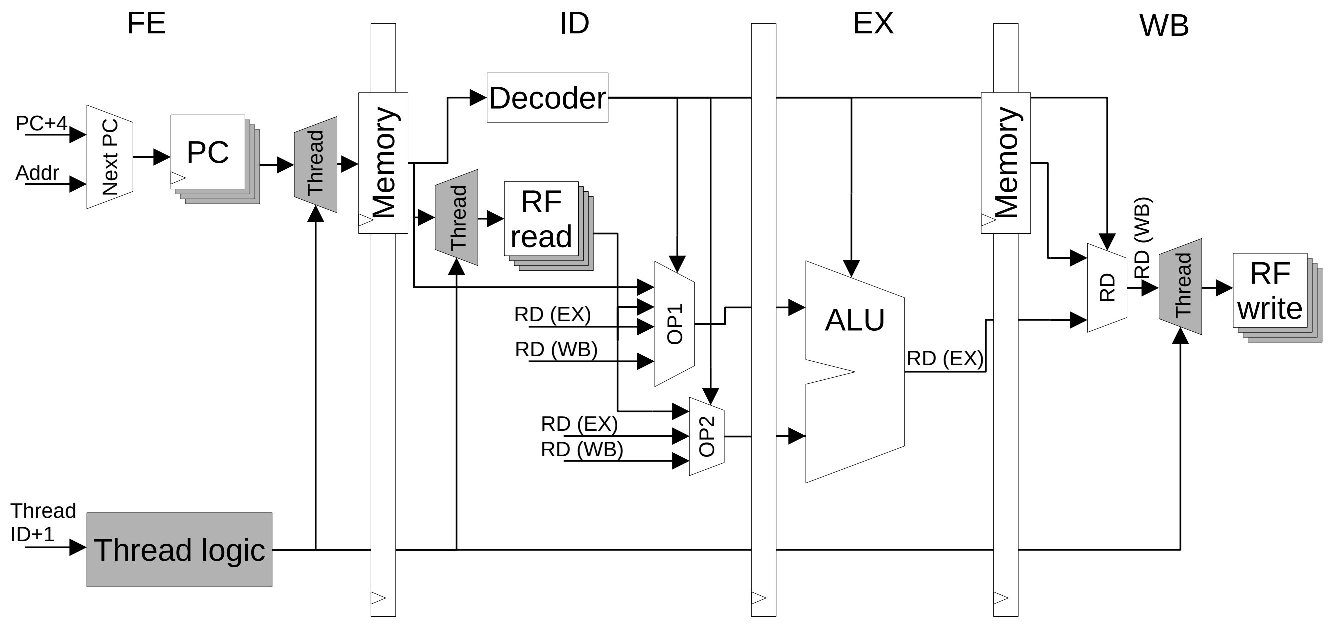

Fig. 1. Schematic of data path in modified PRET uRISC processor, grey colored components are added to the original design in order to implement multithreading

Call interrupt instruction is modified to call one of the four interrupt service routines.

Decode stage issues a call interrupt instruction if the following conditions are met: interrupt enable register is set, interrupt signal is high and both match the thread id of current instruction in decode stage. The fetched instruction of the current thread is discarded and replaced by a call interrupt, which saves the program counter of the current thread and replaces it with an address of ISR. The interrupt enable register corresponding to thread id is cleared, so no interrupt can be serviced in the thread until the ISR ends and enables interrupts by setting the interrupt enable register.

Jump interrupt instruction returns from ISR. It restores the thread to a state before the interrupt. The right program counter and interrupt enable register of the corresponding thread are set.

B. Test platform

The uRISC is a plain core. For full functionality, it is coupled with peripherals, which enables the whole platform to execute software. All peripherals are connected through the AHB3 Lite bus to the core. Any transaction on the bus takes two clock cycles, one address cycle and one data cycle. Effectively, every transaction takes only one clock cycle due to pipelining. The address is issued in execute stage and data in the writeback stage.

To fully support isolated interrupts, four programmable interrupt controllers (PIC) are present for a multithreaded platform and a single PIC for a single-threaded platform. Each PIC is coupled with a timer. The timers are original sources of interrupts on the presented platform. Both types of peripherals are connected through the AHB3 Lite interface. The peripherals support read and write access to its control registers over the bus. PICs have an additional one-bit interface for interrupt signals, which are connected directly to the core.

The memory subsystem is a crucial part of the platform. There are separate memories for the program and for data. Such a setup allows adjusting latencies for each memory independently. An approximation of complex cache memory subsystem is achieved by changing the latencies of memories which may affect the time-predictability of the platform.

TABLE I
COMPARISON OF RESOURCE UTILIZATION IN FPGA

FPGA resource type	uRISC	PRET uRISC	increase
Slice LUTs	1715	3532	206 %
Slice Registers	1416	4624	327 %
F7 Muxes	285	1117	392 %
F8 Muxes	0	512	-

TABLE II
COMPARISON OF EXECUTION TIMES ON SINGLE AND MULTITHREADED URISC

benchmark	uRISC		PRET uRISC	
	(clk)	normalised	(clk)	normalised
iir	3815	1	12052	0,79
bitcount	23161	1	73656	0,79

V. EXPERIMENTAL EVALUATION

We evaluate our modified uRISC processor in terms of increased FPGA resource allocation and in terms of real-time properties.

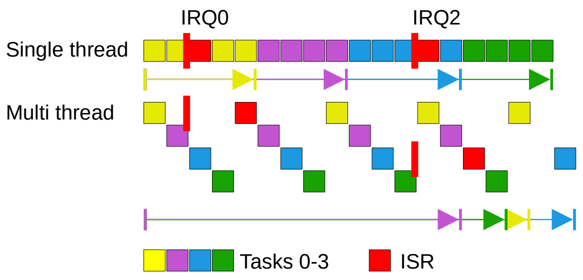

Fig. 2. Comparison of task execution on singlethread and multithreaded uRISC; Example shows task 0 (Yellow) and 2 (blue) preempted by interrupts on both platforms, completion time of task 1 (purple) and 3 (green) is prolonged by interrupts on singlethread uRISC and not influenced on multithreaded PRET-like uRISC.

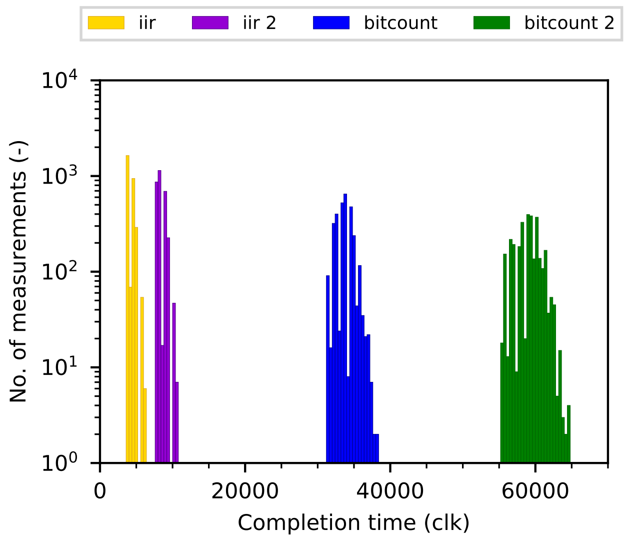

Fig. 3. Completion times of *iir* and *bitcount* benchmarks on singlethreaded core, Interrupts are enabled.

A. Resource cost

The whole platform design has been synthesized for Xilinx Artix 7 FPGA. The FPGA resource requirements loosely translate to the area. As researchers often demonstrate their designs on FPGAs, we present area requirements. Table I shows a comparison of the platforms for single and multithreaded uRISC. The multithreaded uRISC requires 206% of LUTs, 327% of registers, 392% of F7 muxes and additional 512 F8 muxes in comparison to single-threaded uRISC. If four uRISC cores were used, 400% of resources would be required, and, if it were to share the memory, memory arbitration would be required. We achieve this ratio on a simple uRISC core without branch predictor, divider or floating-point unit and essential ISA. If more complex and thus more resource-heavy processors were modified, the resource requirements ratio would be smaller. It should be noted that over 300% increase in registers is due to quadrupled register file, which could be

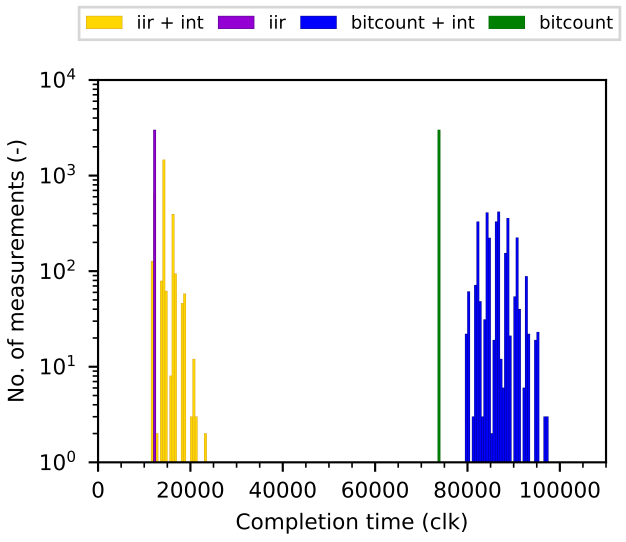

Fig. 4. Completion times of *iir* and *bitcount* benchmarks on multithreaded configuration. All four benchmarks are run simultaneously, but in a separate thread. Two threads have interrupt enabled and two disabled.

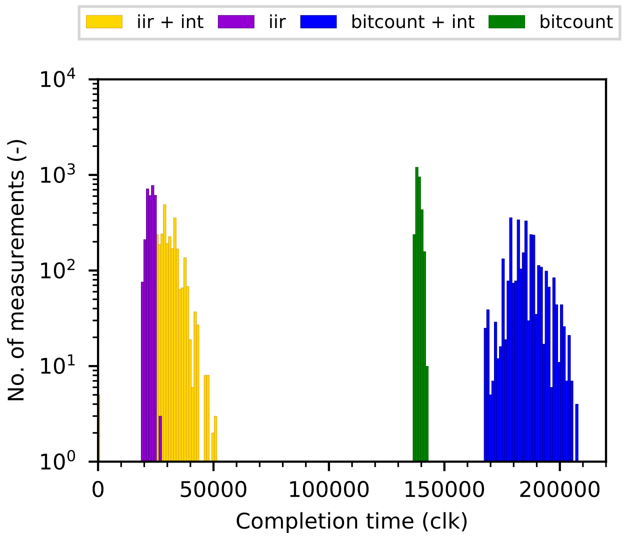

Fig. 5. Completion times of *iir* and *bitcount* benchmarks on multithreaded configuration when data memory latency is 10 clock cycles. All four benchmarks are run simultaneously, but in a separate thread. Two threads have interrupts enabled and two disabled..

mitigated by halving its size by adopting modifications similar to RISC-V extension E, which proposes to use only 16 general-purpose registers.

B. Real-time properties

We select two single-thread benchmarks from the TACleBench suite [17] to demonstrate how the real-time applications can benefit from interrupt isolation. One benchmark is *iir* and the other is *bitcount*. This selection is made to show the behaviour of tasks with short and long execution time. We do not show the rest of the benchmarks from the suite because all benchmarks are influenced by the interrupts in the same way. The benchmarks are compiled with LLVM based (Clang) Codasip compiler, which is automatically generated based on the processor description in CodAL. The compiler offers optimization presets -O0 (no optimization) through -O3 (maximal optimization). We choose to compile with optimization level set as -O2, which is commonly used by software developers and generates smaller code than -O3, which is preferred in embedded systems.

As the TACleBench does not have benchmarks, which would use interrupts, we simulate synthetic interrupts to evaluate the benefits of interrupt isolation. The interrupts are generated pseudo-randomly from the ISR. The ISR generates a pseudo-random number with uniform distribution ranging from 1 through 25 000 and sets the timer. Every benchmarking setup is run 3 000 times to get enough data to evaluate execution time jitter.

Two scenarios are benchmarked. First, for single-threaded uRISC processor, two sources of interrupts are enabled, and a simple periodic round-robin schedule of four tasks is executed, *iir*, *iir 2*, *bitcount* and *bitcount 2*. This scenario mimics a traditional approach of software threads that execute tasks non-preemptively. We measure the completion time of all tasks from the start of task 0 till the end of each task, as shown in Fig. 2. The histogram of measured execution times of this scenario is shown in Fig. 3. It is obvious that the only source of time-nondeterminism in this scenario is from ISR.

The second benchmarked scenario is a multithreaded uRISC processor with PRET-like modifications. Again four tasks are executed, but this time concurrently in four hardware threads. Two threads have interrupts enabled, and the other two execute without interrupts. In Fig. 4 we show the completion times, which directly translates to execution time for this scenario. The longer execution times caused by interrupts affect only the respective threads, and the execution of other threads remains unaffected. If we normalize the completion times of the unaffected tasks to a thread time, the execution time is shorter than on a single-threaded core, as shown in Table II. This is due to the increased efficiency of branching instructions, which no longer clear the pipeline of the consecutive instructions, as there is no dependency of consecutive instructions in the pipeline.

Next, we want to evaluate the effect of interrupt isolation under the presence of time-nondeterminism caused by increased memory access latency. Higher memory access latency is typically present in bigger CPUs. We increase the memory latency to 10 clock cycles. Therefore, each load-store instruction stalls the whole pipeline, which, with our implementation, influences all threads, not only the one accessing the memory. Fig. 5 shows the results of our benchmark in the multithreaded scenario. It can be seen that the jitter of interrupt-isolated tasks (magenta, green) is greater than zero but still significantly smaller than the jitter of interrupt-enabled tasks.

VI. Conclusion

We proposed an enhancement of an in-order single-issue processor IP by PRET-like modifications to increase time-determinism for mixed-criticality systems. We evaluated our proposal on the uRISC processor. We demonstrated the behaviour of the modified PRET uRISC on two benchmarks. We conclude that the execution time of a single task is increased proportionally to the number of hardware threads, the execution efficiency of real-time tasks is increased, and time-determinism of interrupt independent tasks can be guaranteed even under the presence of other simultaneously executing interrupt-driven tasks. The execution time jitter of interrupt independent tasks is eliminated by 100%. These results are also valid for any other single-threaded application.

The cost of our modifications lies in the increased chip/FPGA area. Specifically, when compared to the original uRISC, our implementation needs 206% of LUTs, 327% of registers, 392% of F7 muxes and additional 512 F8 muxes. This is due to multiplying the register file for storing the thread context, but those numbers are smaller than for instantiating a complete core for each thread.

Although time-determinism of the CPU is degraded when memory latencies are higher than one clock cycle, we have shown that our interrupt isolation decreases the execution time jitter even in this setting.

References

[1] T. Mitra, “INVITED time-predictable computing by design: Looking back, looking forward,” in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–4, ISSN: 0738-100X.

[2] H. Ding, Y. Liang, and T. Mitra, “WCET-centric partial instruction cache locking,” in *DAC Design Automation Conference 2012*, pp. 412–420, ISSN: 0738-100X.

[3] D. Sanchez and C. Kozyrakis, “Vantage: scalable and efficient fine-grain cache partitioning,” *ACM SIGARCH Computer Architecture News*, vol. 39, no. 3, pp. 57–68. doi: 10.1145/2024723.2000073

[4] G. Gracioli, R. Tabish, R. Mancuso, R. Mirosanlou, R. Pellizzoni, and M. Caccamo, “Designing mixed criticality applications on modern heterogeneous MPSoC platforms,” in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Quinton, Ed., vol. 133. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi: 10.4230/LIPIcs.ECRTS.2019.27. ISBN 978-3-95977-110-8 pp. 27:1–27:25, ISSN: 1868-8969.

[5] M. Schoeberl, B. Rouxel, and I. Puaut, “A time-predictable branch predictor,” in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, ser. SAC ’19. Association for Computing Machinery. doi: 10.1145/3297280.3297337. ISBN 978-1-4503-5933-7 pp. 607–616.

[6] S. A. Edwards and E. A. Lee, “The case for the precision timed (PRET) machine,” in *Proceedings of the 44th annual Design Automation Conference*, ser. DAC ’07. Association for Computing Machinery. doi: 10.1145/1278480.1278545. ISBN 978-1-59593-627-1 pp. 264–265.

[7] M. Zimmer, D. Broman, C. Shaver, and E. A. Lee, “FlexPRET: A processor platform for mixed-criticality systems,” in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. doi: 10.1109/RTAS.2014.6925994 pp. 101–110, ISSN: 1545-3421.

[8] M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, C. W. Probst, S. Karlsson, and T. Thorn, “Towards a time-predictable dual-issue microprocessor: The patmos approach,” vol. 18. doi: 10.4230/OASIcs.PPES.2011.11 p. 11.

[9] C. Sung, M. Kusano, and C. Wang, “Modular verification of interrupt-driven software,” in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. doi: 10.1109/ASE.2017.8115634 pp. 206–216.

[10] Y. Wang, L. Wang, T. Yu, J. Zhao, and X. Li, “Automatic detection and validation of race conditions in interrupt-driven embedded software,” in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. Association for Computing Machinery. doi: 10.1145/3092703.3092724. ISBN 978-1-4503-5076-1 pp. 113–124.

[11] M. Pan, S. Chen, Y. Pei, T. Zhang, and X. Li, “Easy modelling and verification of unpredictable and preemptive interrupt-driven systems,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. doi: 10.1109/ICSE.2019.00037 pp. 212–222, ISSN: 1558-1225.

[12] E. Lee, J. Reineke, and M. Zimmer, “Abstract PRET machines,” in *2017 IEEE Real-Time Systems Symposium (RTSS)*. doi: 10.1109/RTSS.2017.00041 pp. 1–11, ISSN: 2576-3172.

[13] Z. Prikryl, “Fast simulation of pipeline in ASIP simulators,” in *2014 15th International Microprocessor Test and Verification Workshop*. doi: 10.1109/MTV.2014.18 pp. 10–15, ISSN: 2332-5674.

[14] P. Sláma, “Instruction level parallelism in modern processors,” Master Thesis, BUT, 2020.

[15] M. Fajčík, “Automation of verification using artificial neural networks,” Master Thesis, BUT, 2016.

[16] M. Fajcik, P. Smrz, and M. Zachariasova, “Automation of processor verification using recurrent neural networks,” in *2017 18th International Workshop on Microprocessor and SOC Test and Verification (MTV)*. doi: 10.1109/MTV.2017.15 pp. 15–20, ISSN: 2332-5674.

[17] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener, “TACLeBench: A benchmark collection to support worst-case execution time research,” in *16th International Workshop on Worst-Case Execution Time Analysis*. doi: 10.4230/OASIcs.WCET.2016.2