FlexPRET: A Processor Platform for Mixed-Criticality Systems

Michael Zimmer*, David Broman*†, Chris Shaver*, Edward A. Lee*
*University of California, Berkeley, CA, USA
†Linköping University, Sweden
{mzimmer, broman, shaver, eal}@eecs.berkeley.edu

***Abstract*—Mixed-criticality systems, in which multiple tasks of varying criticality execute on a single hardware platform, are an emerging research area in real-time embedded systems. High-criticality tasks require spatial and temporal isolation guarantees for independent verification, and the task set should efficiently utilize hardware resources. Hardware-based isolation is desirable but often underutilizes hardware resources, which can consist of multiple single-core, multicore, or multithreaded processors. We present FlexPRET, a processor designed specifically for mixed-criticality systems by allowing each task to make a trade-off between hardware-based isolation and efficient processor utilization. FlexPRET uses fine-grained multithreading with flexible scheduling and timing instructions to provide this functionality.**

I. Introduction

A current trend in real-time embedded systems, driven by size, weight, power, and cost concerns, is consolidating many increasingly complex software tasks onto fewer hardware platforms. A single processor must then execute multiple tasks with differing importance, safety, or certification requirements—creating a *mixed-criticality system* [1], [2], [3]. These requirements are often specified using criticality levels, such as the five levels (A-E) used in the DO-178C avionics standard [4]. A task is considered *hard real-time* if it must never miss a deadline, and *soft real-time* if deadline misses may be acceptable. In this paper, we assume each criticality level has a requirement regarding deadline importance, with high-criticality tasks classified as hard real-time and low-criticality tasks classified as soft real-time.

Spatial and temporal isolation prevent an independent task from being adversely affected by another task: spatial isolation protects a task's state (stored in registers and memory), and temporal isolation protects a task's desired timing behavior. The *timing predictability* of an isolated task facilitates tight bounds on worst-case execution time (WCET) to avoid over provisioning resources. These are desirable task properties for verifying hard real-time tasks, but can be sacrificed for more efficient processor utilization with soft real-time tasks.

Hardware-based isolation that uses one processor per task is robust and used in safety-critical systems [5], but often utilizes hardware resources inefficiently [6]. As a consequence, extensive research has been performed during the past several years on software scheduling of mixed-criticality systems [1], [2], [7], [8], [9], [10], where *software-based* isolation is provided by a real-time operating system (RTOS). Although this can drastically reduce hardware costs, the RTOS itself must be verified and certified. This includes accounting for the overhead and behavior of run-time monitoring and control, which could include preemption to switch tasks, monitor execution times, or handle sensing and actuation [11]. In this paper, we take a different approach and investigate if a new processor architecture can provide hardware-based isolation to mixed-criticality systems without underutilizing resources.

Hardware-based isolation can be achieved by deploying *each* task on *separate* computational components: either processors, cores in a multicore processor, or hardware threads in a multithreaded processor. A multithreaded processor uses hardware support to share the pipeline between multiple hardware threads, subsequently referred to as just threads. If a task's computational requirement is not substantial, being limited to one task per core can greatly underutilize resources. One task per thread can better utilize resources by allowing multiple tasks to execute on a single processor, but thread scheduling must preserve hardware-based temporal isolation, which is uncommon in many multithreaded processors.

Fine-grained multithreaded processors [12], [13] interleave instructions from different threads in the pipeline every cycle. Some existing fine-grained multithreaded processors can preserve isolation, but they have inflexible thread scheduling mechanisms. PTARM [14] isolates each thread, but exactly four threads must be constantly active to fully utilize the processor, whereas XMOS X1 [15] can fully utilize the processor if at least four threads are active, but variation in the number of active threads reduces temporal isolation.

This paper presents *FlexPRET*, 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 novel thread scheduler. By classifying 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.

This work was supported in part by the Industrial Cyber-Physical Systems Center (iCyPhy, supported by IBM and United Technologies), the Swedish Research Council (#623-2011-955), and the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley (supported by the National Science Foundation, NSF awards #0720882 (CSR-EHS: PRET), #1035672 (CPS: Medium: Timing Centric Software), and #0931843 (CPS: Large: ActionWebs), the Naval Research Laboratory (NRL #N0013-12-1-G015), and the following companies: Bosch, National Instruments, and Toyota).

Current research in scheduling mixed-critically systems observes that using the same WCET estimate for all criticality levels is too conservative [1], [2], [8]. Even with isolation, a task's WCET can be difficult to bound; the predictability of the underlying processor depends on architectural features, such as branch prediction, pipeline ordering, and caches. If an instruction's execution time depends largely on execution history, the processor lacks *fine-grained predictability*. The problem is that complex branch prediction and multilevel caches, used to optimize average-case performance, reduce fine-grained predictability and make WCET analysis a non-trivial task [16]. FlexPRET is influenced by previous precision-timed (PRET) machines [14], [17], [18], where fine-grained multithreading uses concurrency to reduce the performance penalty of removing dynamic branch prediction and caches.

FlexPRET also uses a variation of timing instructions [17], [18], [19] to provide direct control over timing (in nanoseconds). With these instructions, the processor itself is responsible to execute with the timing specified by the instruction—enabling a more direct temporal mapping between languages with timing semantics and the processor. Unlike previous work, cycles are not left unused to satisfy temporal constraints; FlexPRET reallocates these cycles to SRTTs.

Specific contributions of this paper include:

- A novel processor design exhibiting architectural techniques targeted for mixed-criticality systems, providing hardware-based isolation to hard real-time threads while allowing soft real-time threads to efficiently utilize processor resources. (Sections III-B, III-D, III-C).
- Timing instructions, extending the RISC-V ISA [20], that enable cycles to be reallocated to other threads when not needed to satisfy a temporal constraint (Section III-D).
- A concrete soft-core FPGA implementation, evaluated for resource usage, and the execution of two mixed criticality task sets to demonstrate properties and a possible scheduling methodology (Section IV).

II. Motivating Example

In this section, we use a simple mixed-criticality example to demonstrate some useful properties of FlexPRET. Although in this example each task is deployed on a separate thread, in practice, software-based scheduling can be used to deploy multiple tasks on a single thread; the deployment of a large mixed-criticality task set on FlexPRET is shown in Section IV-D.

Consider a mixed-criticality system that consists of three independent, periodic tasks, τ_A, τ_B, and τ_C, where each task τ_i has a deadline equal to its period T_i. Each task executes on its own thread, with hard real-time tasks τ_A and τ_B on HRTTs and soft real-time task τ_C on a SRTT. Consequently, FlexPRET's thread scheduler ensures hard real-time tasks are executed at a constant rate for isolation and predictability; when a cycle is not used for a hard real-time task, which includes when a task finishes early, that cycle is used by a soft real-time task.

In Figure 1, a potential execution trace for a single hyperperiod ($t = 10,000$ processor cycles is the least common multiple of task periods) is shown, where rectangles indicate a task is executing. The upper plot is unusual in that the values on the horizontal axis (processor cycles) are restricted to multiples of four, with the vertical direction indicating from which thread an instruction is fetched each cycle over a four cycle interval; this makes thread scheduling changes easier to visualize. The sequence of thread interleaving is read top to bottom, left to right, as demonstrated for two intervals by the lower plot.

In each four cycle interval, τ_A is allocated the first and third cycle, τ_B the second cycle, and the fourth cycle is unallocated. Initially, both τ_A and τ_B execute during their allocated cycles and τ_C uses the unallocated cycles. When τ_B completes ($t = 2,000$), its allocated cycles are not needed until its next period ($t = 5,000$), so τ_C uses these cycles, temporarily executing every other cycle. Notice that τ_A's scheduling is unchanged; as a hard real-time task, τ_A would be verified to meet all deadlines with only its allocated cycles and does not benefit by completing earlier. When both τ_A and τ_B complete ($t = 8,000$), τ_C temporarily uses every cycle. By only using allocated cycles, τ_A and τ_B are temporally isolated and can be verified independently. Task τ_C efficiently uses every cycle not needed by τ_A or τ_B, but has sacrificed temporal isolation: timing behavior depends on when τ_A and τ_B start and end.

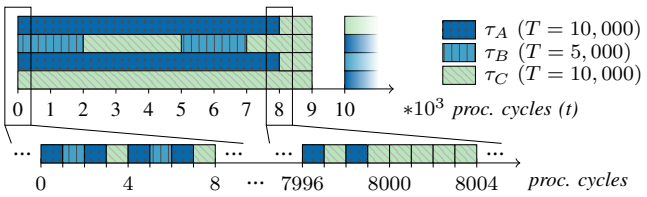


Fig. 1: FlexPRET executing a simple mixed-criticality example. Vertical direction shows from which thread an instruction is fetched each processor cycle over a four cycle interval.

III. FlexPRET Design

FlexPRET is a 32-bit, 5-stage, fine-grained multithreaded processor with software-controlled, flexible thread scheduling. It uses a classical RISC 5-stage pipeline: instruction fetch (F), decode (D), execute (E), memory access (M), and writeback (W). Predict not-taken branching and software-controlled local memories are used for fine-grained predictability. It also implements the RISC-V ISA [20], an ISA designed to support computer architecture research, that we extended to include timing instructions.

A. Background

Fine-grained multithreading is the ability to fetch instructions from different *hardware threads* each clock cycle, allowing instructions from multiple hardware threads to be interleaved in the pipeline. Each thread maintains its own state: general-purpose registers, program counter, and other control registers. The *thread scheduler* decides from which thread to fetch an instruction each cycle and will be discussed in Section III-C. A pipeline hazard occurs when continuing to execute a particular instruction in the next clock cycle could cause incorrect behavior and can be prevented by stalling (waiting) or flushing (aborting) this instruction—wasting cycles. If multiple threads are interleaved in the pipeline, the

previous instruction has progressed further through the pipeline when the next instruction from that thread is fetched, reducing or eliminating cycles that are wasted to prevent hazards by increasing the *spacing* between dependent instructions. Such interleaving increases overall processor throughput (total number of instructions processed on all threads), but increases the latency (total processor cycles between start and finish) of computing a task, compared to if the tasks were executed on a single-threaded processor.

Example 1: Consider a single-threaded processor executing a branch instruction that should be taken. This particular processor does not calculate the branch decision and target address until the end of the execute stage, so two fetch cycles (2 and 3) are wasted (flushed) if a branch is taken.

TID	Addr.	Inst.	Cycle							
			1	2	3	4	5	6	7	8
0	0x00	BR 0x0C	F	D	E	M	W			
0	0x04	I		F	D	-	-	-		
0	0x08	I			F	-	-	-	-	
0	0x0C	I				F	D	E	M	W

The thread ID (TID) column shows that each cycle an instruction is fetched from the same thread (0). The instruction and address columns show example instructions and their address in memory, where BR 0x0C means branch to address 0x0C, and I is an arbitrary instruction. Dashes indicate an instruction was flushed (instructions at 0x04 and 0x08).

Example 2: Now consider the same program running on a fine-grained multithreaded processor sharing the pipeline with three other threads in a round-robin fashion. The thread (0) is not scheduled again until after the branch decision and target address are calculated, so no cycles are wasted, but the thread has a larger latency.

TID	Addr.	Inst.	Cycle							
			1	2	3	4	5	6	7	8
0	0x00	BR 0x0C	F	D	E	M	W			
1	0x30	I		F	D	E	M	W		
2	0x60	I			F	D	E	M	W	
3	0x90	I				F	D	E	M	W
0	0x0C	I					F	D	E	M

In single-threaded processors, switching to a different task involves a *context switch*, saving the state of one task and restoring the state of another, a time-consuming operation performed entirely by software unless the processor provides hardware support. If each task is assigned to a different thread, a fine-grained multithreaded processor is capable of context switching every clock cycle. In addition to reduced overhead when switching between tasks, this also allows low-latency reactions to external IO; a task can start reacting within a few cycles instead of waiting for a RTOS to context switch.

B. Pipeline

FlexPRET allows an arbitrary interleaving of threads[1] in the pipeline (i.e. no restrictions on the schedule) to enable flexible thread scheduling. Unfortunately, this also means the pipeline is more susceptible to data and control hazards, which can occur when the spacing between two instructions from the same thread is too close. For example, the thread scheduler could schedule only one thread to be executed in the pipeline, and two instructions would need to be flushed when a branch is taken (as occurred in Example 1).

As in a typical single-threaded RISC pipeline, FlexPRET avoids most data hazards with forwarding paths, which supply required data from later pipeline stages to avoid waiting until it is written back to the register file. The only difference is that thread IDs must also be compared so that forwarding only occurs between instructions from the same thread. There are still hazards that cannot always be avoided with forwarding because the required data is not yet computed, such as a data hazard with memory load or a control hazard with a jump or branch taken. Unlike a typical single-threaded RISC pipeline, stalling and flushing must be carefully performed as to not disrupt the schedule, which would reduce temporal isolation. Stalling is done by replaying the instruction in the thread's next scheduled slot, and flushing (decision made by execute stage) is only done on instructions in the fetch or decode stage with the same thread ID.

The spacing required between two particular instructions from the same thread to prevent hazards depends on both the ISA and how it is implemented. For FlexPRET, if a jump or branch occurs, the subsequent two processor cycles must not execute an instruction from that thread, which could require the flush operation just described. Memory loads and stores occur in a single processor cycle, but in the pipeline stage after the execute stage; if the execute stage needs the result of a memory read (e.g. to perform an arithmetic operation), these instructions must not be scheduled next to each other. Even though the number of scheduled processor cycles required to execute a sequence of instructions varies with scheduling, this number *is still predictable*—it can be exactly computed for any sequence of instructions if the scheduling is known.

Example 3: Consider FlexPRET executing a schedule that alternates between two threads. Only one instruction (at 0x04) needs to be flushed when thread 0 branches, and in thread 1, forwarding allows the *ADD* instruction to use the result of the *LD* instruction without stalling.

TID	Addr.	Inst.	Cycle								
			1	2	3	4	5	6	7	8	9
0	0x00	BR 0x0C	F	D	E	M	W				
1	0x30	LD		F	D	E	M	W			
0	0x04	I			F	-	-	-	-		
1	0x34	ADD				F	D	E	M	W	
0	0x0C	I					F	D	E	M	W

Some fine-grained multithreaded processors, such as the XMOS XS1 [15] and PTARM [14], do not support an arbitrary interleaving of threads. They require a sufficient spacing between instructions from the same thread as to not require forwarding, stalling, or flushing, saving the area cost of these mechanisms. This is overly restrictive for some applications—there must be at least four threads active (for instance) to fully utilize the pipeline and a single thread cannot be scheduled more frequently than once every four cycles. By allowing an arbitrary interleaving, FlexPRET allows a trade-off between overall throughput and single thread latency. If a deadline needs to be met, a thread can be scheduled more frequently, but could waste more cycles preventing hazards.

[1]The physical number is a hardware decision; we support 1-8 threads.

C. Thread Scheduling

The pipeline supports an arbitrary interleaving of threads by using knowledge of instructions in the pipeline to forward or stall to prevent data or control hazards. Without any restrictions on thread scheduling, however, it is difficult to predict how many *thread cycles* (cycles a thread is scheduled) it would require for a thread to execute a sequence of instructions because pipeline spacing between them can vary unpredictably.

Definition 1: If the *scheduling frequency* of a thread is $1/X$, it executes exactly once every X processor cycles. For example, if thread T_0 has a scheduling frequency of $1/2$ and T is a different thread, then a resulting schedule could be: $T_0\ T\ T_0\ T\ T_0\ T\ \ldots$.

A constant scheduling frequency is useful for WCET analysis because the thread cycle cost of each instruction is constant and known for each scheduling frequency. A branch-taken or jump instruction, for example, requires three, two, or one thread cycles for scheduling frequencies 1, $1/2$, or $1/3+$, respectively.

Constant scheduling frequency is required for isolation and timing predictability of each hard real-time thread (HRTT), but not required for soft real-time threads (SRTTs) so that they can use all available cycles. To implement this, both HRTTs and SRTTs can either be active or sleeping. If active, the thread is allowed to be scheduled. If sleeping, the thread is not allowed to be scheduled and thus will not consume any processor cycles. A thread is put in this mode using the `TS` (thread sleep) instruction. A thread can be activated by an interrupt mechanism, such as a timing instruction or external I/O—allowing rapid, event-driven responses without wasting cycles polling. An active HRTT is only scheduled at prescribed cycles to maintain a constant scheduling frequency. In addition to being potentially scheduled at prescribed cycles, an active SRTT will share available cycles between all other active SRTTs in a round-robin fashion. Available cycles themselves can be prescribed for SRTTs or occur when an SRTT or HRTT is sleeping.

Example 4: Consider a system with active HRTT T_0, sleeping HRTT T_1, active SRTTs T_2 and T_3, and sleeping SRTT T_4. If T_0 has a scheduling frequency of $1/4$, and T_1 has a scheduling frequency of $1/2$, then a resulting schedule could be as follows:

$T_0\ T_2\ T_3\ T_2\ T_0\ T_3\ T_2\ T_3\ \ldots$

T_0 is the only active HRTT and is scheduled every forth cycle to satisfy its scheduling frequency. The rest of the cycles are shared round-robin between active SRTTs T_2 and T_3.

Example 5: Consider the same system as Example 4, but HRTT T_1 and SRTT T_4 have been activated. A resulting schedule could be as follows:

$T_0\ T_1\ T_2\ T_1\ T_0\ T_1\ T_3\ T_1\ T_0\ T_1\ T_4\ T_1\ \ldots$

Now only one out of every four cycles is used for SRTTs. An SRTT can be scheduled like an HRTT to guarantee a minimum scheduling frequency, but unlike an HRTT, would also share available cycles with other SRTTs.

In addition to being able to provide scheduling frequency guarantees to threads and utilizing unused cycles, this scheduling technique is useful for applications with varying concurrency and deadlines. Although the thread scheduling could be statically set at boot and never changed, it could also be changed dynamically during runtime. For example, an HRTT's scheduling frequency could be increased to meet a deadline it would otherwise miss, or a platform could switch modes of operation.

The thread scheduler uses two *control registers* to implement this technique: the *slots control register* prescribes cycles to certain threads and the *thread mode control register* stores whether a thread is active or sleeping. A thread requires *supervisory mode* to modify the slots control register or the thread mode of a different thread. The slots control register provides 8 slots, where each slot can have one of the following values: D (the slot is disabled), S (the slot is used for SRTTs), or $T_0 - T_7$[2] (the slot is dedicated to that thread ID and only used for SRTTs if the thread is sleeping). To decide which thread to schedule next, non-disabled slots are cycled through in a round-robin fashion, either using the specified thread ID if active or delegating to SRTT round-robin. It is the responsibility of the programmer or compiler to then assign the slots control register with HRTT IDs such that each HRTT has a constant scheduling frequency.

While the slots control register guarantees cycles to certain threads, the thread mode register specifies the mode of each thread and is used to schedule SRTTs. A thread can be in one of four modes: active HRTT (HA), sleeping HRTT (HZ), active SRTT (SA), or sleeping SRTT (SZ). A disabled thread can just be set to an HRTT mode and not be specified in the slots control register. When a cycle is delegated to SRTTs (its value is S or the specified thread is sleeping), the next SRTT in a round-robin rotation of active SRTTs is selected.

The layout of the slots control register is shown in Table Ia, with a possible assignment thats implements the schedule in Examples 4 and 5 in parentheses; the actual schedules are only different because of different thread modes. If $S_3(T_0)$ and $S_2(T_1)$ were to be swapped, T_1 would no longer have a constant scheduling frequency because the spacing between instructions from that thread would no longer be constant ($T_1\ T_0\ S\ T_1\ T_1\ T_0\ S\ T_1\ \ldots$). The layout of the thread mode register is shown in Table Ib, with values corresponding to Example 4 in parentheses. Hardware can also modify the mode of a thread, allowing interrupts to wake a thread from sleep.

[2]This is the case for a configuration with 8 threads. A 32-bit register can support up to 14 unique thread IDs.

31–28	27–24	23–20	19–16	15–12	11–8	7–4	3–0
$S_7(D)$	$S_6(D)$	$S_5(D)$	$S_4(D)$	$S_3(T_0)$	$S_2(T_1)$	$S_1(S)$	$S_0(T_1)$

(a) The slots control register

15–14	13–12	11–10	9–8	7–6	5–4	3–2	1–0
$T_7(D)$	$T_6(D)$	$T_5(D)$	$T_4(SZ)$	$T_3(SA)$	$T_2(SA)$	$T_1(HZ)$	$T_0(HA)$

(b) The thread mode control register

TABLE I: Thread scheduling is set by two control registers.

D. Timing Instructions

New *timing instructions* augment the RISC-V ISA for expressing real-time semantics. In contrast to previous PRET architectures supporting timing instructions [14], [18], [21], our design is targeted for mixed-critical systems.

The FlexPRET processor contains an internal clock that counts the number of elapsed nanoseconds since the processor was booted. The current time is stored in a 64-bit register, meaning that the processor can be active for 584 years without the clock counter wrapping around. Two new instructions can be used to get the current time: *get time high* `GTH r1` and *get time low* `GTL r2` store the higher and lower 32 bits in register `r1` and `r2`, respectively. When `GTL` is executed, the processor stores internally the higher 32 bits of the clock and then returns this stored value when executing `GTH`. As a consequence, executing `GTL` followed by `GTH` is atomic, as long as the instruction order is preserved.

To provide a *lower bound* on the execution time for a code fragment, the RISC-V ISA is extended with a *delay until* instruction `DU r1,r2`, where `r1` is the higher 32 bits and `r2` is the lower 32 bits of an absolute time value. Semantically, the thread is delayed (replays this instruction) until the current time becomes larger or equal to the time value specified by `r1` and `r2`. However, in contrast to previous processors supporting timing instructions (e.g., PTARM [14], [18]), the clock cycles are not wasted, but can instead be utilized for other SRTTs.

To provide an *upper bound* on execution time without constantly polling, a task needs to be interrupted. Instruction *exception on expire* `EE r1,r2` enables a timer exception that is executed when the current time exceeds `r1,r2`. The jump address is specified by setting a control register with `MTPCR` (move to program control register). Only one exception per thread can be active at any point in time; nested exceptions must be implemented in software. The instruction *deactivate exception on expire* `DE` deactivates the timer exception.

Exception on expire can be used for many purposes, such as detecting and handling a deadline miss, implementing a preemptive scheduler, or performing timed I/O. By first issuing an exception on expire and then executing a new *thread sleep* `TS` instruction, the clock cycles for the sleeping thread can be utilized by other active SRTTs. Another use of exception on expire is for *anytime algorithms*, that is, algorithms that can be interrupted at any point in time and returns a better solution the longer time it is executed.

E. Memory Hierarchy

For spatial isolation between threads, FlexPRET allows threads to read anywhere in memory, but only write to certain regions. The regions are specified by control registers that can only be set by a thread in supervisory mode with `MTPCR`. Virtual memory is a standard and suitable approach, but FlexPRET currently uses a different scheme for simplicity. There is one control register for the upper address of a shared region (which starts at the bottom of data memory) and two control registers per thread for the lower and upper addresses of a thread-specific region. Memory is divided into 1kB regions, and a write only succeeds if the address is within the shared or thread-specific region. By specifying all thread-specific regions and the shared region to be disjoint, each thread will have both private memory and access to shared memory.

For timing predictability, FlexPRET uses *scratchpad memories* [22]. These are local memories that have a separate address space than main memory and are explicitly controlled by software; all valid memory accesses always succeed and are single cycle, unlike caches where execution time depends on cache state. There is active research in scratchpad memory management techniques to reduce WCET [23]. Instructions are stored in *instruction scratchpad memory (I-SPM)* and data is stored separately in *data scratchpad memory (D-SPM)*. Scratchpad memories are not required; caches could be used instead if the reduction in fine-grained predictability is acceptable. We envision a hybrid approach where HRTTs tasks use scratchpads and SRTTs use caches for future versions of FlexPRET.

F. Programming, Compilation, and Timing Analysis

FlexPRET can be programmed using low level programming languages, such as C, that are augmented with constructs for expressing temporal semantics. FlexPRET can be an integral part of a precision timed infrastructure [24] that includes languages and compilers with an ubiquitous notion of time. Such a complete infrastructure with timing-aware compilers is outside the scope of this paper; instead, we use a RISC-V port of the `gcc` compiler and implement the new timing instructions using inline assembly. The following code fragment illustrates how a simple periodic control loop can be implemented.

```
1 int h,l;              // High and low 32-bit values
2 get_time(h,l);        // Current time in nanoseconds
3 while(1){             // Repeat control loop forever
4   add_ms(h,l,10);     // Add 10 milliseconds
5   exception_on_expire(h,l,missed_deadline_handler);
6   compute_task();     // Sense, compute, and actuate
7   deactivate_exception(); // Deadline met
8   delay_until(h,l); // Delay until next period
9 }
```

Before the control loop is executed, the current time (in nanoseconds) is stored in variables `h` and `l` (line 2). The time is incremented by 10ms (line 4) and a timer exception is enabled (line 5), followed by task execution (line 6). If a deadline is missed, an exception handler `missed_deadline_handler` is called. To force a lower bound on the timing loop, the execution is delayed until the time period has elapsed (line 8); the cycles during the delay can be used by an active SRTT. Functions `get_time`, `exception_on_expire`, `deactivate_exception`, and `delay_until` implement the new RISC-V timing instructions using inline assembly.

To have full control over timing, real-time applications can be implemented as bare-metal software, using only lightweight libraries for hardware interaction. As a scheduling design methodology, we propose that tasks with the highest criticality level (e.g. A in DO-178C [4]) are assigned individual HRTTs, thus providing both temporal and spatial isolation. The next-highest criticality level tasks (e.g. B in DO-178C) also use HRTTs, but several tasks can share the same thread, thus reducing the hardware enforced isolation. Lower criticality tasks (e.g. C, D, E in DO-178C) can then share SRTTs

using standard scheduling algorithms, such as rate-monotonic scheduling and earliest deadline first (EDF). In the evaluation section (Section IV-D), we apply this scheduling methodology to a mixed-criticality avionics example.

For high-criticality tasks it is typically required that the upper bound is guaranteed statically at compile time. Well established worst-case execution (WCET) analysis techniques can be applied to compute safe upper bounds on tasks. In particular, HRTTs possess fine-grained timing predictability (at the instruction level), making hardware timing analysis [25] especially simple. Although no timing analysis tools currently exist for the RISC-V ISA, we contend that standard WCET computation techniques [16] and state-of-the-art industrial WCET tools, such as AbsInt[3], can easily be adapted to compute WCET estimates for HRTTs for FlexPRET. Timing analysis for SRTT is, however, inherently harder, because of cycle stealing. Instead, we propose to use measurement-based approaches for low-criticality tasks. Fortunately, measurement of time is particular simple in the proposed architecture; the ISA timing instructions can be used to give precise measurements with minimal overhead.

IV. Evaluation

We implemented and deployed FlexPRET as a soft-core on an FPGA to demonstrate feasibility and provide quantitative resource costs of the proposed architectural techniques. We also simulated two mixed-criticality task sets running on FlexPRET. The first is simple and shows how FlexPRET provides isolation to HRTTs and efficiently executes SRTTs, even when an error occurs on an SRTT. The second is more complex (21 tasks on 8 threads) and requires certain threads to use software-based scheduling to execute multiple tasks. We explain the methodology used and discuss the implications.

A. Implementation

We implemented FlexPRET in Chisel [26], a hardware construction language that generates both Verilog code and a cycle-accurate C++-based simulator. Chisel allows us to easily parameterize the code to produce various configurations for both FlexPRET and two baseline processors for comparison. We use two similar baseline processors instead of comparing to existing processors to remove the influence of differing ISAs and optimization techniques. The intent is to show and discuss the incremental costs of flexible thread scheduling and timing instructions, as these techniques are not restricted in use to a 5-stage RISC-V processor. The FlexPRET processor may be referred to as *FlexPRET-4T*, where the number represents the physical number of threads available to be used.

The first baseline processor will be referred to as *Base-1T* and is a single-threaded 5-stage RISC-V processor with scratchpad memories, predict not-taken branching, and forwarding, stalling, or flushing to resolve data and control hazards. This processor functions identically to FlexPRET when FlexPRET's scheduler executes the same thread every cycle. The second baseline processor will be referred to as *Base-4T-RR* and is a fine-grained multithreaded 5-stage RISC-V processor with scratchpad memories, much like PTARM [14] and XMOS [15]. It interleaves a fixed four threads in the pipeline, and consequently does not require any forwarding, stalling, or flushing. This processor functions identically to FlexPRET when FlexPRET's scheduler executes four threads as HRTTs in a round-robin pattern.

[3]http://www.absint.com/

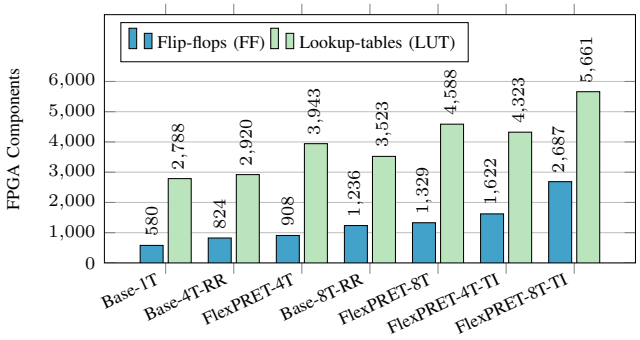


Fig. 2: FPGA resource usage for different processors

In all processors, a subset of the RISC-V ISA is implemented, currently excluding floating point arithmetic instructions, atomic memory operations, integer division, and support for packed instructions. These instructions are not needed to demonstrate the main ideas. To verify correct implementation of both FlexPRET and the two baseline processors, an assembly test suite and more than ten C benchmarks from the Mälardalen [27] benchmark suite were run for all configurations on both the cycle-accurate simulator and an FPGA implementation of the Verilog code, with results being examined for correctness.

B. FPGA Resources

Several configurations of FlexPRET and the baseline processors were deployed on an FPGA (Xilinx Virtex-5 XC5VLX110T) to evaluate the area cost associated with different features. The processors were all clocked at 80MHz with a 16kB I-SPM and a 16kB D-SPM in block RAM. A block RAM has a fixed size that is large enough for eight register files, so only a single dedicated block RAM is used for all register files in each configuration.

The flip-flop (FF) and lookup-table (LUT) usages are shown in Figure 2, where *TI* implies the processor implements timing instructions as described in Section III-D. Although the percentage increase may appear large in some cases, these numbers are from bare-minimum implementations and the absolute cost is relatively low. As a processor's area increases with more complex functionality, such as supporting integer division, a floating-point unit, different memory hierarchy, or more peripherals, the relative percentage cost drops.

The resource difference between *Base-1T* and *Base-4T-RR* shows the cost of fine-grained multithreading, with a 5% increase in LUTs and a 42% increase in FFs. Although *Base-4T-RR* removes forwarding paths and control logic for stalling and flushing, it requires more multiplexing based on thread IDs. There is also state that must be stored for each thread, such as program counter and some control registers.

The resource difference between *Base-4T-RR* and *FlexPRET-4T* shows the cost of adding flexible thread

scheduling. The 35% increase in LUTs and 10% increase in FFs is caused by the thread scheduler, forwarding paths, and additional control logic for stalling and flushing. The eight thread versions, *Base-8T-RR* and *FlexPRET-8T*, are useful for applications where more isolated threads are desired, but do so at the cost of additional area.

It is important to notice that a system of *Base-1T* cores could provide higher throughput per area than a system of FlexPRET cores for certain task sets, particularly soft real-time task sets. However, for a mixed-criticality task set where hardware-based isolation guarantees are the controlling constraint, fewer FlexPRET processors could provide the same functionality of many more traditional processors—a substantial area and power savings. For example, if four tasks that require hardware-based isolation can execute on a single *FlexPRET-4T*, then four *Base-1T* processors are not needed.

At first glance, adding the timing instructions described in Section III-D to *FlexPRET-4T* looks costly, with a 79% increase of FFs and a 10% increase of LUTs. The main source is supporting delay until and exception on expire instructions, where two 64-bit expiration times need to be stored for each thread and compared to the current time every cycle. This additional cost could be reduced by roughly half if absolute time is reduced from 64-bits to 32-bits, but rollover becomes an issue if any time interval is on the order of seconds. In practice, FlexPRET could use less bits for time and use software to handle larger time intervals to reduce area, but we showed the 64-bit version because it is worst-case for area. On most microcontrollers, timers are implemented outside the processor and similar area would be required to achieve the same precision and flexibility.

C. Demonstration of Hardware-Based Isolation and Efficient Resource Utilization

A simple mixed-criticality example with four periodic tasks, τ_A, τ_B, τ_C and τ_D, was simulated on FlexPRET as a more concrete demonstration of the differences between HRTTs and SRTTs. Although the FPGA implementation is useful for evaluating correctness, feasibility, and area costs, our cycle-accurate simulator is useful for varying configurations and monitoring the timing behavior of each task. This example simulated tasks running on a 100MHz *FlexPRET-4T* with a 32kB I-SPM and 32kB D-SPM, a configuration that would not be unreasonable for a soft-core or ASIC implementation.

The task identifiers A, B, C, D also correspond to criticality levels from highest to lowest, with A and B as hard real-time tasks and C and D soft real-time tasks. Each task executes on its own thread, so each thread may be referred to by the task it executes. Each task's thread ID, initial thread mode, period (T_i), deadline (D_i), and worst-case execution cycles for each scheduling frequency ($E_{i,1}, E_{i,1/2}, E_{i,1/3+}$) are shown in Table IIa. The WCET C_i depends on the worst-case execution cycles $E_{i,x}$ and the frequency of the thread $f*x$ ($f = 100MHz$ is the frequency of the processor), where $C = E_x/(f * x)$. For example, if τ_A executes every processor cycle ($x = 1$) it would take $4 * 10^5/(100 * 10^6 Hz) = 4ms$ to complete, and if it executed τ_A every other processor cycle ($x = \frac{1}{2}$), it would take $3.65 * 10^5/(100 * 10^6 Hz * \frac{1}{2}) = 7.3ms$ to complete.

One of the properties of FlexPRET is the number of cycles required to complete a task can depend on the scheduling frequency, as more cycles can be wasted at higher scheduling frequencies to prevent hazards. To account for this variation in execution cycles, each task iterates a program from the Mälardalen benchmark suite a fixed number of times to reach the value of $E_{i,1}$ specified in Table IIa, and the values of $E_{i,1/2}$ and $E_{i,1/3+}$ are then measured. The values of $E_{i,1}$ are contrived to highlight properties. Tasks τ_A and τ_B use *statemate* (generated automotive controller), τ_C uses *jfdctint* (discrete-cosine transformation), and τ_D uses *insertsort* (sorting algorithm). Inputs are always the same, so each *job* (iteration of the task) takes the same number of cycles. Periodic release of each task is simple: after the task completes a *delay until* instruction prevents the thread from being scheduled until the next period starts.

The schedule set by the slots control register is shown in Table IIb. Task τ_A executes every 3rd processor cycle, τ_B every 6th processor cycle, τ_C every 6th processor cycle. Recall that if a task does not need to use its cycle or the slot is marked as S, that cycle is used by active SRTTs (potentially τ_C and τ_D in this case) in a round-robin fashion. Even though τ_C is a soft real-time task, it is given a scheduling slot in order to execute more frequently than τ_D because it requires higher throughput.

Figure 3 shows execution traces for a single hyperperiod ($t = 12$) for different situations on *FlexPRET-4T*. The horizontal axis is time and the vertical direction is one iteration through the slots control register, with the value of each slot in parentheses. Figure 3a shows normal operation, where all tasks meet their deadlines. The first job of both τ_A and τ_B finishes before the deadline and the remaining allocated cycles are shared by active SRTTs, which means τ_C and τ_D until τ_D completes. Note that this task set would not even be schedulable on *Base-4T-RR* because τ_A and τ_C each require more than $1/4$ of the processor cycles to meet their deadlines, which cannot be provided by four thread round-robin.

Figure 3b shows an error case where τ_D completes immediately, and Figure 3c and shows an error case where τ_D executes infinitely. Their cause is not important, these two extremes are just to demonstrate isolation of HRTTs. Regardless of the operation of τ_D, the timing behavior of τ_A and τ_B, the most critical tasks, is identical. When τ_D finishes immediately, τ_C executes more frequently and finishes sooner than in normal operation. When τ_D is in an infinite loop, τ_C takes slightly longer but still meets its deadline.

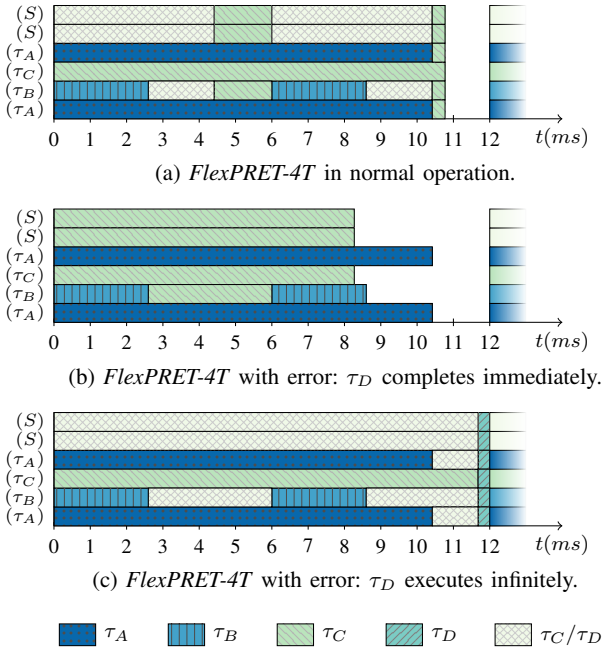
Task	Thread ID	Thread Mode	T_i, D_i (ms)	$E_{i,1}$ ($*10^5$)	$E_{i,1/2}$ ($*10^5$)	$E_{i,1/3+}$ ($*10^5$)
τ_A	0	HA	12	4.00	3.65	3.45
τ_B	1	HA	6	0.50	0.45	0.43
τ_C	2	SA	12	4.80	4.69	4.59
τ_D	3	SA	6	1.00	0.93	0.86

(a) The task set

D	D	S	S	$0(\tau_A)$	$2(\tau_C)$	$1(\tau_B)$	$0(\tau_A)$

(b) The slots control register

TABLE II: A simple mixed-criticality example



(a) *FlexPRET-4T* in normal operation.

(b) *FlexPRET-4T* with error: τ_D completes immediately.

(c) *FlexPRET-4T* with error: τ_D executes infinitely.

Fig. 3: *FlexPRET-4T* demonstrating hardware-based isolation and resource utilization efficiency.

D. Case Study: Avionics Mixed-Criticality System

To demonstrate a possible methodology for mapping and scheduling a mixed criticality task set on FlexPRET, we simulated a task set from Vestal's influential paper on mixed-criticality scheduling [1]. The abstract workload contains 21 tasks and is derived from a time-partitioned avionics systems at Honeywell. The allocated execution time numbers are used to have a combined single-threaded processor utilization of 93%. We simulate execution time using the same method as in the previous section; A and B criticality level tasks iterate the *statemate* program, and C and D use either *jfdctint* or *insertsort*. This example simulates tasks running on a 100MHz *FlexPRET-8T* with a 128kB I-SPM and 128kB D-SPM.

Our approach is to isolate and over-allocate resources to the hard real-time tasks (criticality levels A and B) and use slack stealing to efficiently utilize the processor for soft real-time tasks (criticality levels C and D), similar to reservation-based scheduling used in commercial RTOSs for mixed-criticality systems [28], but with hardware-based isolation guarantees. Each task's thread ID, period (T_i), deadline (D_i), and worst-case execution cycles for each scheduling frequency ($E_{i,1}, E_{i,1/2}, E_{i,1/3+}$) are shown in Table IIIa, and the slots control register is shown in Table IIIb.

Tasks $\tau_{A1} - \tau_{A3}$ each execute on their own HRTT, isolating each task from all other tasks. Once a task completes, the *delay until* instruction is used to wait until the next period release. Each are given a scheduling frequency of $1/8$ because this is sufficient for meeting respective deadlines; over-allocation is acceptable because *delay until* will donate cycles to other threads. Complete hardware-based isolation and lack of preemption simplifies WCET analysis and provides the highest level of confidence.

Due to resource constraints, tasks $\tau_{B1} - \tau_{B7}$ cannot each execute on their own HRTT, but they can still be isolated from the A, C, and D criticality level tasks. Tasks $\tau_{B1} - \tau_{B3}$ are mapped to one HRTT and are able to use use a non-preemptive static schedule to meet their respective deadlines at scheduling frequency $1/4$; tasks $\tau_{B4} - \tau_{B7}$ are mapped to another HRTT and use preemptive rate-monotonic software scheduling to meet their respective deadlines at schedule frequency $1/4$. Different scheduling algorithms could be used, but we selected the simplest ones that provided schedulability for these task sets to provide the highest level of confidence. Both HRTT will donate cycles when not needed: the static schedule uses *delay until* until next periodic release, and the rate-monotonic scheduler uses *thread sleep* until an *exception on expire* occurs to release tasks.

Even though $\tau_{C1}, \tau_{D1} - \tau_{D9}$ could be mapped to one SRTT, throughput is typically higher if two threads are interleaved, so the tasks are split between two SRTTs. Each SRTT uses a preemptive EDF scheduler for simplicity, although other scheduling algorithms could also be used. *Exception on expire* is used to add new jobs to a priority queue sorted by deadline, and the scheduler is run whenever a job finishes or jobs are added, possibly preempting an executing job. Despite not being allocated a slot in the schedule, slack stealing from the over-allocation of cycles to the HRTTs of the A and B criticality level tasks is enough to meet all deadlines in this example.

Figure 4 shows execution traces for a single hyperperiod ($t = 200$). Each subplot is for a different thread and the rectangles show when the jobs of each task are executing. Whenever the job is preempted, the rectangle is a lighter shade with a dotted line. Up arrows are release times and down arrows are deadlines for each task. Each job of a task always takes the same number of cycles. For tasks $\tau_{A1} - \tau_{A4}$ and $\tau_{B1} - \tau_{B7}$, the threads are isolated so each job takes the same amount of time. Notice that the execution times of tasks τ_{C1} and $\tau_{D1} - \tau_{D9}$ vary, this is because the number of cycles donated by $\tau_{A1} - \tau_{A4}$ and $\tau_{B1} - \tau_{B7}$ varies as well.

V. Related Work

This paper is most closely related to two areas of research: timing predictable processors and software-based scheduling for mixed-criticality systems.

A. Timing Predictable Processors

Berg et al. [29] and Heckmann et al. [30] identified the architectural properties that complicate WCET analysis and proposed design principles that facilitate it. Edwards and Lee [31] went further to argue for precision time (PRET) machines that incorporate time into the abstraction levels, making temporal behavior as important as logical functionality. Researchers have proposed many timing predictable processor for real-time systems; each processor making different trade-offs to target an application space. The SPEAR processor by Delvai et al. [32] is a 16-bit, 3-stage processor that has constant-time instructions by removing caches and assuming single-path programming. Schoeberl's Java Optimized Processor (JOP) [33] predictably executes Java bytecode by translating it to microcode for a simple 3-stage pipeline. To execute a program with synchronous semantics, Andalam's ARPRET [34] achieves predictability by customizing an existing soft-core processor.

Task	Thread ID	Thread Mode	T_i, D_i (ms)	$E_{i,1}$ $(*10^5)$	$E_{i,1/2}$ $(*10^5)$	$E_{i,1/3+}$ $(*10^5)$
τ_{A1}	0	HA	25	1.10	1.00	0.95
τ_{A2}	1	HA	50	1.80	1.64	1.55
τ_{A3}	2	HA	100	2.00	1.82	1.72
τ_{A4}	3	HA	200	5.30	4.83	4.56
τ_{B1}	4	HA	25	1.40	1.27	1.20
τ_{B2}	4	HA	50	3.90	3.54	3.34
τ_{B3}	4	HA	50	2.80	2.54	2.40
τ_{B4}	5	HA	50	1.40	1.28	1.21
τ_{B5}	5	HA	50	3.70	3.37	3.19
τ_{B6}	5	HA	100	1.80	1.64	1.55
τ_{B7}	5	HA	200	8.50	7.75	7.32
τ_{C1}	6	SA	50	1.90	1.77	1.63
τ_{D1}	6	SA	50	5.40	5.03	4.65
τ_{D2}	6	SA	200	2.40	2.33	2.28
τ_{D3}	6	SA	50	1.30	1.26	1.23
τ_{D4}	6	SA	200	1.50	1.45	1.42
τ_{D5}	7	SA	25	2.30	2.14	1.98
τ_{D6}	7	SA	100	4.80	4.65	4.30
τ_{D7}	7	SA	200	13.00	12.70	12.44
τ_{D8}	7	SA	100	0.60	0.57	0.56
τ_{D9}	7	SA	50	2.40	2.33	2.28

(a) The task set

5	3	4	2	5	1	4	0

(b) The slots control register

TABLE III: A mixed-criticality avionics case study

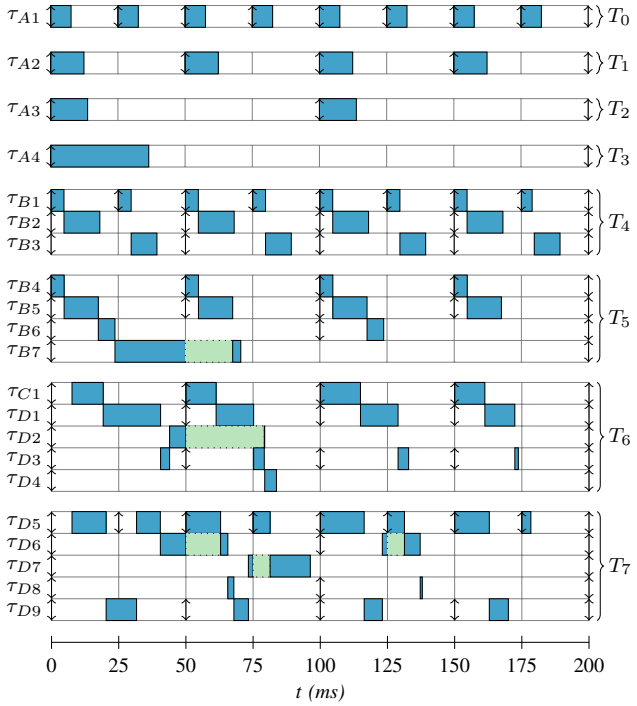


Fig. 4: *FlexPRET-8T* executing a mixed-criticality avionics case study.

PTARM [14] by Liu et al. and XMOS X1 [15] are architecturally similar to FlexPRET. Both are fine-grained multithreaded 5-stage RISC processors that require at least four threads (exactly four threads for PTARM) to be round-robin interleaved in the pipeline; cycles are wasted if there are fewer than four active threads, and a single thread can only be executed at most once every four cycles. PTARM is better suited for hard real-time tasks because all threads have a constant scheduling frequency. Conversely, XMOS is better suited for soft real-time tasks because inactive tasks can be left out of round-robin scheduling, but scheduling frequency depends on the maximum number of simultaneously active threads. The Merasa project [35] is the most closely related work on hardware for mixed-criticality systems and uses similar approaches, but is focused more at the multicore level. Like FlexPRET, it provides isolation and timing predictability to hard real-time threads by using predictable thread scheduling and scratchpad memories instead of caches, but is limited to one hard real-time thread per core.

B. Software-based Scheduling

Software based scheduling for mixed-criticality software is typically either reservation-based or priority-based [2]. Reservation-based is best demonstrated by the ARINC 653 standard used in integrated modular avionic (IMA) systems [36]. Critical tasks are guaranteed segments of time, and most RTOSs will steal cycles for other tasks if a task finishes early, as done by Wind River's VxWorks 653 RTOS [28].

Using priority-based preemptive scheduling for mixed-criticality systems was first proposed by Vestal [1]. Since then, there has been much work addressing scheduling theory of mixed-criticality systems, as recently summarized by Burns [10]. Scheduling sporadic tasks was first addressed by Baruah and Vestal [7], and Niz et al. [8] presented a scheduling algorithm that protects high-criticality tasks from low-criticality tasks, even if a nominal WCET is overrun. More recently, Mollison et al. [9] proposed an approach for multicore platforms. Although FlexPRET does not implement priority-based scheduling in hardware, it can still be used as a platform for these algorithms: either scheduling tasks within a single thread or changing the thread scheduling.

VI. Conclusions and Future Work

Hardware-based isolation requires executing each task on a separate computational component, which could be a processor, core, or hardware thread, and typically results in underutilization of hardware resources. FlexPRET uses fine-grained multithreading and flexible thread scheduling to provide hardware-based isolation and predictability to HRTTs, but also allows SRTTs to use any cycle not needed by an HRTT. If there are more tasks than hardware threads available, either software-based scheduling can be used on some threads or additional FlexPRET cores can be added to the system.

We consider FlexPRET a key contribution to a precision timed infrastructure [24], where languages, compilers, and architectures allow the specification and preservation of timing semantics. The next steps in that direction are tool support for formal verification of hard real-time tasks and investigating how languages can leverage FlexPRET's properties. From the hardware perspective, the presented architectural techniques

could be applied to other processors, or FlexPRET could be a core in a multicore system that uses a predictable network-on-a-chip for communication.

This type of hardware platform presents an interesting new scheduling problem. Not only do tasks need to be mapped and scheduled on threads, but the logical execution frequency of each thread can be controlled by modifying the slots control register *during run-time*. ILP-based multiprocessor scheduling methodologies [37] can be extended with constraints for modeling these possible configurations, but such an approach may not scale. How to optimally exploit this flexibility to control resource distribution amongst SRTTs while still maintaining hardware-based isolation for HRTTs is an open scheduling problem.

REFERENCES

[1] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *Proc. of the 28th IEEE International Real-Time Systems Symposium (RTSS)*, 2007, pp. 239–243.

[2] S. Baruah, H. Li, and L. Stougie, "Towards the design of certifiable mixed-criticality systems," in *Proc. of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2010, pp. 13–22.

[3] *653P1-3 Avionics Application Software Standard Interface, Part 1, Required Services*, Aeronautical Radio, Inc. Std., 2010.

[4] *DO178C: Software Considerations in Airborne Systems and Equipment Certification*, RTCA Std., 2012.

[5] J. Lala and R. Harper, "Architectural principles for safety-critical real-time applications," *Proc. of the IEEE*, vol. 82, no. 1, pp. 25–40, 1994.

[6] C. Watkins and R. Walter, "Transitioning from federated avionics architectures to integrated modular avionics," in *Proc. of the 26th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, 2007, pp. 2.A.1–1–2.A.1–10.

[7] S. Baruah and S. Vestal, "Schedulability analysis of sporadic tasks with multiple criticality specifications," in *Proc. of the Euromicro Conference on Real-Time Systems (ECRTS)*, 2008, pp. 147–155.

[8] D. de Niz, K. Lakshmanan, and R. Rajkumar, "On the scheduling of mixed-criticality real-time task sets," in *Proc. of the 30th IEEE Real-Time Systems Symposium (RTSS)*, 2009, pp. 291–300.

[9] M. Mollison, J. Erickson, J. Anderson, S. Baruah, and J. Scoredos, "Mixed-criticality real-time scheduling for multicore systems," in *Proc. of the 10th IEEE International Conference on Computer and Information Technology (CIT)*, 2010, pp. 1864–1871.

[10] A. Burns and R. I. Davis, "Mixed criticality systems: A review," Department of Computer Science, University of York, Tech. Rep. MCC-1(b), 2013.

[11] J. L. Herman, C. J. Kenna, M. S. Mollison, J. H. Anderson, and D. M. Johnson, "RTOS support for multicore mixed-criticality systems," in *Proc. of the 18th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2012, pp. 197–208.

[12] J. E. Thornton, *Design of a Computer: The Control Data 6600*. Scott Foresman & Co, 1970.

[13] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: a 32-way multithreaded Sparc processor," *IEEE Micro*, vol. 25, no. 2, pp. 21–29, 2005.

[14] I. Liu, J. Reineke, D. Broman, M. Zimmer, and E. Lee, "A PRET microarchitecture implementation with repeatable timing and competitive performance," in *Proc. of the 30th IEEE International Conference on Computer Design (ICCD)*, 2012, pp. 87–93.

[15] D. May, *The XMOS XS1 Architecture*. XMOS, 2009.

[16] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 36:1–36:53, 2008.

[17] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee, "Predictable programming on a precision timed architecture," in *Proc. of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2008, pp. 137–146.

[18] I. Liu, "Precision timed machines," Ph.D. dissertation, University of California at Berkeley, 2012.

[19] N. J. H. Ip and S. A. Edwards, "A processor extension for cycle-accurate real-time software," in *Proc. of the 2006 International Conference on Embedded and Ubiquitous Computing*, vol. 4096, 2006, pp. 449–458.

[20] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The RISC-V instruction set manual, volume I: Base user-level ISA," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2011-62, 2011.

[21] D. Bui, E. Lee, I. Liu, H. Patel, and J. Reineke, "Temporal isolation on multiprocessing architectures," in *Proc of the 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2011, pp. 274–279.

[22] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: design alternative for cache on-chip memory in embedded systems," in *Proc. of the 10th International Symposium on Hardware/Software Codesign (CODES)*, 2002, pp. 73–78.

[23] Y. Kim, D. Broman, J. Cai, and A. Shrivastaval, "WCET-aware dynamic code management on scratchpads for software-managed multicores," in *Proc. of the 20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

[24] D. Broman, M. Zimmer, Y. Kim, H. Kim, J. Cai, A. Shrivastava, S. A. Edwards, and E. A. Lee, "Precision timed infrastructure: Design challenges," in *Proc. of the Electronic System Level Synthesis Conference (ESLsyn)*, 2013.

[25] C. Ferdinand and R. Wilhelm, "Efficient and precise cache behavior prediction for real-time systems," *Real-Time Systems*, vol. 17, no. 2, pp. 131–181, 1999.

[26] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a Scala embedded language," in *Proc. of the 49th ACM/EDAC/IEEE Design Automation Conference*, 2012, pp. 1216–1225.

[27] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The mälardalen WCET benchmarks: Past, present and future," in *10th International Workshop on Worst-Case Execution Time Analysis (WCET)*, vol. 15, 2010, pp. 136–146.

[28] Wind River VxWorks 653 Platform. [Online]. Available: http://www.windriver.com/products/platforms/safety_critical_arinc_653/

[29] C. Berg, J. Engblom, and R. Wilhelm, "Requirements for and design of a processor with predictable timing," in *Perspectives Workshop: Design of Systems with Predictable Behaviour*, 2004.

[30] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm, "The influence of processor architecture on the design and the results of WCET tools," *Proc. of the IEEE*, vol. 91, no. 7, pp. 1038–1054, 2003.

[31] S. Edwards and E. Lee, "The case for the precision timed (PRET) machine," in *Proc. of the 44th ACM/IEEE Design Automation Conference (DAC)*, 2007, pp. 264–265.

[32] M. Delvai, W. Huber, P. Puschner, and A. Steininger, "Processor support for temporal predictability - the SPEAR design example," in *Proc. of the 15th Euromicro Conference on Real-Time Systems (ECRTS)*, 2003, pp. 169–176.

[33] M. Schoeberl, *JOP: A java optimized processor for embedded real-time systems*. VDM Publishing, 2008.

[34] S. Andalam, "Predictable platforms for safety-critical embedded systems," Thesis, The University of Auckland, 2013.

[35] T. Ungerer *et al.*, "Merasa: Multicore execution of hard real-time applications supporting analyzability," *IEEE, Micro*, vol. 30, no. 5, pp. 66–75, 2010.

[36] P. Prisaznuk, "ARINC 653 role in integrated modular avionics (IMA)," in *Proc. of the 27th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, 2008, pp. 1.E.5–1–1.E.5–10.

[37] E. Yip, M. Kuo, D. Broman, and P. S. Roop, "Relaxing the synchronous approach for mixed-criticality systems," in *Proc. of the 20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.