Design Automation for Embedded Systems (2020) 24:45–62
https://doi.org/10.1007/s10617-019-09224-3

A simultaneous multithreading processor architecture with predictable timing behavior

Hadley Magno Siqueira[1] · Marcio Eduardo Kreutz[1]

Received: 24 May 2019 / Accepted: 31 August 2019 / Published online: 10 September 2019

Abstract
Real-time embedded systems need software and hardware to be time-predictable to guarantee the correct behavior of the system. Precision Timed Machines are architectures designed for timing predictability and repeatability. They help to improve design time and the efficiency of real-time embedded systems by allowing to separately verify the timing properties of modules. This paper presents a Simultaneous Multithreading Precision Timed Machine named Hivek-RT that can execute hard real-time and conventional threads in parallel. It employs a repeatable thread-interleaved pipeline with an exposed memory hierarchy composed of scratchpads, caches, and a predictable SDRAM memory controller. The proposed architecture is well suited for real-time embedded systems as experimentation results show that the proposed architecture has improved throughput, presents low memory footprint and achieve a memory bandwidth of 90% of the theoretical value while providing deterministic time access to the memory hierarchy. This paper is an extended version of the paper presented on the 8th Brazilian Symposium on Computing Systems Engineering.

Keywords Cyber physical systems · Precision Timed Machines · Real-time · Embedded systems

1 Introduction

Developing hard real-time software on modern processors has become very difficult because processors' complexity has made predicting timing and execution speed nearly impossible. The usage of techniques such as pipelines, bypassing, superscalar, out-of-order execution, three-level memory hierarchies with complex cache replacement policies and other techniques makes it extremely difficult to accurately predict exactly how many cycles it will take to execute a sequence of simple instructions [2], let alone code with conditional branches.

✉ Hadley Magno Siqueira
hadley.siqueira@gmail.com

Marcio Eduardo Kreutz
kreutz@dimap.ufrn.br

1 Federal University of Rio Grande do Norte, Campus Universitário Lagoa Nova, Natal, RN, Brazil

Also, these techniques demand more energy when compared to simple processors, preventing the use of such processors in embedded systems with severe energy constraints.

The development of real-time software uses the concept of Worst-Case Execution Time (WCET) to provide the maximum amount of time that a thread will need to execute. WCET bounds are the foundation on which real-time software engineering is built. While one can always be conservative with over-estimates, this has become unrealistic since the difference between hitting level one cache and accessing main memory can be a thousand cycles. To address those issues, there is a growing interest in projecting architectures with time predictability in mind. One of such works is the Precision-Timed (PRET) Machines that serves as the base for the proposed architecture: a Simultaneous Multithreading (SMT) PRET Machine targeting real-time embedded systems with memory constraints. The architecture is capable of executing a mixing of Hard Real-Time (HRT) and Non-Hard Real-Time (NHRT) hardware threads, up to a total of 16 hardware threads in parallel. To the best of the authors' knowledge, this is the first PRET machine that exploits both Thread-Level Parallelism (TLP) and Instruction-Level Parallelism (ILP).

This paper is an extended version of the paper of the same title presented on the 8th Brazilian Symposium on Computing Systems Engineering. This paper extends the previous one by providing more details of the proposed architecture and by presenting new experimental results. The rest of the paper is divided as follows: Sect. 2 presents the main ideas of PRET philosophy. Section 3 discusses related works. Section. 4 presents the proposed architecture with the experimentation and results presented in Sect. 5. Conclusions are presented in Sect. 6.

2 Precision-Timed Machines

Precision-Timed (PRET) Machines comes from Berkeley University [1]. Its main argument is that modern processor architectures have gone down an unpredictability hole due to its single-minded focus on average-case performance. The complexity of modern processors [4] has made the task of calculating or even bounding the WCET of a thread very difficult [2]. While this is not critical for best-effort computing, it is a disaster for hard real-time systems. PRET philosophy defends that it is time to consider architectures that provide timing as predictable as other functions. It goes beyond by stating that execution should also be repeatable. It wants the timing to be equally consistent, predictable and documented as other features such as arithmetic, for example.

To achieve time predictability and repeatability, PRET machines are currently implemented with a thread-interleaved pipeline and an exposed memory hierarchy. The thread-interleaved pipeline allows one to eliminate the use of bypass logic and branch prediction, for example, thus eliminating some hazards that are common in other architectures. The exposed memory hierarchy is generally implemented using scratchpads and predictable dynamic memory controllers. All the instructions in a PRET machine execute in a predictable and repeatable number of steps, including those that access main memory, providing a predictable and repeatable behavior to the program. The instruction set also presents instructions for manipulating time, allowing to set up alarms and interrupts when a deadline is missed.

3 Related work

In the last years, an effort could be observed in the community in the direction of obtaining predictable behaviors for processors dedicated to executing real-time applications.

The PATMOS [6] architecture is a VLIW processor capable of executing up to two instructions in parallel. The Instruction Set Architecture (ISA) provides a deterministic execution time. The memory hierarchy is composed of several special caches. There is a cache to store data that belongs to stack, another cache to store data allocated on the heap and so on. The idea is that each cache has an access pattern that the programmer can use to estimate the WCET of a given thread.

PTARM [5] is a multithreading PRET machine implementing the ARM ISA in a five-stage thread-interleaved pipeline capable of executing four HRT threads. Its memory hierarchy consists of scratchpads that are shared among the threads and the access to main memory goes through a predictable dynamic memory controller that exposes the Dynamic RAM (DRAM) internals, with each DRAM bank acting as a resource capable of being accessed in parallel with other banks.

FlexPRET [11] is another PRET machine. It is a fine-grained multithreaded processor designed to exhibit architectural techniques useful for mixed-criticality systems. FlexPRET supports an arbitrary interleaving of threads, controlled by a thread scheduler. It classifies each thread as either a hard real-time thread (HRTT) or a soft real-time thread (SRTT). FlexPRET provides hardware-based isolation to HRTTs while allowing SRTTs to efficiently utilize the processor. Each thread, either an HRTT or SRTT, can be guaranteed to be scheduled at certain clock cycles for isolation or throughput guarantees. If no thread is scheduled for a cycle or a scheduled thread has completed its task, that cycle is used by some SRTT in a round-robin fashion, efficiently utilizing the processor.

The main contribution of this paper is the proposal of a PRET processor with a Simultaneous Multithreading (SMT) superscalar architecture, associated with a Very Long Instruction Word (VLIW) design. The proposed architecture is capable of executing up to 16 threads while still retaining predictability and repeatability, characteristics required for PRET machines. The proposed architecture also supports the best-effort execution of non-real-time tasks, by scheduling their instructions into the pipeline whenever real-time tasks' instructions cannot be executed in parallel (e.g. due to dependencies) or are not eligible (e.g. when threads are not active). It is hence capable of exploiting both ILP and TLP and supports an increased number of tasks being processed when compared to other proposed PRET machines—a very important evolution in the direction of enabling their use in future multi-task real-time systems. To the best of the authors' knowledge, this is the first PRET machine to explore both TLP and ILP. PTARM, for instance, is only capable of exploring only TLP. FlexPRET is another PRET machine that can mix HRTT and SRTT but it also only explores TLP. Our architecture is capable of exploring both ILP and TLP because it is based on a VLIW architecture.

4 Proposed architecture

The proposed architecture presented in this work is based on the Hivek architecture [9]. Hivek is a VLIW processor targeting embedded systems with memory constraints. Hivek's current implementation employs a pipeline with seven stages with two lanes of execution. Although the current implementation relies on two execution lanes, more lanes can be added without any significant impact on the microarchitecture. It also supports instruction predication and static

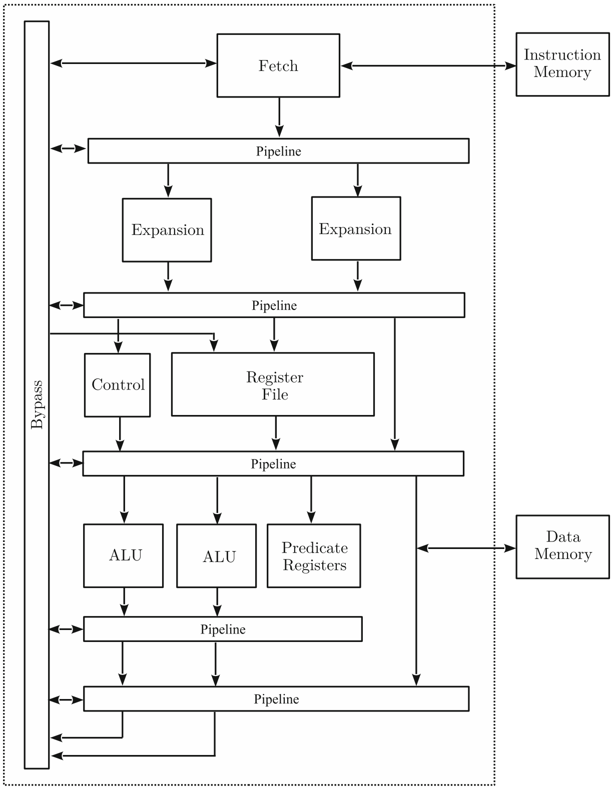


Fig. 1 Simplified view of Hivek's pipeline

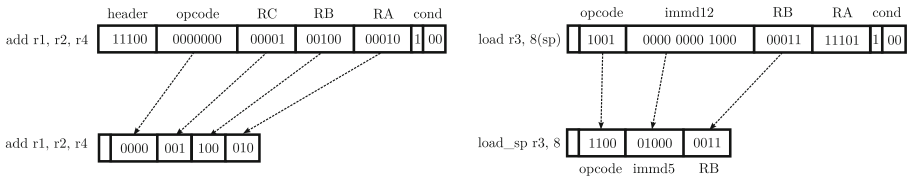


Fig. 2 An example of compact instructions

branch prediction by using hints encoded in instructions by the compiler. While traditional VLIW architectures use fixed-length instruction that can lead to poor code density, Hivek uses a variable-length instruction set to achieve high code density. Variable length instruction is a common technique used to achieve high code density and can be found on other architectures such as Itanium [7], ARM Thumb and ARM Thumb2 [10].

Figure 1 shows Hivek's current pipeline implementation. The first stage fetches two instructions and the second stage is used to handle memory latency. The third stage is responsible for expanding 16-bit instructions up to 32-bit instructions. The third stage verifies the header bits to decide how to expand the instructions. Figure 2 show how the instructions are expanded.

On the fourth stage, it decodes the two expanded instructions. The rest of the pipeline resembles a classic RISC pipeline and executes the two decoded instructions.

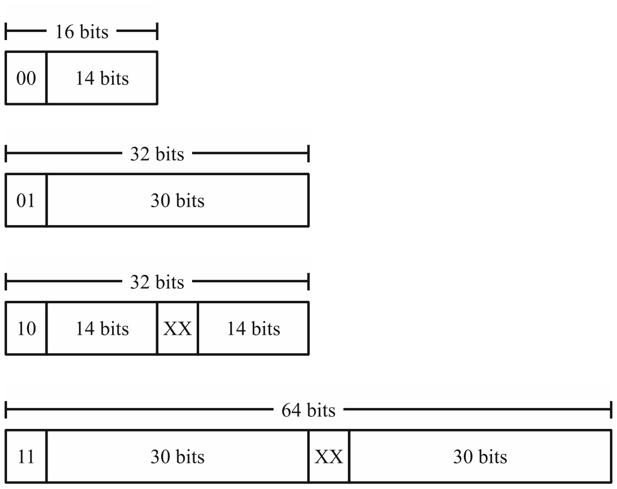


Fig. 3 Bundle formats

4.1 Instruction format

Hivek employs a custom instruction set. The ISA consists of instructions of 14-bit instructions and 30-bit instructions. The 14-bit instructions are a subset of the 30-bit instructions set. These instructions can be combined in bundles of two instructions as shown in Fig. 3. Together with the header of two bits, the instruction has a total size of 16 bits and 32 bits, respectively.

The instructions are based on a classical RISC ISA. There are operations among registers, between registers and immediates, and operations to move data around registers and memory. As expected, there are also jump and branch instructions as well.

As the program counter can be incremented by different amounts but it always fetches 64-bit leading to misaligned accesses, the ISA is supported by a special cache design that handles these misaligned accesses with a clock cycle. This cache has an internal architecture composed of four memory banks that are accessed as a kind of circular buffer. A decoder checks the requested address and generates the real addresses to be used on the four memory banks. The fetched bytes are then reordered to form the correct bundle that will be sent to the expansion stage.

Instructions can have a predication bit allowing for the conversion of control flow dependencies into data dependencies, thus reducing the number of branches. Branch instructions also have a bit for allowing static branch predictions. This bit works as a hint to indicate if the branch should be taken or not.

4.2 The Hivek-RT architecture

While Hivek is well suited for embedded systems with memory constraints because of its variable length ISA, it is not well suited for real-time embedded systems because it employs techniques such as bypassing logic, branch prediction, and others that make it difficult to estimate accurately the timing behavior of the software running on it. This way, it is proposed a modification to the original Hivek architecture, named Hivek-RT (Real-Time). Currently, PRET machines are implemented using a thread interleaved pipeline. We then convert the

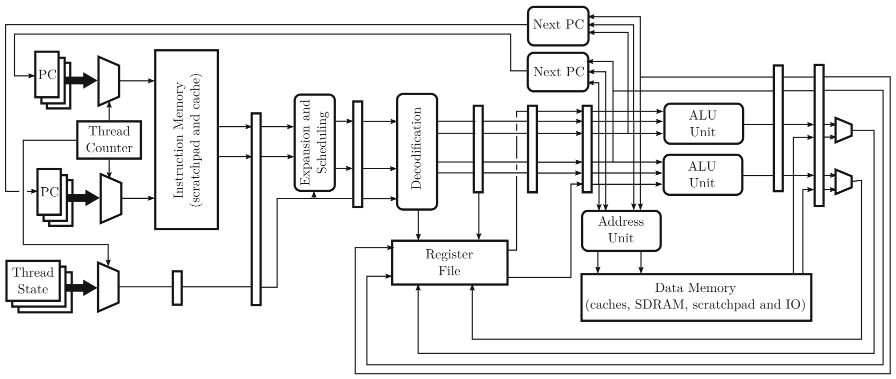


Fig. 4 Hivek-RT pipeline

VLIW Hivek to the Simultaneous Multithreading (SMT) Hivek-RT. Figure 4 shows a simplified version of the proposed pipeline.

The main idea of our architecture is that our architecture is capable of executing Hard Real-Time (HRT) threads and normal threads. Given an HRT thread T, it always tries to execute in parallel as many instructions as possible of T. If T is blocked by a long latency operation such as memory access, for example, or there isn't enough Instruction Level Parallelism (ILP), then it schedules instructions of a Non-Hard Real-Time (NHRT) thread to fill the remaining functional units. This way, the proposed architecture exploits Instruction Level Parallelism (ILP) and Thread Level Parallelism (TLP) with the HRT threads always having the highest priority of execution and NHRT threads executing when possible.

As an SMT architecture, Hivek-RT is capable of storing up to 16 threads' contexts in hardware. These 16 threads are partitioned in two subsets, each one with eight threads. The first subset contains HRT threads while the other subset contains NHRT threads. Threads are scheduled every clock cycle in a round-robin fashion: at each clock cycle, an HRT thread is always scheduled together with an NHRT thread. Our current implementation is capable of fetching up to four instructions but can execute only two instructions in parallel. The two instructions of the current HRT thread are fetched together with two instructions of the current NHRT thread. The next step is to decide which of the four instructions are going to be executed as shown in Fig. 5. These instructions are then scheduled and any eventual 16 bits instructions are converted to 32-bits one. In the next cycle, the thread pointer is updated and a new group of four instructions is fetched from the new threads.

The interleaved execution of threads (caused by the round-robin scheduling of threads) allows the elimination of bypass logic and branch prediction in the Hivek-RT. The next instruction of a given thread is only fetched after all the remaining threads are also executed, giving enough time to calculate the next-PC (eliminating control hazards generated by jumps and exceptions) and to commit results in the register-file, eliminating data hazards. Structural hazards are eliminated by avoiding the share of functional units among the threads.

The removal of hazards makes the execution of each instruction predictable. For example, in a regular architecture, a branch can take a different number of cycles depending if the branch is taken or not. This happens because, if the branch is taken, the pipeline must be refilled with new instructions. But if the branch is not taken, the next instruction is already in the pipeline. In our architecture, however, whether a branch is taken or not in thread T, the next instruction is always from another thread. Thus, when the next instruction from thread T is fetched, the next-PC is already determined, thus achieving time predictability.

Fig. 5 Scheduling of instructions

```
fetch instructions HRT1, HRT2, NHRT1, NHRT2
if HRT thread is not blocked then
    if HRT1 and HRT2 can run in parallel then
        execute HRT1 and HRT2 in parallel
    else if NHRT thread is not blocked then
        execute HRT1 and NHRT1 in parallel
    else
        execute HRT1 and a NOP
    end if
else if NHRT thread is not blocked then
    if NHRT1 and NHRT2 can run in parallel then
        execute NHRT1 and NHRT2 in parallel
    else
        execute NHRT1 and a NOP in parallel
    end if
else
    execute two NOPs in parallel
end if
```

The memory hierarchy of a PRET machine must also be time-predictable to deliver instructions and data as the processor needs them. If the memory hierarchy does not present time predictability, the processor also loses its time predictability. Hivek-RT's memory hierarchy is composed of scratchpads, caches, and a predictable SDRAM controller. Scratchpads are local memories managed by software instead of hardware, giving the programmer the freedom to choose when data (or instructions) are stored or not in the scratchpad. This has the advantage of presenting time-predictable behavior: the programmer knows if data will be there or not before accessing it.

Although scratchpads are time-predictable, it burdens the programmer to keep track of what is stored in the scratchpad and explicitly transfer data between main memory and scratchpad. Caches, on the other hand, allow for automatic verification of data presence and automatic replacement of data as needed, all transparent to the processor. But this comes with costs: caching logic is unpredictable, as the programmer does not control when data will be on cache or not. In Hivek-RT, HRT threads only access scratchpads and main memory, but not caches to avoid their unpredictable behavior. The scratchpad is shared among all threads, including the NHRT ones, allowing for efficient data transfer between threads with predictable behavior. Caches are used exclusively by NHRT threads. As these threads do not need to be time-predictable, caches are used to ease programming.

Hivek-RT uses as the main memory an SDR SDRAM attached to a predictable SDRAM controller similar to the one presented in [5]. SDRAMs store data in capacitors that need to be refreshed periodically. In traditional SDRAM controllers, refreshes are a source of unpredictability because the software is not aware when the controller is performing a refresh, leading to different access latencies. Our controller exchanges high performance by predictability by always performing a refresh before access, even when the refresh is not necessary. This means that any access to the main memory has a fixed and well-known latency. The controller also exploits the internal organization of SDR SDRAMs. These memories are partitioned in banks that can work in parallel but share control and data buses. In the current implementation, only four HRT threads can access in parallel the four banks presented in SDRAM memory. Because control and data buses are shared among the banks, the threads' accesses are interleaved in a round-robin strategy. This fits gracefully with the thread interleaved pipeline from the Hivek-RT processor. If an HRT is not accessing the SDRAM, an NHRT thread can access it.

When an HRT thread starts a transfer between scratchpad and main memory, the controller pauses any ongoing NHRT transfer. It then performs a refresh to the bank being accessed and then opens a row and accesses the desired column in a burst operation. Refresh operations are always executed at each new access, meaning that memory is refreshed more than necessary. If there is no ongoing cache data transfer, the controller issues a NOP. This behavior diminishes the available memory bandwidth but allows time-predictable access.

4.2.1 DRAM controller

In this work, we use a special DRAM controller to provide predictable and repeatable access to the main memory. On conventional memory controllers, throughput and bandwidth are prioritized. On the controller used by Hivek-RT, on the other hand, predictability and repeatability are prioritized.

To achieve predictability and repeatability, the memory controller exposes the internal DRAM structure. These memories generally have four memory banks internally. The controller exposes these banks by guaranteeing access to them in an interleaved pattern, thus each bank is accessed periodically.

DRAM memories need to be refreshed from time to time since it uses capacitors as its main storage element. One of the sources of unpredictability on conventional memory controllers is the fact that when those capacitors need to be refreshed, the controller does not inform this to the client. It instead makes the client wait for the refresh operation to finish before it can access the requested data.

Our memory controller handles this problem by using the following strategy: before each access to a bank, it performs always the same steps. First, it verifies if there is cache access going on and cancels it. It then performs a refresh operation and only then it guarantees access to the HRT thread.

Data is transferred in bursts of fixed length. At each interval of a burst to another one, a refresh operation is performed. In our current implementation, Hivek-RT runs on an FPGA at 50 MHz with the DRAM controller running at 100 MHz.

An HRT thread sees an access latency of five thread cycles as shown in Fig. 6. This figure is split into two rows: the first row shows what happens when a load instruction is executed and the second row is an example. In the first row, one can see that at cycle 0 a load instruction begins its execution. On cycle 1, it pauses any ongoing transfer that a normal thread may be doing. On cycle 2, it performs a refresh operation. Cycle 3 is responsible for opening the row and column of the requested bank in DRAM and on cycle 4 data burst begins. This way, when the programmer makes a data transfer between scratchpad to DRAM, it knows it is always going to take 5 thread cycles to data to begin to be transferred.

The second row of Fig. 6 shows an example. Instruction (a) is an addition whose result is used in the following (b) load instruction. The load instruction (b) result is used in the addition in instruction (c). Notice that there are no stalls from the software's point of view. Instead, the software sees a load instruction as a multicycle operation that takes 5 cycles. It is important to mention that the memory address space of Hivek-RT is not flat in the sense that the main memory and scratchpad memories are separated address spaces. The caches' memory address space overlaps the main memory address space as usual. This distinction is important because a memory instruction (load or store) is multicycle only when accessing the main memory address space and single-cycle when accessing the scratchpad memory address space.

As the current FPGA implementation uses a DRAM with four banks and Hivek-RT has 8 HRT, only four of the eight HRT threads are allowed to access the DRAM memory simultaneously. It is up to the programmer to handle when each thread is going to have access to

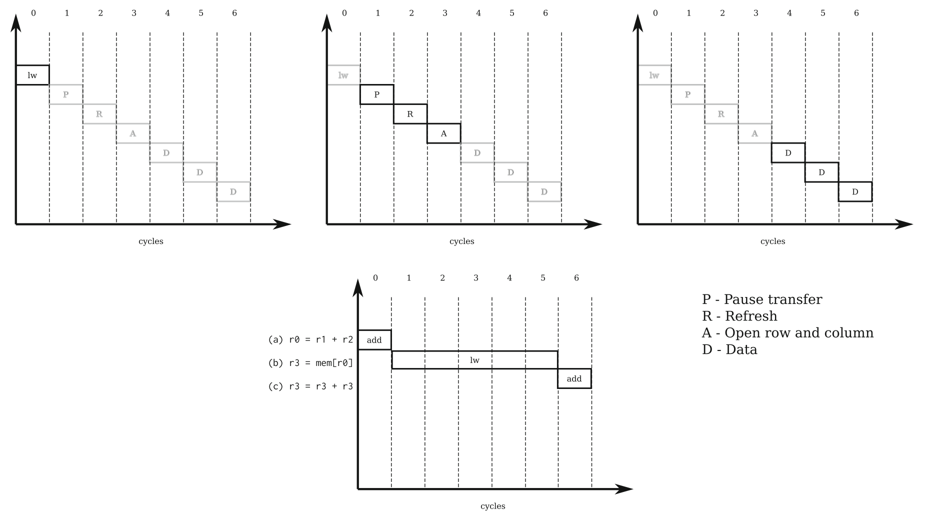


Fig. 6 Access latency to DRAM memory

Table 1 Precision timed instructions

Instruction	Description
getTime	Get the current time
delayUntil	Guarantees a minimum time of execution
exceptionOnExpired	Register interrupt routines to handle missed deadlines
deactivateException	Disable time interrupts

DRAM memory. This burdens the programmer with more complicated programming but in exchange, it gives a predictable and repeatable behavior thus satisfying PRET requisites.

4.3 PRET instructions

Hivek-RT, as PTARM, has special instructions to handle the passage of time during program execution. The instructions are presented in Table 1.

The instruction *getTime* is used to get the current time at the moment of execution of the instruction. This time comes from a counter that runs on a fixed frequency and thus serves as a clock for the processor. The time returned by this instruction can be used to make decisions on the control flow of the program.

The instruction *delayUntil* is used to guarantee a minimum amount of execution time to a certain block of instructions. This instruction receives as argument a timestamp and when executed it verifies if the timestamp matches the current time. While the timestamp is greater than the current time, the thread is stalled until the timestamp gets equal to the current time. This instruction can be used to perform synchronization tasks among threads or to guarantee that a certain block of instructions is executed with at least a minimum number of cycles.

The instruction *exceptionOnExpire* is used to implement exceptions when deadlines are missed. This instruction receives as arguments a timestamp and the address of an interrupt routine. When this instruction is executed, the arguments are registered and program execution continues. If during the program execution the registered timestamp is reached, the interrupt routine is executed. Generally, this interrupt contains code to handle the missed deadline. The last instruction is *deactivateException* and is used to disable exceptions registered using the instruction *exceptionOnExpire*.

5 Experiments and results

We compare Hivek-RT with the processors PTARM [5], StrongARM 1100 and the original Hivek. PTARM is our main architecture of reference since Hivek-RT and PTARM are PRET machines, but PTARM is a multithread machine capable of exploring only TLP while Hivek-RT is an SMT capable of exploring both ILP and TLP.

Although StrongARM 1100 is an old architecture, we decided to use StrongARM 1100 for a direct comparison with the results presented in [5] that compares PTARM and StrongARM 1100. The SimIT-ARM [8] is used to simulate the StrongARM 1100.

Clock cycles and throughput (IPC, instructions/cycle) are used as measurement units. The Mälardalen benchmark [3] suite is used to evaluate the architectures. The Mälardalen benchmark was chosen mainly because the original PTARM's article [5] used them. PTARM is the PRET processor that we compare with our proposal and we tried to reproduce as much as possible the original results. We couldn't reproduce the same results because it depends on compiler technology and on modifications made to run on bare metal but we gave our best effort to reproduce and resemble the original results.

As Mälardalen's benchmarks are single-threaded, we set up our experiments as if the same benchmark was running on all of the eight HRT threads of the Hivek-RT and twice on all four threads of the PTARM architecture. Hivek and StrongARM 1100 both execute eight times in a row on the respective simulators. This way, the total number of instructions executed on all architectures are roughly the same, while the setup mimics independent threads running in parallel.

As the architectures use different instruction sets, the benchmarks binaries were generated using GCC targeting ARMv4 architecture instruction set used by PTARM and StrongARM 1100. Hivek-RT and Hivek binaries were generated by manual translation of PTARM and StrongARM binaries: we disassembled the binaries, opened an editor and typed manually, instruction by instruction the equivalent ARM instruction to Hivek instruction. This was possible because, even though the instructions encodings are different, the instructions available in the architectures are quite similar, with the difference mainly on where and how the operands are encoded, for example, but with the same semantics. This means that almost all instructions have a one-to-one mapping. Only a few need to be synthesized with more instructions on the Hivek architectures. For example, some ARM conditional instructions need to be translated for a compare-and-branch.

To remove the impact of scratchpads or caches, the benchmarks fit entirely within the scratchpad and caches of processors and all memory accesses occur in a single cycle. This is a suitable way to guarantee that all architectures are executing almost the same binaries, thus removing eventual differences that could appear because of different ISAs, compiler technology or memory hierarchy.

Figure 7 presents the throughput achieved for each architecture when running different benchmarks. To present more realistic results, since memory with single-cycle access only exists in simulation, we performed an extra execution in Hivek-RT running as a softcore in FPGA, fetching instructions from scratchpads, caches, and SDRAM. This experiment

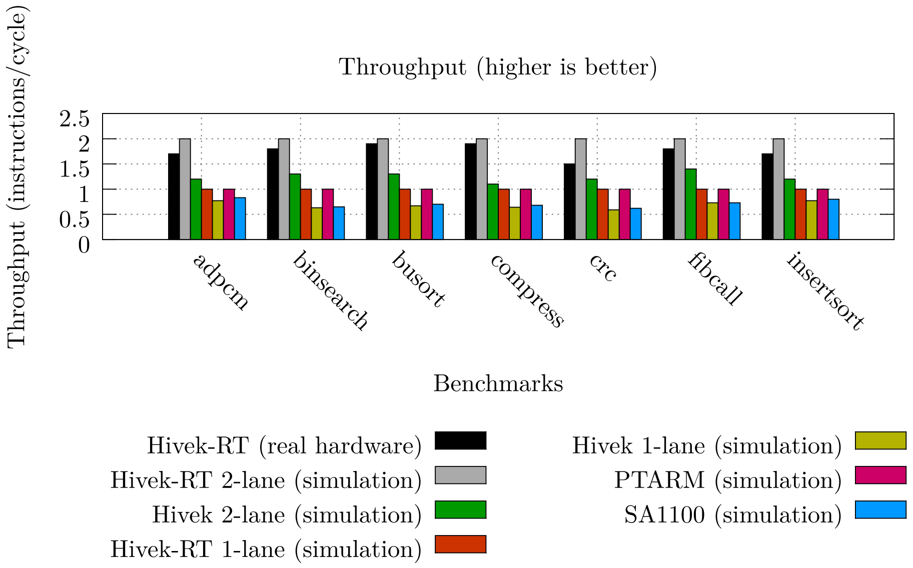


Fig. 7 Instruction throughput of each architecture

was executed only in Hivek-RT since we do not have access to real hardware of the other architectures. For the sake of brevity and ease of viewing, we put the results of Hivek-RT running on real hardware side-by-side with simulation results of the other architectures in Fig. 7. The main purpose of the results in Fig. 7 is to show that despite providing time predictability and repeatability, PRET machines can have good performance when compared to conventional architectures on the proposed experimentation setup. Results regarding the timing properties are presented in Sect. 5.1

The best result is achieved by Hivek-RT executing two instructions in parallel in a simulation environment, achieving an ideal IPC of two, as there are enough threads to be interleaved, hiding hazards' latencies. However, it is interesting to note that even when the benchmarks execute on the softcore version of Hivek-RT (thus, suffering memory performance penalties) the throughput remains better than the simulated results got of the other architectures. Although we do not have access to real hardware of the other architectures, it is straightforward to predict that similar effects on memory penalties would also be present if they were executed on real hardware.

PTARM, as a PRET machine, achieves its maximum IPC of 1. Although is a multithreading interleaved architecture capable of hiding hazards latencies, it can only commit one instruction per clock cycle because it only explores TLP. Hivek-RT, however, can explore both ILP and TLP, achieving better results even when the comparison is between real hardware versus simulation.

Although Hivek is a VLIW processor, the lack of ILP on the benchmarks resulted in IPCs lower than the ones obtained by Hivek-RT. Hivek-RT on the other hand, being a multithreaded architecture, even though the benchmark itself doesn't present enough ILP to use all functional units of Hivek-RT, the combination of several threads running in parallel are enough to fill all the functional units. As in this experiment all data and instruction were on the scratchpad, Hivek-RT was possible to achieve its maximum theoretical throughput of two instructions per clock cycle while the original Hivek wasn't. Again, such result is because Hivek-RT explores both ILP and TLP while Hivek only explores ILP.

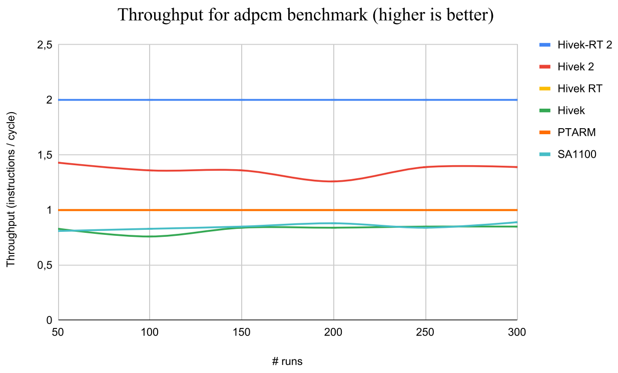


Fig. 8 Throughput for the adpcm benchmark

Table 2 Number of runs for the adpcm benchmark

Processor	#Threads	# Runs	Total
Hivek-RT	16	20	320
PTARM	4	80	320
Hivek	1	320	320
SA1100	1	320	320

The StrongARM 1100 presents the lowest IPC of all architectures. This is expected as it is a single-thread processor that doesn't explore both ILP and TLP.

Although Fig. 7 shows good performance results for various benchmarks, it fails to convince of the temporal characteristics of PRET architectures. Figure 8 shows the throughput results for the adpcm benchmark. It can be observed that PRET processors (Hivek-RT and PTARM) have straight lines while the other processors have variations.

The results of Fig. 8 were obtained by making the architectures run the same amount of times the benchmark. Table 2 shows the values. As Hivek-RT has a total of 16 threads, each running the adpcm benchmark, it took a total of 20 repetitions to total 320 times. PTARM needed to run 80 times to a total of 320 times as it has four threads. The other single-threaded processors have looped the benchmark 320 times. To discard memory effects, the entire benchmark is replicated in cache and scratchpad to avoid cache misses or to need to transfer between the scratchpad and main memory. This guarantees a hit to any instruction or data requested at any time.

The lines of PRET processors in Fig. 8 are straight because of their temporal characteristics. Because PRET processors run their threads interleaved, data dependencies of a thread can be resolved within the schedule of the next instruction, thus eliminating the need for bypass logic. The same logic applies to jump instructions. When a jump instruction is performed on a thread, the next time the thread is scheduled to execute the instruction is already available and because of this, there are no stalls in the pipeline due to jump instructions. This is different from conventional processors which, even though instructions and data are available in memory, because of data dependency and branches the processor may present variable timing.

It is important to emphasize once again that these results were obtained using a theoretical memory hierarchy that always guarantees a hit. Since the goal was to test the processors

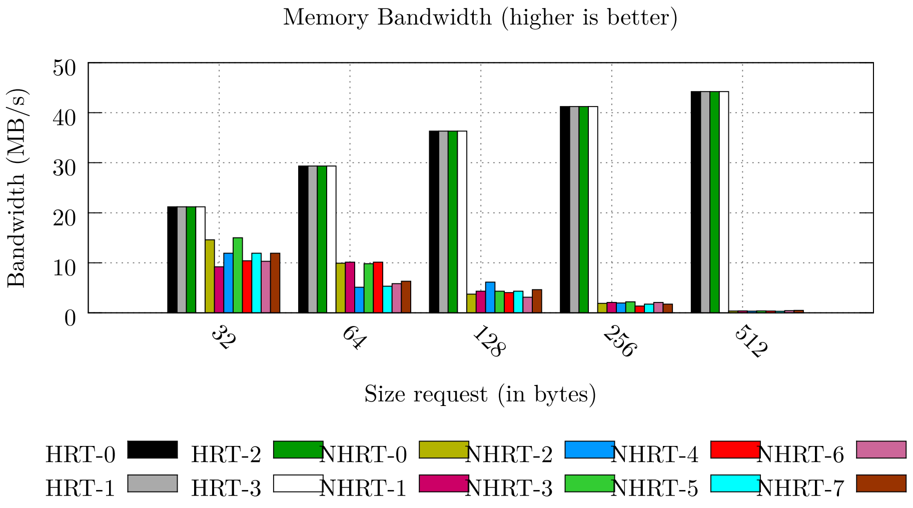


Fig. 9 Memory bandwidth available to threads

themselves, we left the memory hierarchy complexity out of this experiment. However, it is not difficult to imagine how a memory hierarchy can influence PRET processors and conventional processors. PRET processors use software-controlled scratchpad memories by default. This way, the software has full control over when the data and instructions are or are not on the scratchpad. On the other hand, conventional processors use caches and are influenced by replacements policies: sometimes the instruction will be in the cache, sometimes not. In the case of the Hivek-RT processor that uses both caches and scratchpads, they are isolated from each other, that is, the address spaces are distinct, so that one cannot influence the other. In addition, the memory controller guarantees full priority for transfers between main memory and scratchpads.

We also performed experiments to verify the bandwidth provided by the SDRAM controller. An SDR (Single Data Rate) SDRAM of 16 bits running at 100 MHz is used, giving a maximum theoretical bandwidth of 200 MB/s. Experimentation consists of synthetic programs reading in a loop bursts of 32 up to 512 bytes. Results are shown in Fig. 9 where (N)HRT-n represents the available bandwidth for the n-th (N)HRT thread.

As the SDR SDRAM used has only four banks, only four HRT threads can access the memory in parallel because of the interleaved access pattern. All the NHRT threads are free to compete for memory with the HRT threads. One can observe that the HRT threads have the same bandwidth, an indication of the interleaved access and predictability. The remaining NHRT threads have different bandwidths, showing the impact of cache usage and competition with the HRT threads. In extreme cases, NHRT can suffer from starvation, but the HRT threads are guaranteed to have available bandwidth.

In all scenarios, the average of total bandwidth measured was 180 MB/s, a value that is 90% of the maximum theoretical value of 200 MB/s, showing that the fixed latency access to main memory leads to a degradation of 10% of memory bandwidth.

One can see in Fig. 9 that the bandwidth available for the real-time threads are the same. Visually this is represented by bars of the same height. The bars have the same height because a transfer between a scratchpad and main memory is performed in a deterministic way. When the real-time thread issues a memory operation to the main memory, the access latency is constant (5 thread cycles). Then, data is transferred in a burst without suffering interruptions

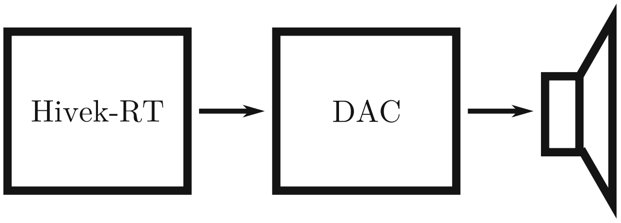


Fig. 10 Block diagram of the case study

of other threads. This is guaranteed because a real-time thread owns the internal DRAM's bank and doesn't suffer interruptions from other threads while accessing it.

5.1 A case study

We now present an example of an application running on Hivek-RT. The application consists of a PCM player that also executes a low pass filter. Figure 10 shows a block diagram of the proposed example. The processor is connected to an audio output. Audio samples are sent to the audio output by a memory-mapped register. The audio output does not perform any operation and is used only to handle the physical aspect of audio emission.

Figure 11 shows what is being executed on the processor. An HRT thread is responsible for fetching PCM data from the main memory to the scratchpad memory. A second thread runs a low pass filter on the PCM data fetched by the first thread and then sends the processed sample to the audio output.

The PCM data is sampled at 44,100 Hz, which is a common sample value. We use a mono channel with 2 bytes per sample, which means that a total of $44{,}100 \times 2 = 88{,}200$ bytes should be processed per second. This means that the audio output must receive a sample within a 23 μs interval to keep the audio at real-time without distortions caused by latencies. Our current implementation runs on an FPGA with the processor running at 50 MHz while the memory controller runs at 100 MHz. With these clocks, the processor has a total of 1133 cycles to process a sample, but as eight threads are being interleaved in the pipeline simultaneously, from the thread perspective, it has a total of $1133/8 = 141$ cycles to process a sample. With the memory controller running at 100 MHz and the processor running at 50 MHz, a thread perceives a latency of 5 clock thread cycles to begin access to the main memory and can transfer 8 bytes per thread cycle. This means that in theory, each thread has guaranteed a maximum bandwidth of 50 MB/s but in practice, this value is not achieved due to access latencies.

To analyze the first thread responsible for fetching data from memory, we present the code in Fig. 11. the first three instructions are just loading constant values to registers by using the load immediate instruction. The constant 22,560 represents the period (in nanoseconds) of the sampling frequency of 44,100 Hz. The two other constants are PCM_DATA_ADDR and FILTER_ADDR that represents, respectively, the address where the samples are stored in the main memory and the address in the scratchpad where fetched samples will be stored to be read by the thread running the low-pass filter.

After the three first instructions, a loop begins. The loop's instructions are annotated with relative timestamps to ease the discussion of the loop. At 0 ns, the current time is obtained and stored in registers r10 and r11. The next two instructions execute a 64-bit addition whose result is going to be used by a delay_until instruction. The *adc* instruction is a *add with carry*

```
                li r20, 22560
                li r2, PCM_DATA_ADDR
                li r3, FILTER_ADDR
        LOOP:
(0 ns)          get_time r10, r11 // r10,r11 = current_time
(160 ns)        add r11, r11, r20 // 64-bit addition r10,r11 = r10,r11 + r0,r20
(320 ns)        adc r10, r10, r0  // continues the 64 bit addition (adc = add with carry bit)
(480 ns)        lwp r1, 0(r2)     // loads a sample from the main memory (that's why it is lwp and not lw)
(1440 ns)       sw r1, 0(r3)      // saves the sample on the scratchpad to be read by the low-pass filter
(1600 ns)       addi r2, r2, 4    // advances to the next sample
(1760 ns)       delay_until r10, r11 // delays execution until next sample
(22560 ns)      jmp loop
```

Fig. 11 Code of the first thread

bit. The add instruction is executed at 160 ns while the *adc* instruction is executed at 320 ns. These values come from our current implementation that runs at 50 MHz and each thread perceives a clock of $50/8 = 6.25$ MHz or a period of $1/50/8 = 160$ ns.

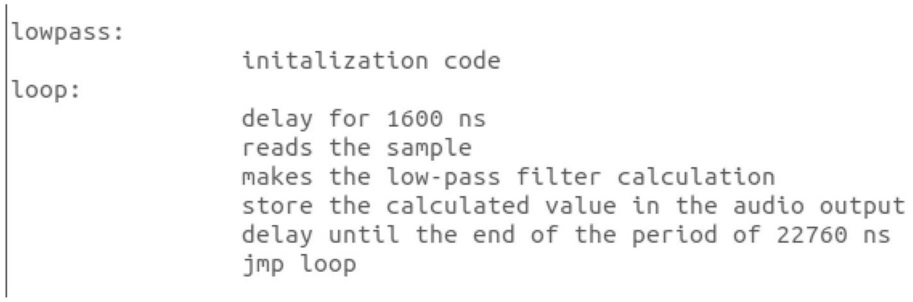
The next instruction is *lwp* which is a load word from main memory and is executed at 480 ns. The p in *lwp* works as a suffix. A *lw* instruction fetches data from the scratchpad while *lwp* loads data directly from the main memory. As the instruction is reading from memory, there are 5 thread cycles of access latency plus one more thread cycle to transfer the sample itself, making a total of 6 thread cycles to transfer a sample from the main memory to the processor. Here we note that we could easily make burst transfers to transfer more samples per transaction, but the idea here is to stress the main memory access and to keep code simple so that it can be discussed.

The next instruction is a store instruction. It stores the sample fetched in the scratchpad memory so the sample can be read by the low-pass filter thread. This instruction is executed at 1440 ns which because of the 6 thread cycles latency of the *lwp* instruction (5 cycles because of latency and 1 cycle transferring the sample). The next add instruction advances the address to the next sample and then *delay_until* instruction is executed.

The *delay_until* receives two registers as arguments. The two registers form a total of 64-bit timestamp. Here we explain the logic of the 22,560 constant mentioned above. The sampling frequency of 44,100 Hz has a period of 22,675.73 ns. As each thread has a period of 160 ns, the most closely multiple of 160 is 22,720. This means that when the *delay_until* instruction begins its execution, it has passed already 1760 ns from the beginning of the loop, leaving $22{,}720 - 1760 = 22{,}560$ ns. This way, the *delay_until* instruction pauses the thread until 22,560 ns has passed when then the *jmp* instruction is executed completing the total of 22,720 ns which is the closest value to the period of 22,675.73 ns of the sampling frequency of 44,100 Hz.

The low-pass filter is not shown here for brevity's sake. As the low-pass filter algorithm works with real values and our current implementation doesn't have floating-point units, all the real math was implemented with fixed pointed math that made the code verbose. But we discuss a pseudo-code of it. Its core is the application of the formula $y = (a * y) + (x - (a * x)$ where y represents the output of the filter while x is the input sample. the variable a is a correction factor that comes from the formula $a = dt/RC$ where dt is a discrete-time value and RC a value obtained from a physical RC low-pass filter. Figure 12 shows the assembly pseudo-code of the low pass filter. After the initialization code, the algorithm enters a loop. The first delay instruction is used to synchronize with the first thread. If one pays attention to the code of the first thread, it will see that the fetched sample is available for sure after 1600 ns because the store instruction was already executed. This way, we use the passage of time itself to make synchronization between the threads without the need to use classical mechanisms such as mutex, for instance.

After the sample is read, the low-pass filter calculation is executed and the result sent to the audio output by using a store instruction. The store writes on a memory-mapped register



```
lowpass:
                initalization code
loop:
                delay for 1600 ns
                reads the sample
                makes the low-pass filter calculation
                store the calculated value in the audio output
                delay until the end of the period of 22760 ns
                jmp loop
```

Fig. 12 Code of the lowpass filter thread

that serves as the input to a Digital to Analog Converter (DAC) that generates the analog audio signal. After the store, the thread is stalled by a delay instruction until it ends the period of the current sample.

To stress the architecture, we have used the maximum number of hard real-time threads and non-hard real-time threads. We named real-time threads as HRT0, HRT1 and so on and normal threads as T0, T1 and so on. We partition each set in pairs. One thread fetches the sample and the other executes the low-pass filter. For example, HRT0 makes a pair with HRT1 with HRT1 executing the code to fetch the sample and HRT0 executing the low-pass filter. The same reasoning applies to the other threads (HRT2 makes pair with HRT3, T0 makes pair with T1 and so on). We added a custom circuitry to the processor to monitor when the output of the low-pass filter was written on the DAC. Figure 13 shows the results where the x-axis represents the passage of time while the y-axis is the frequency at which a sample is written to DAC. One can see that all real-time threads write the value at the same time while normal threads suffer from varying latencies.

Because all real-time threads overlap, in Fig. 13 they appear as a green line. The threads HRT0, HRT2, HRT4, and HRT6 are the ones that write to the audio output while threads HRT1, HRT3, HRT5, and HRT7 are the ones that fetch the sample from main memory. The same reasoning applies to the normal threads. Normal threads suffer from different frequencies because they dispute the main memory and pipeline with the hard real-time threads. As the HRT threads have priority over the normal ones, the normal threads have different execution time that leads to different frequencies.

Figure 14 shows the waveforms of when the sample is written to DAC. Each spike means that a sample was written to DAC. The red rectangle contains the waveforms of the real-time threads while the green rectangle contains the waveforms of the normal threads. One can see that the spikes of the real-time threads are evenly spaced (in the thread itself and among them) while the spikes of the normal threads present variations (in the thread itself and among them). The evenly spaced spikes from real-time threads show that the deadline is being respected, leading to the generation of the sound at the correct frequency.

6 Conclusion and future work

This paper presented a Simultaneous Multithreading PRET processor targeting real-time systems that execute Hard Real-Time (HRT) threads and normal threads in parallel. Hard real-time threads execute in a predictable and deterministic way exploring ILP when possible. When there isn't enough ILP in an HRT thread, a normal thread is scheduled together to

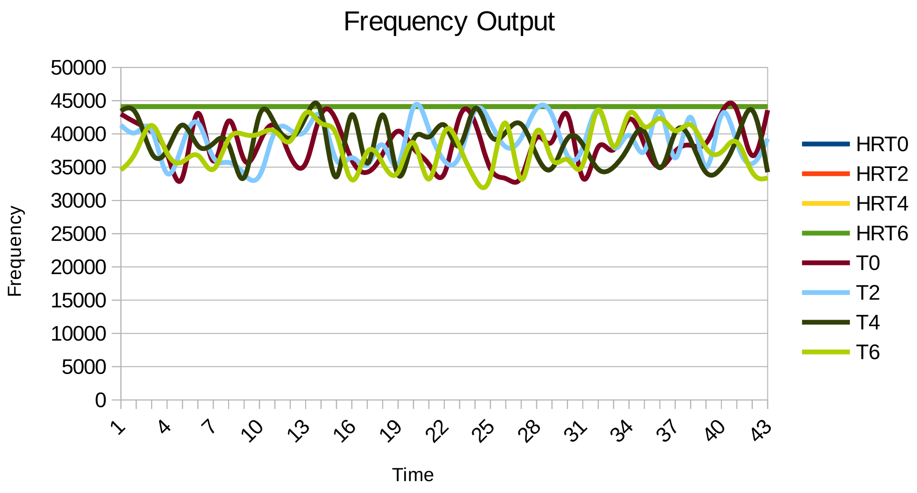


Fig. 13 Frequency results

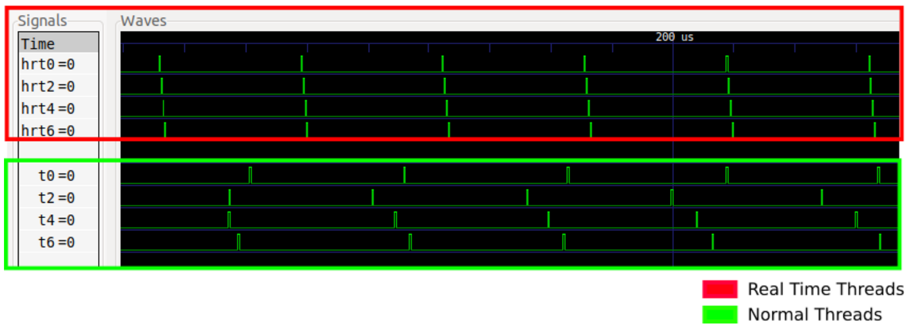


Fig. 14 Sample sent to DAC

achieve Thread Level Parallelism (TLP). The memory hierarchy also presents predictable and deterministic behavior that helps its usage in real-time systems. Experimentation showed that the proposed architecture presents a good IPC ratio as well as memory bandwidth.

As future work, we intend to modify the proposed architecture to be capable of executing soft real-time threads in a way similar to the one proposed by FlexPRET [11]. Hivek-RT's current implementation is capable of executing a mixing of hard real-time threads and non-real-time threads. With a few modifications, it is possible to allow also for the execution of soft real-time threads. The processor is also intended to be used in the implementation of a multicore architecture that follows the PRET principles and also on experimentation with Coarse-Grained Reconfigurable Architectures (CGRAs).

References

1. Edwards SA, Lee EA (2007) The case for the precision timed (PRET) machine. In: Proceedings of the 44th annual design automation conference, DAC'07. ACM, New York, NY, pp 264–265. 10.1145/1278480.1278545. http://doi.acm.org/10.1145/1278480.1278545

2. Ferdinand C, Heckmann R, Langenbach M, Martin F, Schmidt M, Theiling H, Thesing S, Wilhelm R (2001) Embedded software: first international workshop, EMSOFT 2001 Tahoe City, CA, USA, 8–10 Oct 2001 proceedings. Chap. Reliable and Precise WCET Determination for a Real-Life Processor. Springer, Berlin, pp 469–485
3. Gustafsson J, Betts A, Ermedahl A, Lisper B (2010) The Mälardalen WCET benchmarks: past, present and future. WCET 15:136–146
4. Hennessy JL, Patterson DA (2011) Computer architecture: a quantitative approach. Elsevier, Amsterdam
5. Liu I, Reineke J, Broman D, Zimmer M, Lee E et al (2012) A PRET microarchitecture implementation with repeatable timing and competitive performance. In: 2012 IEEE 30th international conference on computer design (ICCD), pp 87–93
6. Schoeberl M, Schleuniger P, Puffitsch W, Brandner F, Probst CW (2011) Towards a time-predictable dual-issue microprocessor: the patmos approach. In: Lucas P, Thiele L, Triquet B, Ungerer T, Wilhelm R (eds) Bringing theory to practice: predictability and performance in embedded systems, OpenAccess series in informatics (OASIcs), vol 18, pp 11–21. Schloss Dagstuhl–Leibniz–Zentrum fuer Informatik, Dagstuhl.https://doi.org/10.4230/OASIcs.PPES.2011.11. http://drops.dagstuhl.de/opus/volltexte/2011/3077
7. Sharangpani H (1999) Intel® Itanium^{TM} processor microarchitecture overview. In: Microprocessor forum
8. Simit-ARM. http://simit-arm.sourceforge.net/. Accessed Aug 2019
9. Siqueira H, Correa E, Silva I, Kreutz E, Pereira M (2014) A VLIW architecture with memory optimization. In: Proceedings of IBERCHIP XX workshop
10. Vijay JV, Bansode B (2015) ARM processor architecture. Int J Sci Eng Technol Res 4:3385–3387
11. Zimmer M, Broman D, Shaver C, Lee EA (2014) Flexpret: a processor platform for mixed-criticality systems. In: 2014 IEEE 20th real-time and embedded technology and applications symposium (RTAS), pp 101–110