Title
Predictable Processors for Mixed-Criticality Systems and Precision-Timed I/O

Author
Zimmer, Michael Patrick

Publication Date
2015

Peer reviewed|Thesis/dissertation

Predictable Processors for Mixed-Criticality Systems and Precision-Timed I/O

by

Michael Patrick Zimmer

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Engineering – Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Edward A. Lee, Chair
Professor Sanjit A. Seshia
Professor J. Karl Hedrick

Summer 2015

Predictable Processors for Mixed-Criticality Systems and Precision-Timed I/O

Abstract

Predictable Processors for Mixed-Criticality Systems and Precision-Timed I/O

by

Michael Patrick Zimmer

Doctor of Philosophy in Engineering – Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Edward A. Lee, Chair

In cyber-physical systems, where embedded computation interacts with physical processes, correctness depends on timing behavior, not just functionality. For high confidence in the timing behavior of real-time embedded software, the underlying hardware must support predictable and isolated task execution. These properties are sacrificed in many conventional processors that use frequent interrupts and hardware prediction mechanisms to improve average-case performance. Mixed-criticality systems—where tasks with different levels of safety criticality are integrated on a single hardware platform to share resources and reduce costs—facilitate complex functionality but further complicate design and verification. The challenge is designing processor architectures that provide high confidence in software functionality and timing behavior without sacrificing processor throughput.

This dissertation presents architectural techniques for task-level trade-offs between predictability, hardware-based isolation, and overall instruction throughput—facilitating the verification of safety-critical tasks and allowing software to meet precise timing constraints. Our processor design, named FlexPRET, uses fine-grained multithreading with flexible hardware thread scheduling and integrated timers to evaluate these techniques. With no restrictions on thread interleaving, the hardware thread scheduler executes hard real-time threads (HRTTs) at specific cycles for isolated and predictable behavior and allows soft real-time threads (SRTTs) to use both specific and spare cycles for efficient operation.

A configurable version of FlexPRET is implemented in simulation and on FPGA using Chisel, a hardware construction language that generates both C++ and Verilog code. For a given program path at a constant thread scheduling frequency, the latency of every instruction is constant and known, and the precision of input/output (I/O) instruction timing is bounded. The comparison of FlexPRET with two baseline processors provides the FPGA resource costs of FlexPRET's microarchitectural features. Using two example applications, we demonstrate a mixed-criticality deployment methodology that provides hardware-based isolation to critical tasks and improves overall instruction throughput by using spare cycles and software scheduling for less critical tasks. FlexPRET can also use software to perform multiple independent I/O operations instead of requiring hardware peripherals.

To my family.

Contents

List of Figures

List of Tables

Acknowledgments

First, I would like to thank Prof. Edward A. Lee for his constant support and guidance. He has been a great advisor, both accommodating my research interests and providing valuable insights that improved my research focus. Without his encouragement, I may not have pursued a Ph.D. I would also like to thank Prof. J. Karl Hedrick and Prof. Sanjit A. Seshia for providing valuable feedback while on my qualifying exam and dissertation committees.

I would like to thank all my collaborators at UC Berkeley for the helpful discussions and contributions. I have worked closely with David Broman over the past three years, and he been a great colleague, friend, and mentor, helping me navigate the subtleties of academia. His attention to detail and emphasis on strong research motivation and evaluation has made me a better researcher.

Isaac Liu and Jan Reineke sparked my initial interest in the PRET project and taught me the relevant background knowledge. Chris Shaver helped develop the first version of the FlexPRET processor, and Hokeun Kim was always available for debug assistance and hardware design discussions. During class projects, two groups in EECS 149/249A in Fall 2014 tested the FlexPRET processor and provided feedback for the software interface.

Before the PRET project, I was involved in the PTIDES project, which introduced me to cyber-physical system challenges and research in general. John Eidson and Slobodan Matic mentored me during my first project, which was time synchronization for PTIDES. I learned the details of PTIDES semantics from Jia Zou, and Patricia Derler taught me the inner workings of Ptolemy software.

I would like to thank my external collaborators for their feedback and insight from different perspectives, particularly Hugo Andrade and Trung Tran for the industry perspective on cyber-physical systems. Alain Girault and Partha Roop's group at Auckland University introduced me to synchronous programming ideas for real-time embedded systems and its compatibility with my work. The work of Aviral Shrivastava's group on scratchpad memory management has influenced and justified my memory hierarchy decisions.

I would like to thank everyone in the Ptolemy group and DOP Center for their general helpfulness and friendliness. There was always someone more knowledgeable in some area willing to stop their work and help. I really appreciate the work of Christopher, Barb, and Mary to keep everything running smoothly. I would also like to thank everyone in the RISC-V and Chisel teams at Berkeley for their debug assistance and development efforts, which were leveraged for the FlexPRET processor.

I would like to thank my amazing family and friends for their unwavering support. I was fortunate to have a fun and thoughtful group of graduate school friends. I am deeply grateful to my parents, Jean and Paul, for the values they taught me, the opportunities they have given me, and their love, advice, and encouragement. The kindness and constant enthusiasm of my sister, Julie, has always been heartening during stressful times. My twin brother, Brian, and his wife, Anjali, have been incredibly supportive and I am glad that we all ended up at Berkeley. Lastly, I am extremely thankful for the steadfast love, patience, and encouragement from my future wife, Sarah.

Chapter 1

Introduction

1.1 Motivation

In cyber-physical systems (CPS), computation interacts with physical processes [1]. Embedded computers, often connected through networks, both monitor and control physical processes using sensing and actuation. Application areas such as automotive, avionics, industrial automation, and medical devices contain these systems with diverse requirements. The variety of applications poses challenges for flexible design techniques but presents potential for high-impact solutions.

Unlike the more traditional field of real-time embedded systems [2], cyber-physical systems focus on and leverage the feedback interaction between computation and physical processes. In both types of systems, computation is not instantaneous, and its timing behavior affects the physical world. As a result, *software timing behavior is part of correctness* [3]. In contrast, correctness in general-purpose software only depends on input-output values, and software timing behavior is just a measure of performance. Many hardware and software techniques used in general-purpose computing—which abstract away timing information to simplify programming and improve average-case performance—cause unpredictable timing behavior and are not directly applicable to real-time systems [4]. However, system developers still use these techniques, because utilizing existing hardware, compilers, operating systems, programming languages, and experience reduces costs and effort, at least initially.

In general, engineers, regulatory agencies, and consumers demand *higher confidence in software timing and functionality* for cyber-physical than general-purpose systems. Many application areas, such as automotive and avionics, contain safety-critical functionality where failure has potentially catastrophic consequences. Regardless of whether bodily injuries occur, software bugs or low correctness confidence causes expensive damage, recalls, litigation, and diminished functionality. High-profile examples are court cases for unintended acceleration in vehicles involving Toyota [5] and problems caused by priority inversion on the Mars Pathfinder [6].

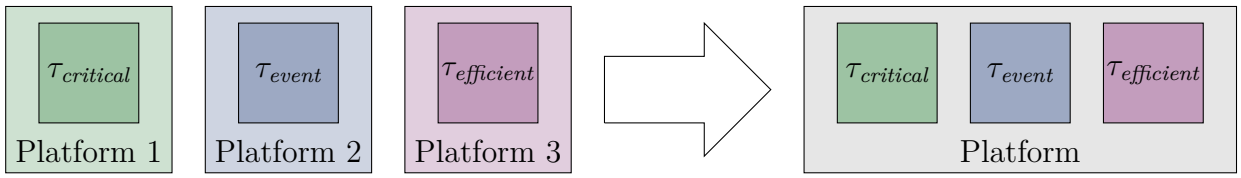


Figure 1.1: Transition from a federated to integrated architecture for task execution.

Achieving higher confidence in software timing and functionality is a difficult problem. Testing alone cannot provide enough guarantees because the state space is too large to exhaustively test for any application of reasonable complexity. Formal verification and analysis techniques provide further guarantees but operate on abstractions of the implementation [7], [8]. These abstractions and other assumptions do not always hold. In most systems, software timing behavior is not directly specified and is instead a consequence of implementation on a particular hardware platform—producing brittle designs where hardware selection is fixed. Even with timing behavior precisely known, the physical world is unpredictable, and software functionality must be robust to changes in the environment.

Despite these challenges, complex cyber-physical systems are built today and (usually) function correctly and safely. With *increasing system complexity*, however, the design, testing, and verification of future systems will be a major challenge. In the automotive domain, a high-end car produced in 2009 executes around 100 million lines of code on 70-100 microprocessor-based electronic control units (ECUs), with software and electronics contributing to 35 to 40% of the car's cost [9]. With massive investments in autonomous driving by leading technology and automotive companies worldwide, complexity will only further increase. Fundamental changes in the software and hardware design practices are needed to handle further increases in the complexity of cyber-physical systems.

Consolidating many increasingly complex software tasks onto fewer hardware platforms is one approach for reducing the cost, size, weight, and power (SWaP) of hardware (Figure 1.1). A single processor must then execute multiple tasks with differing importance, safety, or certification requirements—creating a *mixed-criticality system* [10] [11]. These requirements are often specified using criticality levels, such as up to five levels used in the ISO 26262 automotive standard [12] or the DO-178C avionics standard [13]. Unfortunately, *resource sharing increases interference* between tasks regardless of criticality level; testing and verification must then also account for timing behavior interference when determining correctness.

Timing requirements on the order of milliseconds is not precise enough for some applications, even though it is sufficient for many. Applications that use software to replace functionality typically provided by hardware or interact with other hardware devices require *precision-timed input/output (I/O)* with timing accuracy and precision on the order of nanoseconds, which is at the granularity of clock cycles for processors.

1.2 Objective

This dissertation investigates what new processor architecture could provide higher confidence in software functionality and timing behavior without sacrificing efficiency. A processor architecture consists of both the *instruction-set architecture (ISA)*, which is the contract between software and hardware, and the *microarchitecture*, which determines how the instructions of the ISA are implemented [14]. Although not directly addressed in this dissertation, there are also challenges at the software level: programming languages or models and the associated toolchains need to include timing semantics and handle the concurrency of the physical world [15]. However, any software approach relies on assumptions about the timing behavior of lower abstraction layers. Therefore, the underlying hardware, of which the processor is a key component, must either expose timing semantics or at least have analyzable timing behavior.

Current software approaches already make assumptions about the timing behavior of underlying hardware. For example, over 30 years of research in real-time scheduling theory assumes the computation time of every task is known or bounded [2], [7]. In practice, these bounds are important but can only be guaranteed for a restricted subset of programming techniques and processor architectures [16].

A current trend in both real-time software and hardware is the effort and cost of testing and verification surpassing that of design. Without higher confidence in correctness—by either exposing more information in the abstraction layers or improving analysis—testing and verification difficulties will limit system complexity. Additionally, new processor architectures must be efficient with respect to area, power, or performance; otherwise, it would be unsuitable for most industry application areas.

We evaluate different approaches by defining properties of isolation, timing predictability, timing control, and efficiency. We also assume an application is composed of *tasks*: each task performs computation, potentially using input data from sensors or other tasks and providing output data to actuators or other tasks. A task can execute multiple times with state saved between invocations, and it can be dependent on other tasks. For systems with differing requirements, such as mixed-criticality systems, each task will prioritize different properties.

Spatial and temporal isolation prevent independent tasks from being adversely affected by each other: spatial isolation protects a task's state (stored in registers and memory), and temporal isolation protects a task's timing behavior. With complete isolation, a task can be tested and verified for correctness by itself, then have identical behavior when deployed in a system with other tasks. The composability provided by isolation allows modular design practices that scale [17]. The *timing predictability* of an isolated task can assist *worst-case execution time (WCET)* analysis techniques in providing safe and tight bounds (above and close to actual) on WCET. Without tight bounds, system resources are over provisioned to maintain guarantees. *Timing control* refers to timing behavior being directly specified by software instead of a consequence of implementation on a particular processor.

We define *efficiency* as meeting requirements with minimal resources, often measured by resource area or time costs. With this definition, average-case performance could influence efficiency for a general-purpose system, whereas the bound on WCET could influence efficiency for a safety-critical system. The efficiency of an overall system may be different from the efficiency of a single task. The properties desired or required by each task vary, and the choice of underlying processor architecture affects feasibility of meeting these task requirements.

1.3 Problem

There are fundamental trade-offs between these properties. The main trade-off is between isolation and efficiency. On a platform where resources, such as processor execution cycles, are shared between tasks, a task invocation could use less than its allocated resources. If no other task uses this spare resource, overall system efficiency is not maximized; however, if another task uses this spare resource, isolation is reduced because a dependency is introduced into task behavior (on amount of spare resources). Although more nuanced, there are relationships between predictability and efficiency, depending on the underlying hardware. Many techniques that improve average task efficiency reduce predictability [16], and designing predictable yet efficient processor architectures is an active research area.

Existing approaches support trade-offs more at the processor level than task level—a problem for mixed-criticality systems where requirements vary by task. For example, a safety-critical task values isolation and predictability more than a low-priority logging task. Consider an example application that consists of three tasks, each initially executing on separate processors. Figure 1.2 shows a possible run-time trace over a segment of time, where shaded boxes indicate a task is executing. An up arrow indicates the *release* time when a task can start executing, and a down arrow indicates the *deadline* time when it must complete; an up and down arrow indicates the deadline of the task instance is also the release time of the next task instance. All tasks start at the release time because each is on a separate processor with no resource conflicts.

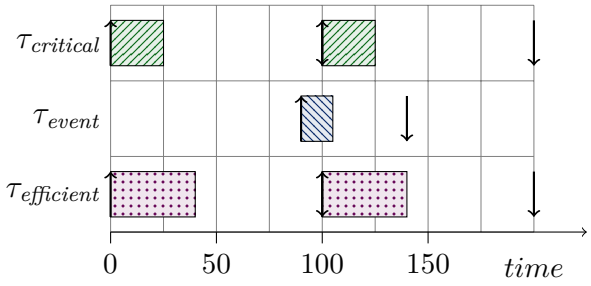


Figure 1.2: Possible run-time behavior for an example application with three tasks, each on a separate processor.

The first task $\tau_{critical}$ performs safety-critical feedback control of a physical process using sensors and actuators. For stability of the physical process, the task must execute periodically and finish before the next period (indicated by arrows). For high confidence in functionality and timing behavior, the task should be isolated and predictable. The second task τ_{event} uses an interrupt-driven approach to detect an external event (demonstrated by the up arrow at time 90). Each infrequent event must be handled rapidly without wasting resources, so interrupts are used instead of polling. This task could change the mode of operation for the critical task, so it should also be isolated and predictable. The third task $\tau_{efficient}$ has less stringent requirements than the other two tasks. If resources are shared between these three tasks, this task can use spare resources to improve overall efficiency, unlike the other two tasks with isolation requirements. This task could perform high-level planning or logging.

An example of this type of application in the automotive domain is a yaw moment controller, which prevents a vehicle from spinning or drifting out. The $\tau_{critical}$ task could be the control algorithm, the τ_{event} task a mode change for emergency braking, and the $\tau_{efficient}$ task for approximating vehicle and environmental variables used by the controller. Tasks $\tau_{critical}$ and τ_{event} are *hard real-time* since a deadline miss must never occur, and task $\tau_{efficient}$ is *soft real-time* since a deadline miss is acceptable but can degrade functionality. In feedback control systems, meeting deadlines may not be the only timing constraint; variations in the latency between sensing and actuation called *jitter* can also affect the performance of a controller with respect to responsiveness and stability [18], [19].

Worst-Case Execution Time (WCET) Analysis

Some challenges and limitations of existing approaches are exposed using this example application. The difficulty of *worst-case execution time (WCET) analysis* [8] on task computation depends on properties of the program and the processor architecture. Analysis techniques can statically compute possible paths through a program, but determining the longest path requires assigning latency costs to different program segments. For certain microarchitectures, the latency of certain instructions in a program segment depends on program and processor state. Consequently, *timing behavior depends on execution history* and can be difficult to tightly bound; most hardware prediction mechanisms for improving average case performance, such as dynamic branch prediction and caching, exhibit this property [20]. For a single uninterrupted task, WCET analysis techniques are still able to handle some complex processors that use hardware prediction mechanisms [21], [22], and several tools are used in academia and industry [23].

The underlying assumption that task execution is uninterrupted is restrictive and often violated, however, due to the prevalence of interrupts for event handling and task preemption [20], [24]. The number and duration of interrupts affect the timing behavior [25], and more problematic is that an interrupt can affect processor state, particularly the caches [26]. The effects on processor state must be bounded [27], isolated [28], or prevented by restoring original state [29]. As a consequence, interrupts are prohibited in many safety-critical applications. Not only does this exclude many approaches from real-time scheduling theory but

also eliminates a common I/O mechanism in systems whose primary function is to interact with the physical environment.

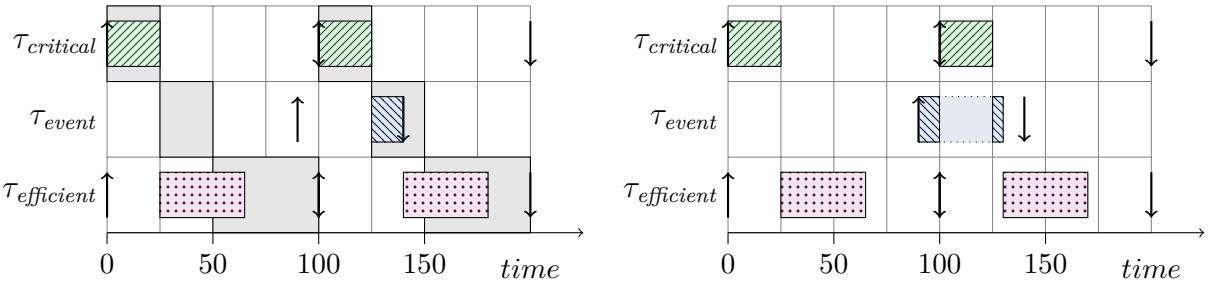
Architectures that remove the mechanisms with instruction latency dependence on execution history are more predictable but potentially also sacrifice the performance gains of decades of research in computer architecture. Researchers have proposed predictable processor pipelines [30]–[33], but the predictability of the memory hierarchy is also important because memory accesses are usually the main source of execution time variations. Instead of a cache, processors can use *scratchpad memories*, which are local memories with contents controlled explicitly by software [34], [35]. There is active research on scratchpad memory management techniques to reduce WCET, but providing performance similar to a cache is challenging [36]. Caches can also be used in a more predictable way, such as locking cache lines [28], caching entire functions [37], or using separate caches for the stack and heap [37].

Real-Time Software Scheduling

For the example application with three tasks, the selected processor's predictability and efficiency must support a WCET bound that meets the deadlines of tasks $\tau_{critical}$ and τ_{event}. These tasks also have an isolation requirement. The standard industry approach—without using an entire processor or core for each task and suffering the associated efficiency loss—is *software scheduling* on a single-threaded processor. A scheduling algorithm selects which task executes for different segments of time, as there can only be one executing. A software real-time operating system (RTOS) implements a scheduling algorithm, and it can also provide mechanisms for communication and synchronization between tasks and I/O functionality. The RTOS must be carefully designed and verified because it enforces software-based isolation and affects the timing behavior of all tasks.

Software scheduling is typically either reservation-based or priority-based, and both approaches are used in mixed-criticality systems [38]. A *context switch*, where software saves the state of one task and restores the state of another, occurs when a *preemptive* scheduler switches between uncompleted tasks using interrupts. In the reservation-based approach [39], [40], the scheduler allocates segments of time to particular tasks. Figure 1.3a shows a reservation-based scheduling of the three tasks in the example application. Tasks $\tau_{critical}$ and τ_{event} are both isolated by only executing during their own reserved time segment, and task $\tau_{efficient}$, depending on scheduler implementation, can use the reserved time segments of other tasks if they finish early (as demonstrated from time 25 to 50). The disadvantage of this approach is the responsiveness of τ_{event} could be insufficient to meet requirements. To maintain isolation, τ_{event} only executes during its allocated time segment, even if an event occurs outside of it, as seen by τ_{event} barely meeting the deadline (at time 140). The granularity of reservations for time segments affects the responsiveness of τ_{event}, and the overhead of context switches prevents shorter time segments.

In the priority-based approach [10], [11], every task has a priority level, and the scheduler always executes the highest priority task. Only the task with the highest priority is fully isolated, as the scheduling of all lower priority tasks depends on the execution behavior of



(a) Reservation-based scheduling, where reservations are shown by grayed boxes.

(b) Priority-based scheduling, where $\tau_{critical}$ has highest priority and $\tau_{efficient}$ the lowest.

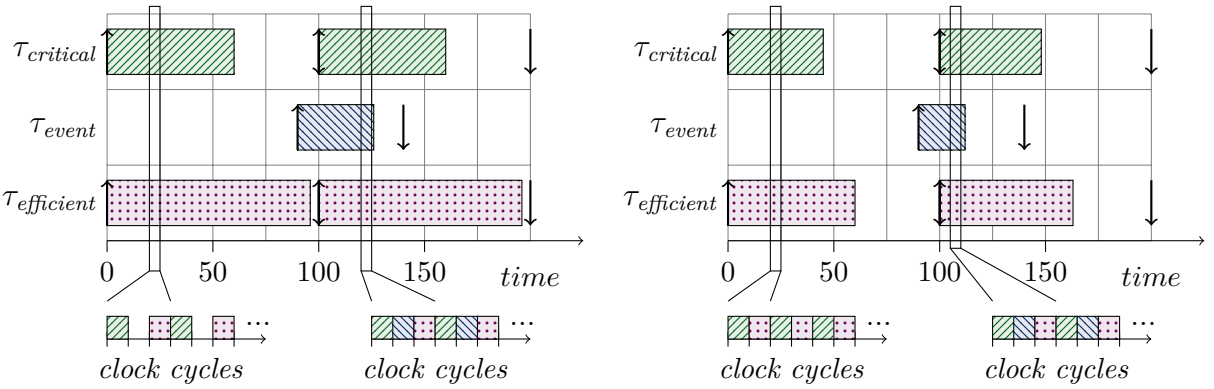
Figure 1.3: Possible run-time behavior for the example application using software scheduling on a single-threaded processor.

the highest priority task. Figure 1.3b shows a priority-based scheduling of the three tasks in the example application. Task $\tau_{critical}$ is isolated, task τ_{event} starts reacting immediately but is preempted before it completes (at time 100), and task $\tau_{efficient}$ uses resources when the other tasks do not need to execute. As with reservation-based scheduling, the responsiveness of task τ_{event} could not meet requirements because it depends on the execution time of task $\tau_{critical}$ to preserve isolation. These scheduling examples demonstrate a limitation of software-based scheduling on single-threaded processors: one task must always have priority, so the timing behavior of other tasks is either dependent or restricted. The RTOS implementation and any preemption effects on processor state, albeit limited in this simple example, must be considered during worst-case execution time analysis.

Multithreaded Processors

An alternative approach is fine-grained multithreading (interleaved multithreading, barrel processor), where instructions from different hardware threads are interleaved in the processor pipeline every cycle [14], [41], [42]. A hardware thread is logically a separate processor with its own program counter and registers, but it shares the pipeline with other hardware threads. For example, 4 hardware threads appear to each execute at 25 MHz if interleaved on a 100 MHz pipeline (each hardware thread enters the pipeline once every 4 cycles). Each hardware thread either executes a single task or uses software scheduling for multiple tasks. By executing multiple tasks in parallel but each at a slower rate, the processor has higher *overall* efficiency because instruction latencies are hidden, reducing the burden on branch prediction and the memory hierarchy. Most fine-grained multithreaded processors require a minimum number of hardware threads in the round-robin rotation, such as four, to reduce hardware costs.

Fine-grained multithreading enables hardware-based isolation between tasks that are deployed to separate hardware threads, but isolation still depends on hardware thread scheduling, which is inflexible in many processors, such as PTARM [43] and XMOS [44]. One



(a) Fixed round-robin hardware thread scheduling.

(b) Active round-robin hardware thread scheduling.

Figure 1.4: Possible run-time behavior for the example application using different hardware thread scheduling techniques on a fine-grained multithreaded processor.

technique, used by the PTARM processor, is *fixed round-robin* scheduling, which alternates between all hardware threads regardless of status. Figure 1.4a shows fixed round-robin scheduling for the three tasks in the example application, each deployed on separate hardware threads. The lower timelines of Figure 1.4 show which hardware thread is scheduled each cycle for a few time segments. All tasks are isolated, and task τ_{event} can start executing within several cycles of an event occurring, albeit at a slower execution rate. The disadvantage of this approach is that when tasks $\tau_{critical}$ and τ_{event} are not executing, task $\tau_{efficient}$ is still only using one out of every three clock cycles, and the other two are not used.

Another technique, used by the XMOS processor, is *active round-robin* scheduling, which alternates between all hardware threads that are ready to execute. Figure 1.4b shows active round-robin scheduling, assuming the pipeline supports any number of interleaved hardware threads, including just one (XMOS requires four). No cycles are left unused whenever any task needs to execute. For example, at different points task $\tau_{efficient}$ is scheduled every cycle, every second cycle, and every third cycle. The disadvantage is none of the tasks are fully isolated because the execution frequency of each hardware thread depends on the number of active threads. The properties of fixed round-robin scheduling are more suitable for tasks $\tau_{critical}$ and τ_{event}, and the properties of active round-robin scheduling are more suitable for task $\tau_{efficient}$, but only one scheduler can be selected for the whole system.

In summary, software scheduling must handle the effects of interrupts and cannot provide hardware-based isolation or responsiveness for multiple tasks, and inflexible hardware thread scheduling in a fine-grained multithreaded processor forces a decision between isolation and efficiency.

1.4 Contributions

This dissertation presents FlexPRET, a processor designed to provide thread-level trade-offs between predictability, hardware-based isolation, and efficiency in order to simplify the verification of safety-critical tasks, allow software to meet precise timing constraints, and reduce hardware costs. The main architectural features are flexible thread scheduling for fine-grained multithreading, tightly coupled timers exposed in the ISA, and instruction latencies that are independent of execution history.

Similar to virtual machines in general-purpose computing, the hardware threads in FlexPRET can be considered *virtual real-time machines*; they provide either precise or minimum guarantees on execution resources at the cycle level and hardware support for mechanisms typically provided by a real-time operating system. Each hardware thread can be programmed using different languages or techniques and isolated from interrupts occurring on other hardware threads. Instead of trying to minimize interference between mostly independent tasks, a programmer can use isolated virtual real-time machines for composability and predictability.

Hardware threads are classified as either *hard real-time threads (HRTTs)* or *soft real-time threads (SRTTs)*, and each executes either one task or multiple tasks with software scheduling. Unlike other fine-grained multithreaded processors [43], [44], FlexPRET supports an arbitrary interleaving of threads in the pipeline and uses a novel thread scheduler. HRTTs are only scheduled at a constant rate, and SRTTs can be scheduled at a minimum rate. 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. For control over timing, a variation of timing instructions [43], [45], [46] allows software to stall until or interrupt execution at a point in time, again allowing SRTTs to use spare cycles for efficiency. For predictability, the processor uses predict not-taken branching for all hardware threads and scratchpad memories instead of caches for at least HRTTs.

The main techniques are demonstrated by the example application when tasks $\tau_{critical}$ and τ_{event} are each deployed on a HRTT, and $\tau_{efficient}$ is deployed on a SRTT. Figure 1.5 shows a possible execution of this example application, with the scheduler prioritizing each hardware thread once every third cycle. Both tasks $\tau_{critical}$ and τ_{event} are isolated with predictable behavior (execute once every three cycles), and task τ_{event} will react within a few cycles to an event. Furthermore, task $\tau_{efficient}$ uses all available cycles when either task $\tau_{critical}$ or τ_{event} is not executing because it is a SRTT (e.g. at time 20). When task τ_{event} starts executing (at time 90), its only scheduled every third cycle to preserve isolation.

Other scheduling configurations are possible depending on task requirements. This trade-off between task properties is not achievable using existing processor architectures or software techniques. For this processor, mixed-criticality timing requirements are a benefit instead of hindrance: less critical tasks use spare cycles that isolated critical tasks cannot use to improve overall efficiency.

We define a set of tasks with low individual task throughput (compute and memory) requirements and strict isolation requirements as an *isolation-bound task set*, and FlexPRET

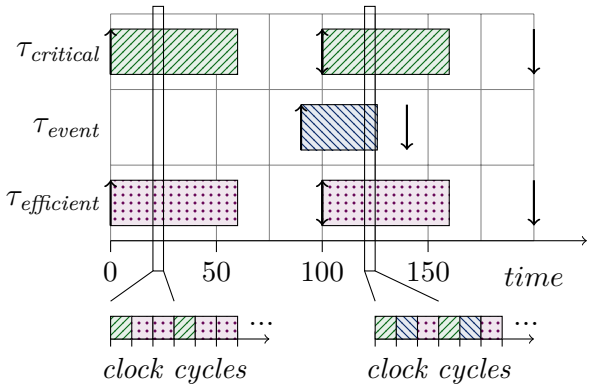


Figure 1.5: Possible run-time behavior for the example application using FlexPRET's hardware thread scheduling.

is designed for these task sets. While this is an unreasonable assumption for many general-purpose applications, it is not for many cyber-physical applications. The field of real-time scheduling assumes task sets consist of dozens or hundreds of tasks, so most of these tasks must have low throughput requirements with a sufficient amount of parallelism. FlexPRET does not require any parallelism, however, because it can operate as a simple single-threaded processor by scheduling only one thread.

Some applications have tasks that are not well suited for FlexPRET; for example, a task with a throughput requirement not achievable using a single execution pipeline. We envision a system where one or multiple FlexPRET cores execute the isolation-bound tasks as *coprocessors*, and the other tasks execute on either a predictable processor with higher single-threaded throughput, such as Patmos [47], or a conventional embedded processor, such as an ARM Cortex M-7 or R-4.

Work in real-time software scheduling, worst-case execution time analysis, and real-time programming languages influence the design decisions (Chapter 2). For a concrete implementation, a 32-bit, 5-stage pipeline for the RISC-V ISA is augmented with the proposed techniques (Chapter 3). To accurately evaluate performance and area costs, the processor design is specified in Chisel, which generates Verilog for both FlexPRET and comparison processors, and deployed as a soft-core on FPGA (Chapter 4). FlexPRET is also evaluated using some example applications from the mixed-criticality and precision-timed I/O domains (Chapter 5). This work provides the underlying processor to support higher confidence in future techniques at the hardware and software levels (Chapter 6).

In summary, the contributions presented in this dissertation are:

- A processor design that uses microarchitectural techniques for thread-level trade-offs between predictability, hardware-based isolation, and efficiency (Chapter 3).
  - Flexible hardware thread scheduling with pipeline support that differentiates between hard and soft real-time execution requirements.

  - Timing instructions that enable software specification of timing behavior and reallocation of spare resources.

- A programming methodology that conceptually uses hardware threads as virtual real-time machines, potentially underneath an RTOS (Chapter 5).
  - Precise or minimum allocation of processor cycles for each virtual entity.
  - Ability to isolate a hardware thread from interrupts on other hardware threads.
  - Hardware support for features typically provided by an RTOS.

- An evaluation and demonstration of the FlexPRET processor design and programming methodology (Chapters 4 and 5).
  - FPGA implementation with area usage compared with two baseline processors.
  - Mixed-criticality application from the avionics domain that is mapped to different hardware threads that use software scheduling to support more tasks than threads.
  - Software that uses precision-timed I/O operations for bit banging to replace hardware peripherals.

Chapter 2

Background and Related Work

This chapter presents a summary of some existing approaches used to design and verify cyber-physical and real-time embedded systems. The shortcomings of these approaches motivate and support the processor architecture and programming methodology presented in this dissertation. These systems are implementable with commercial-off-the-shelf (COTS) processors, real-time operating systems (RTOSs), and ad-hoc programming and testing. Most RTOSs schedule different tasks according to assigned priorities and support interrupts for timely responses. By tweaking priorities, coding fast interrupt handlers, disabling interrupts during critical code segments, and using timer peripherals, engineers can develop usable systems. The resulting system design, however, is often difficult to understand, maintain, and verify. Consequently, researchers have investigated more rigorous approaches at both the software and hardware levels for design and verification of these systems.

On the software side, research in real-time software scheduling has produced theoretical guarantees and improved performance in several metrics when assumptions regarding the application are met (Section 2.1). One of the key assumptions is known bounds on worst-case execution time (WCET) for tasks, which in general is undecidable. The field of WCET analysis uses static analysis of program code with either hardware models or measurement techniques to bound the WCET of program fragments for verification and optimization purposes (Section 2.2).

Instead of conforming an application to a model required by traditional real-time software scheduling and checking the assumptions through analysis and testing, new programming languages and models can more directly specify the timing and functionality of real-time systems (Section 2.3). Also, instead of modeling or attempting to control unpredictable hardware mechanisms, predictable hardware designed specifically for real-time systems can simplify programming and analysis and improve worst-case performance (Section 2.4), which is the focus of this dissertation.

2.1 Real-Time Software Scheduling

If an application consists of multiple tasks, implicit or explicit *scheduling decisions* determine when each task executes on a processor, and on which processor in a multiprocessor system. In real-time systems, scheduling must satisfy each task's timing requirements—part of execution correctness—in addition to any ordering requirements from dependencies between tasks. Even with extensive testing, a scheduling algorithm without theoretical guarantees can fail to meet timing requirements in certain situations, sometimes with catastrophic results.

The field of real-time scheduling theory uses *task models* to describe an application, including its timing requirements, and investigates the properties of different scheduling algorithms. For task sets that meet the required criteria, a scheduling algorithm guarantees correct timing behavior in all possible circumstances. The most suitable scheduling algorithm depends on the requirements and properties of the application and the scheduling overhead on the target platform. This section only gives a high-level overview, and more in-depth surveys exist for both uniprocessor [2], [7] and multiprocessor [48] systems. Software scheduling for mixed-criticality systems is covered later in Section 5.1, and FlexPRET uses software scheduling algorithms for executing multiple tasks on a single hardware thread.

2.1.1 Task Model

Many real-time applications have cyclic components, such as feedback control, with repeating tasks. In a common task model [2], an application consists of a set of n *tasks* (τ_1, τ_2,..., τ_n). A task is a potentially infinite sequence of *jobs*, and each job is ready for execution at some *arrival time*, or *release time*. If the time separation between successive jobs of a task is constant, the task is *periodic*. If the time separation, or *interarrival time*, is not constant but has a minimum value, the task is *sporadic*. Without any limitations on interarrival time, timing guarantees are not possible because many jobs could arrive simultaneously in the worst-case scenario and overload the processor.

Each task τ_i is characterized by a period or minimum interarrival time T_i, *relative deadline* D_i, and *worst-case execution time* C_i. The relative deadline is often equal to the period, but this is not required for all scheduling algorithms. Although essential for timing guarantees, the worst-case execution time cannot always be tightly bounded (Section 2.2).

The parameters and time values for a task invocation are shown in Figure 2.1. At arrival time a_i, the task is ready for execution. Depending on scheduling decisions, the start time s_i of execution can be later than the arrival time. If preemption is allowed, the task may not execute continuously until completion, at finish time f_i. The time duration between arrival and completion is referred to as the *response time* R_i. A *deadline miss* occurs if the finish time is after the *absolute deadline* d_i (response time is greater than the relative deadline).

A task can also be categorized by the consequence of missing a deadline, with each task either hard, firm, and soft real-time. For a *hard* real-time task, a deadline miss can be catastrophic; for a *firm* real-time task, a deadline miss is harmless but the result is useless; and for a *soft* real-time task, a deadline miss can cause degraded functionality. Consequently,

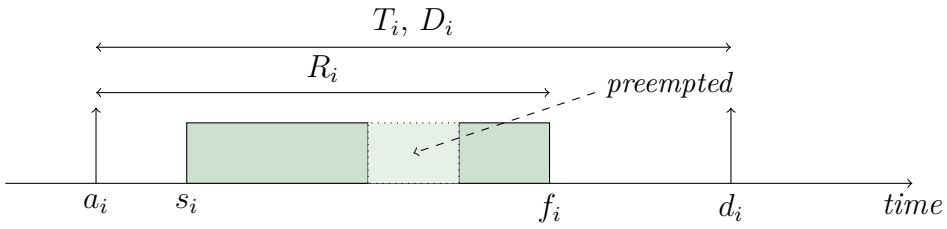


Figure 2.1: Parameters and time values for a task.

a scheduling algorithm should guarantee no deadline misses for hard real-time tasks, but it may accept deadline misses for firm or soft real-time tasks to improve overall performance or meet hard real-time deadlines.

A scheduling algorithm can also be categorized. *Priority-based* schedulers assign each job a *priority* and always select the highest priority to execute. A scheduling algorithm is *fixed-priority* if every job of a task has the same priority; otherwise it is *dynamic priority*. Fixed-priority scheduling can be simpler to implement and analyze but less capable of meeting all deadlines. Another distinction is whether a scheduling decision can occur during the execution of a task: a *non-preemptive* scheduler waits until a task completes, while a *preemptive* scheduler can interrupt and execute a different task. Preemption complicates implementation and analysis, but allows the scheduler to prevent blocking of a high-priority task.

A *feasible schedule* meets all timing requirements for a given task set. A task set is *schedulable* by a scheduling algorithm if the resulting schedule is feasible. The most practical scheduling algorithms provide schedulability for many different task sets, not just one. A useful property within a class of scheduling algorithms is *optimality* with respect to feasibility—if any other scheduling algorithm produces a feasible schedule for a task set, so does it.

Schedulability tests are important metrics in real-time scheduling theory, for both task set guarantees and algorithm comparisons, and exist for both individual and classes of scheduling algorithms. A test is *necessary* if all task sets that fail are unschedulable, and a test is *sufficient* if all task sets that pass are schedulable. The *utilization* u_i of a task is the worst-case fraction of time the processor must spend executing it ($u_i = C_i/T_i$). For n periodic tasks, the *processor utilization* is:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

A common sufficiency test for a scheduling algorithm A determines some processor utilization bound U_A. Any task set with utilization less than the bound ($U < U_A$) is schedulable—and guaranteed to meet timing requirements—otherwise schedulability must be tested another way.

Most schedulability tests assume zero overhead for scheduling decisions and switching between tasks, but RTOS overheads exist and vary between implementations [49]. In addition to execution time overhead, changes in processor state at preemption points should

be considered by WCET analysis. As a consequence, an inferior scheduling algorithm with respect to feasibility could be preferable if it provides fewer and more predictable preemption points.

2.1.2 Uniprocessor Scheduling

The foundational task model for real-time scheduling theory is periodic, independent tasks executing on a uniprocessor with relative deadlines equal to respective task periods. The first common scheduling approach was reservation-based and called *cyclic executive*, or timelime, scheduling [50]. In this approach, time is divided into slots of set length, and tasks are assigned to slots. The schedule is constructed and optimized offline and stored in a table. During run-time, a periodic interrupt routine, triggered by a timer, performs scheduling decisions using the table in memory. Some advantages are a simple run-time implementation and deterministic and isolated task timing behavior. Some disadvantages are a feasible schedule can be non-trivial to construct and fragile to task set or run-time variations. As with all static allocations, unused time fragments are wasted if not explicitly handled.

Dynamic, priority-based schedulers better handle task set or run-time variations. In their seminal work, Liu and Layland [51] showed that rate monotonic (RM) scheduling—where tasks with smaller periods have higher priorities—is optimal with respect to feasibility for fixed-priority scheduling on this foundational task model. The key insight was the *critical instant theorem*, where the worst-case response time of a task occurs when it arrives simultaneously with all higher priority tasks (again, for this task model). This reduces feasibility checking to only considering one combination of arrival times. Also, a sufficient schedulability test for n tasks is:

$$U \leq n(2^{1/n} - 1) \approx ln(2) = 0.69 \ as \ n \rightarrow \infty$$

Later work relaxes some constraints on the task model, improves the sufficient schedulability test, and investigates special conditions. For example, if the task periods are harmonic, schedulability is guaranteed for up to 100% utilization.

Dynamic-priority scheduling algorithms are also analyzable for this foundational task model. One scheduling algorithm is *earliest deadline first (EDF)*, where the task with the earliest absolute deadline has the highest priority. EDF is optimal with respect to feasibility for all preemptive schedulers, with a sufficient schedulability test for n tasks as $U \leq 1$ [51]. As with rate monotonic scheduling, later work relaxes some constraints on the task model while preserving optimality.

Although the sufficient utilization bound is higher for EDF than RM scheduling, implementation complexities still result in extensive use of RM scheduling—particularly in commercial real-time kernels. Buttazzo [52] evaluated the differences between EDF and RM scheduling and discussed some common misconceptions. In practice, EDF requires less preemption than RM scheduling, particularly with high processor utilization. Consequently, overall overhead can be lower even though the overhead of each EDF scheduler invocation is higher.

By guaranteeing deadlines are met for all period tasks, the discussed priority-based schedulers are well suited for hard real-time tasks. For periodic or sporadic soft real-time tasks, however, average response time and high utilization are often more important than worst-case guarantees. Scheduling soft real-time tasks using worst-case execution time and interarrival time (C_i and T_i) over provisions the system when the average times are much lower than the worst-case times and does not reduce response times. Also, aperiodic soft-real time tasks cannot be allowed since interarrival time is unbounded.

A simple way to handle aperiodic tasks is to execute them in the background when periodic tasks are not executing, but the response time of aperiodic tasks is then the lowest priority. An alternative is using a pseudo hard real-time task, called a *server*, that can be scheduled with a higher priority to service aperiodic tasks. *Capacity* levels prevent the aperiodic server task from causing deadline misses of periodic tasks. Dynamically elevating priority to improve response time without causing a deadline miss would require online schedulability analysis, so most methods instead use approximation and precomputation techniques.

Tasks are not all independent in many real-time applications and share resources such as data structures, buffers, and hardware peripherals. The section of code that accesses a shared resource is called the *critical section*, and *mutual exclusion* prevents simultaneous access to a shared resource by only allowing one task to be in its critical section at any time. Blocking a task to preserve mutual exclusion changes run-time behavior, so schedulability analysis must bound the worst-case blocking time. In fixed-priority scheduling, blocking is called *priority inversion* because when a low-priority task blocks a high-priority task it is effectively executing at a higher priority. If an intermediate priority task preempts this low-priority task, the high-priority task can be blocked forever. Solutions to this priority inversion problem, such as the Priority Inheritance Protocols [53], modify priority levels to bound blocking time.

2.1.3 Multiprocessor Scheduling

Multiprocessor scheduling is a more difficult problem than uniprocessor scheduling, with many uniprocessor scheduler properties not generalizing to multiprocessor equivalents. A source of difficulty is *scheduling anomolies*, where unexpected behavior results from certain changes. For example, increasing the number of processors or reducing execution times can actually increase response time [54]. In addition to assigning either fixed or dynamic priorities to tasks, multiprocessor scheduling algorithms must allocate tasks to specific processors. Options include restricting a task's jobs to a single processor (*paritioned*), allowing each job to execute on different processors (*global*), or even allowing a job to move between processors when preempted.

In the partitioned approach, task allocation is difficult—analogous to the NP-Hard bin packing problem [55]—but analysis and implementation are easier. Once allocated, each processor uses local uniprocessor scheduling. For global scheduling, Dhall and Liu [56] showed that the utilization bound for EDF is only $1 + \epsilon$ for m processors. This temporarily reduced

interest in global scheduling, even though it only occurs with high utilization tasks. Although scheduling information and tasks must be shared between processors, global scheduling better uses spare capacity and often has fewer preemptions [57]. As with uniprocessor scheduling, aperiodic tasks and resource sharing are handled by extending scheduling algorithms. Even with schedulability guaranteed for a task set on a uniprocessor or multiprocessor system, the WCET assumptions for each task must also be verified.

2.2 Worst-Case Execution Time (WCET) Analysis

There are different types of timing requirements for software execution. Some tasks require operations at particular points in time within accuracy and precision bounds; an example is precision-timed I/O operations that use software to communicate with hardware components (Section 5.2). Other tasks only need to execute with a *bounded* latency before the deadline—determined by worst-case execution time (WCET) analysis techniques [8], [20].

Each task has a distribution of execution times resulting from different inputs and initial states, with an example shown in Figure 2.2. The distribution has actual best, worst, and average-case execution times (BCET, WCET, and ACET). Testing all possible combinations of input and initial state is infeasible, and the measured execution times from some test cases are likely above the BCET and below the WCET. The ad-hoc approach is trying to select test cases that exercise the critical path then adding a safety margin to produce the upper bound (WCET Bound); this bound is not *safe* if still below the actual WCET.

Properties of the program, compiler optimizations, and the processor architecture all affect both a task's WCET and WCET bound. The difference between the WCET and its bound depends on the employed analysis techniques and causes an over provisioning of resources. Tightly bounding the execution time of the longest path out of an exponential number of possible paths is not trivial but timing-predictable hardware (Section 2.4) can simplify the effort.

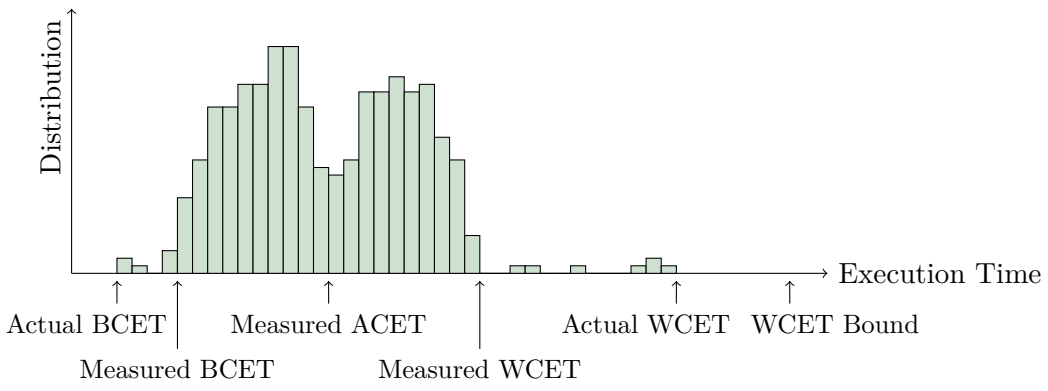


Figure 2.2: Fictitious distribution of execution times for a task resulting from different inputs and initial states.

In general, the worst-case input and initial state are not known and analysis must consider all possible input cases. Therefore, to provide a bound on WCET, analysis has to consider both the possible paths through a program and the execution time bounds for this set of paths. Consider this simple code segment in C.

```
1 for (i = 0; i < n; i++)      // loop bounds?
2     if (a[i] > b[i])         // conditional branch
3         c[i] = a[i];         // memory accesses
4     else
5         c[i] = b[i];
```

For even this simple code segment, a compiler can produce many different correct sequences of instructions, depending on flags and optimizations. For example, loop unrolling increases code size but executes fewer branch instructions by performing multiple iterations per loop. Consequently, analysis often operates on the binary executable instead of the higher-level program code.

2.2.1 Static Methods

Analysis methods can be classified as predominantly either static or measurement-based, even though most tools use both methods. Static methods analyze code without executing it on the target hardware by using abstractions and often provide safe bounds on WCET, provided assumptions hold. Static WCET analysis can be loosely categorized into three steps: (1) obtain information about control and data flow through the program, (2) assign timing costs by modeling hardware behavior, and (3) find the longest path through the program using the prior results.

1. A *control flow graph* represents the superset of all execution paths and is often reconstructed from the binary executable. Each graph node is a *basic block*, which is a sequence of instructions without any jumping or branching, and the graph edges represent possible jumps or branches in control flow. Dynamic calls and jumps, where the destination is computed during run-time, are difficult to represent: without any knowledge of possible values, the instruction could branch to any other instruction in memory. Restricting values of loop variables, such as n in line 1 of the code example, is also important for low WCET bounds.

To handle these situations, *value analysis* techniques determine exact or intervals for values in registers and memory at different program points. This information is then used by *control flow analysis*, which bounds loop iterations and determines paths that cannot execute (infeasible), and later for determining data-specific instruction latencies, as found in data caches and certain arithmetic instructions. Abstract interpretation [58], [59] is one methodology to perform these analyses.

2. Once possible paths are determined, the execution time bound for a path *depends on hardware architecture*. Furthermore, the execution time for a code segment could depend on processor state when this code segment is reached. This is common in conventional architectures with hardware prediction mechanisms used to improve average-case performance.

Dynamic branch prediction uses execution history to predict whether a branch will be taken or not taken, such as for the *if* statement on line 2 in the code example. For a program path, the number of cycles required to complete the branch instruction depends on whether the prediction is correct or not, which depends on execution history. The execution time to complete a memory instruction, such as the array accesses on line 2 in the code example, depends on the current content of the caches, which also depends on execution history.

Pipeline and cache analysis techniques use program analysis and hardware models to predict processor state at different program points and then bound execution time of basic blocks [60]. Predictable architectures without instruction latency dependence on execution history do not require these additional analyses, but do not always meet performance requirements.

3. Once execution times are known for basic blocks, finding the longest path provides the WCET bound. One method for finding the longest path uses integer linear programming (ILP), where constraints represent control flow and the objective function that sums up execution time is maximized [61].

There are many challenges in WCET analysis, particularly regarding analysis of the microarchitecture. *Timing anomalies* can occur where local worst case behavior is not global worst case behavior [62]. An example is a cache hit causing longer execution time than a cache miss because of speculative operations affecting later instructions. Also, preemptive scheduling affects task execution times and must be bounded [27] or restricted [29]. Timing-predictable hardware that prevents timing anomalies and simplifies microarchitectural analysis is discussed in Section 2.4.

2.2.2 Measurement-based Methods

Measurement-based approaches measure the execution time of segments of code and combine that with analysis of the program to computes estimates of WCET. The tool collects execution time values either using a simulator or adding code that runs on the target hardware, which should have a minimal effect on execution for accurate results. The main disadvantage is potentially unsafe bounds resulting from not considering all paths or not considering execution time variations caused by the microarchitecture. The main advantage is not needing a processor model, as incorporating complex processor models into static analysis limits scalability. The models are not accurate without access to the hardware design itself and also time-consuming to produce for complex processors.

RapiTime [63] is a commercial measurement-based tool that provides execution time measurements and identifies potential worst-case paths for optimization. Seshia et al. [64]–[66] have also developed a measurement-based toolkit called GameTime for execution time analysis that uses game-theoretic online learning and systematic testing. The approach extracts a representative subset of feasible program paths called *basis paths* and uses an SMT solver to generate corresponding test cases that cause execution to follow these paths. By measuring execution times for these different tests cases, a learning algorithm generates a platform model that is used to accurately predict timing behavior.

2.3 Programming with Time

An instruction set architecture (ISA) is a contract between software and hardware that defines a set of instructions the processor must execute properly. By not over-constraining the details of instruction implementation, processors can have different microarchitectures for different design goals but still share a software toolchain. The problem for real-time systems is that commonly used ISAs do not specify the program's temporal behavior—which is highly dependent on the internal processor design and toolchain optimizations. An accurate model of the hardware, which is not always available or easy to produce, is required to analyze the timing behavior.

Programs can still specify timing, just not directly. Most processors have external timers, accessible with memory-mapped registers, that can return tick counts and interrupt the processor at specified tick counts. Even still, the timing behavior depends on many factors: external timer design and configuration, interrupt priority and handler routine, and current processor state. RTOS libraries can hide some complexity but must be ported for each platform. For cycle-accurate timing, the programmer or compiler can selectively insert NOP (no operation) instructions, but the timing behavior of the hardware must be well understood.

Some programming languages and models include timing constructs, such as Ada [67], Real-time Euclid [68], synchronous languages [69], and LabVIEW [70]. The specification of timing behavior in a program for a language does not guarantee execution has this timing behavior—it depends on the implementation of the software toolchain and target hardware. Most languages with timing constructs would benefit from more directly controllable timing to reduce the burden on compilers or timing analysis tools.

An example of a programming model that would benefit by more control over timing is PTIDES [71], which specifies functionality and timing of an event-triggered distributed system. Variations in network latencies between distributed platforms and computation times within each platform can result in non-deterministic distributed behavior—a trace of input values at specific times to sensors results in different possible traces of output values at actuators—but PTIDES provides deterministic behavior. To accomplish this, each platform has access to a real-time clock that is synchronized to all other platforms, potentially on the order of nanoseconds with IEEE 1588 [72]. Each event has a timestamp that is only related to real-time at sensors and actuators: the timestamp of an event from a sensor is when it occurred, and the timestamp of an event to an actuator is when it should occur, so timestamp modification specifies end-to-end latency. The passage of time provides information to a platform, such as absence of events, and timestamps preserve ordering even with latency variations. Although PTIDES can be implemented on traditional processors with timer peripherals and interrupts [73], more direct control and access to a real-time clock would reduce overhead and improve accuracy and precision.

2.3.1 Timing Instructions

One possible solution is to add temporal properties to some instructions in the ISA. The challenge is expressing timing behavior without over-constraining the microarchitecture; for example, assigning constant latencies to every instruction would prevent any design from improving performance. Ip and Edwards [45] first proposed an instruction to specify a minimum time for the execution of a section of code, enabling simple software to implement functionality traditionally done by hardware, demonstrated by a small video display controller. Their *deadline* instruction would stall execution until the associated timer, decremented by one every clock cycle, reached zero, then set the timer to a new value. A faster processor can execute software with the same timing behavior, provided the cycle counts are scaled. Lickly et al. [74] implemented the same *deadline* instruction on a fine-grained multithreaded processor.

Timing instructions can control temporal behavior more than just a minimum execution time for a section of code. Bui et al. [46] identified four cases that timing instructions should handle:

1. A minimum execution time for a block of code
2. If a block of code exceeds an execution time, branch after it completes
3. If a block of code exceeds an execution time, branch immediately
4. A maximum execute time for a block of code

The first three cases can be expressed with the following four pseudo-instructions, generalized from instructions proposed in previous work [43], [46], [75]:

- `get_time`: load current time from a clock into a register
- `delay_until`: stall program until current time exceeds the value in provided register
- `interrupt_on_expire`: interrupt program when current time exceeds the value in provided register
- `deactivate_interrupt`: deactivate `interrupt_on_expire`

The following pseudo assembly code demonstrates how a control flow change can immediately occur if a code block's execution time exceeds a value (case 3) and also enforces a minimum time once the code block completes (case 1).

```
1 get_time r1              # get current clock time
2 addi r1, r1, 1000000     # compute time 1 ms in future
3 interrupt_on_expire r1   # interrupt will trigger in 1 ms
4 ... code ...             # interrupted if exceeds 1 ms (#3)
5 deactivate_interrupt     # otherwise deactivate
6 delay_until r1           # stall until 1 ms since get_time (#1)
```

A branch can also occur by comparing against the clock time after completion (case 2).

```
1 get_time r1              # get current clock time
2 addi r1, r1, 1000000     # compute time 1 ms in future
3 ... code ...             # execute block of code
4 get_time r2              # get updated clock time
5 bgt r2, r1, miss         # if more than 1 ms passed, branch to miss
```

Hardware implementation of these timing pseudo-instructions is relatively straightforward. Many possible variations exist, but the general idea is using the result of a timer comparison to either stall, branch, or interrupt. A timer does not need to count clock cycles; counting at an adjustable rate expressed in nanoseconds enables a timer to be synchronized in a distributed system. Software support is less straightforward. A timer has finite width, so overflow must be considered. If a block continues execution after a timer interrupt, the interrupt handling routine must save and restore state. If a block is interrupted and not allowed to finish execution, however, the program must prevent inconsistent state and memory leaks.

Timing instructions add temporal semantics to programs but do not fully specify behavior. The accuracy and precision of timing behavior still depends on the underlying hardware design—clock cycle granularity and pipelines prevent immediate reaction and execution. Also, the fourth case of specifying maximum execution time is not just a run-time operation. It requires modification of the traditional toolchain flow to use static analysis on code blocks so that the compiler can reject programs with segments that exceed the specified maximum execution times.

2.4 Timing-Predictable Hardware

Background

For a sequence of instructions in a program path, not every instruction is dependent on the result of the previous instruction. Therefore, a processor can improve throughput by executing instructions in parallel, exploiting *instruction-level parallelism* (ILP). Pipelining, where the processor executes each instruction in a series of stages and starts the next instruction before the current instruction completes, is a common technique to improve throughput. Instructions partially overlap, often *forwarding* or *bypassing* results from later pipeline stages, and dependencies require a reduction in overlap by stalling the pipeline.

The main limitations for achieving high ILP are branching instructions, memory latencies, and a limited number of functional units. On a branch instruction, both the decision to branch and the next instruction's memory location are not known until several cycles into the pipeline. *Branch prediction* speculatively executes instructions to avoid stalling the pipeline for every branch. The simplest form is *predict not-taken*, where execution assumes the branch is not taken and flushes the pipeline if it is. For deeper pipelines, the cycle cost of flushing the pipeline is high enough to motivate better prediction accuracy. In *static branch prediction*, the compiler predicts the direction of each branch instruction. In *dynamic branch prediction*, the processor uses past history of different branches to predict each branch decision.

The wide gap between processor speed and main memory access time causes a widespread use of caching. *Caches* store copies of data to handle some requests faster but cannot store all data due to limited size. *Replacement policies* in hardware dictate the contents of a cache

and attempt to exploit both temporal locality, if requested data was recently requested, and spatial locality, if requested data is near a previous request. Cache *associativity* determines how many possible locations in the cache can contain a location in main memory; higher associativity reduces multiple main memory locations conflicting at the same cache location but requires more hardware. On a memory request from the processor, a *cache hit* occurs when the data is in the cache, otherwise a *cache miss*. Although a cache hit returns the requested data within a few cycles, a cache miss requires retrieving data from higher-level caches or main memory, potentially taking hundreds of cycles and stalling processor execution. A *scratchpad memory (SPM)* is similar to a cache memory, but contents are managed explicitly by software and not by hardware.

Instructions do not always need to execute in order for correct behavior. A compiler can re-order instructions by using known dependencies and latencies. Instructions with variable dependencies and latencies, such as memory operations on dynamically resolved addresses, are difficult to optimize statically with a compiler. On a cache miss or other high latency instruction, *dynamic execution* (*out-of-order execution*) processors hide some latency by finding later independent instructions to execute, improving ILP.

In a *single-issue* processor, at most one instruction completes every cycle, even if more parallelism exists. In *multi-issue* processors, the pipeline fetches multiple instructions every cycle and has multiple execution pipelines, also called *ways*. Although still restricted by dependencies between instructions and the number of execution pipelines, multiple instructions can complete every cycle. A *Very Long Instruction Word (VLIW)* processor fetches a long instruction that performs multiple operations every cycle and relies on *static scheduling* by the compiler to resolve dependencies between operations by inserting NOPs. In a *dynamic superscalar* processor, dependencies are resolved by hardware logic instead of the compiler to improve throughput for variable dependencies and latencies. Dependencies between instructions and execution latencies limit available ILP, and at some point the hardware costs of further improving ILP are prohibitive.

An application often has multiple mostly independent *processes*, which are each an instance of a program. *Thread level parallelism* is the technique of executing multiple processes in parallel to improve throughput, such as on different processors in a multiprocessor system or different cores in a multicore. A *multithreaded* processor supports multiple processes using *hardware threads*, each with its own program counter and set of registers, and shares the pipeline to execute multiple processes in parallel. In a single-threaded processor, stalls waste processor cycles, but a multithreaded processor can fill cycles with independent instructions from another hardware thread.

A *hardware thread scheduler* decides which hardware thread to issue into the pipeline each cycle. In *fine-grained multithreading*, the fetched hardware thread changes every cycle, whereas in *coarse-grained multithreading*, the fetched hardware thread only changes during long latency instructions or periodically. Multi-issue processors can use *simultaneous multithreading (SMT)* to issue multiple instructions from either a single or multiple hardware threads each cycle.

Design for WCET

After investigating and implementing WCET analysis techniques on different processor microarchitectures, researchers have identified processor properties that aid WCET analysis. The general consensus is timing behavior dependence on execution history and interference between microarchitectural components should be minimized when possible. Heckmann et al. [21] implemented WCET analysis techniques for multiple processors of varying complexity and proposed some guidelines for future processors used in hard real-time systems: (1) separate data and instruction caches to remove any dependencies between instruction prefetching and memory instructions; (2) use cache replacement strategies, such as LRU, that allow state information recovery over time after unknown starting state; (3) use static branch prediction to remove prediction dependence on execution history; (4) limit out-of-order execution to reduce possible instruction interleaving; and (5) do not use hardware to improve special cases if it is difficult to model.

Berg et al. [24] also proposed some design principles—emphasizing microarchitectural techniques that support *recoverability* of state information from an unknown starting state—demonstrated by a 32-bit, 5-stage single-issue pipeline design with forwarding paths, static branch prediction, and physically addressed LRU caches if needed. In addition to similar hardware suggestions, Thiele and Wilhelm [76] proposed approaches for increasing predictability in the software development process on both shared and distributed resources.

With the increasing use of multicore processors in embedded systems, both Wilhelm et al. [77] and Cullmann et al. [78] identified interference caused by shared resources as a major challenge for WCET analysis that must be avoided when possible. Time and space partitioning (reserved time segments and private regions) of a shared resource reduce interference, but sometimes bounds on interference must be sacrificed for performance.

2.4.1 Timing-Predictable Processors

The diversity of real-time embedded applications and associated requirements has resulted in a variety of timing predictable processor architectures, often using different definitions of predictability [16]. One of the main concepts for many predictable processors is using the worst-case execution time *bound* as the metric for performance, not the actual worst-case execution time (often unobtainable) or average-case execution time, because the bound dictates resource provisioning. For example, schedulabilty analysis for real-time task sets, as described in Section 2.1, relies on WCET bounds. Unfortunately, the WCET bound is not an inherent property of the processor and depends on the current state-of-the-art in WCET analysis and its implementation. For applications that use the WCET bound metric, such as safety-critical hard real-time systems, a higher average-case execution time is acceptable for a lower WCET bound.

Some applications, such as mixed-criticality systems, require low WCET bounds for some tasks but also high average-case performance for others. Processors designed solely for WCET bounds or high average-case performance do not provide this flexibility. This sec-

tion will discuss the applicability of existing timing predictable processors to isolated-bound task sets—where hardware isolation ability and high overall throughput is more important than single-threaded throughput—even though most were not designed particularity for this application area. Most proposed processors were only evaluated using simulation models with metrics of bounded or measured worst case execution cycles; the additional area or clock period costs could not be evaluated without a hardware implementation. Timing predictable processors can be categorized as single or multithreaded, and furthermore as single or multi-issue.

Single-threaded, Single-issue

Single-threaded, single-issue predictable processors are the simplest in hardware complexity but do not always provide adequate single-threaded performance. Lower-end commercial processors—with 3-5 pipeline stages, no branch prediction, in-order execution, pipeline forwarding, and scratchpad memories or simple caches—are often sufficient for some real-time embedded applications and well supported by WCET analysis tools. Additional support for single-path programming and fast interrupt responses, as provided by the SPEAR processor [30], can provide faster response times with low jitter.

Single-threaded, Multi-issue

Single-threaded, multi-issue predictable processors improve single-threaded throughput by executing multiple operations in parallel every cycle, but they also increase hardware complexity, sometimes substantially. Dynamically scheduled superscalar processors execute instructions out-of-order to prevent blocking from a single dependency, but many possible instruction interleavings complicates WCET analysis [60]. To reduce dependence on execution history, Rochange and Sainrat [79] proposed regulating instruction issue to force independent execution timing of basic blocks, using a modified version of an instruction prescheduling buffer. Only allowing ILP gains within a basic block reduces performance but still provides a higher single-threaded throughput than a scalar in-order pipeline. Whatham and Audsley [80] extended this approach to obtain ILP gains outside of each basic block. The main idea is enforcing independence only between *virtual traces*, which are sequences of basic blocks on a selected program path; the preferred path is selected to minimize WCET, similar to an algorithm used by Bodin and Puaut [81] for minimizing WCET with static branch prediction. Both evaluations ignored cache effects, which WCET analysis would need to model if used.

VLIW processors rely on a customized compiler to schedule code instead of hardware logic, improving ILP with less hardware complexity. Static decisions are visible to WCET analysis but cannot handle dynamic instruction latencies, such as branching or stalling on a cache miss. Yan and Zhang [82] analyzed and proposed improvements for VLIW architecture predictability, assuming a perfect data cache and either a perfect or direct-mapped instruction cache. To reduce branching and optimize across basic blocks, *if conversion* converts

control dependencies to data dependencies by replacing branches with *predicated* instructions. They use compiler techniques to mitigate some performance degradation of single-path programming—which requires executing instructions from both branch paths instead of just the taken path.

The PATMOS processor by Schoeberl et al. [47], [83] is also similar to VLIW, and is one of the few predictable processors that includes caches for both instructions and data. For predictability, the method cache stores entire functions, so a cache misses only occurs on function call or return, and data is separated into a stack cache and data cache. Any load or store instruction that does not have a statically predictable address bypasses the caches. To support single-path programming and reduce branching, all instructions are fully predicated.

Multithreaded, Single-issue

Multithreaded processors, unlike single-threaded processors, can isolate tasks without using software scheduling by deploying them on separate hardware threads, as discussed in Chapter 1. Additional hardware stores state, such as the program counter and general-purpose registers, for each hardware thread. Existing approaches isolate either all or none of these hardware threads, depending on hardware thread scheduling decisions. If all hardware threads are isolated, the overall throughput of the processor is lower whenever a hardware thread is not *active*, which occurs when program execution is waiting for something before proceeding.

Multithreading was initially used for improving overall throughput—interleaving instructions from independent programs increases pipeline spacing between dependencies and hides latency. The CDC 6600 peripheral processors statically interleaved 10 hardware threads and used them as virtual processors for I/O tasks [41]. Sun Microsystems' Niagara processor was designed to exploit thread level parallelism in server applications to hide memory latencies by interleaving only active hardware threads [42].

More recently, multithreading has been applied to real-time systems. Previous PRET machines [4], such as the SPARC-based processor by Lickly et al. [74] and the ARM-based PTARM processor by Liu et al. [33], [43], use fine-grained multithreading with fixed round-robin scheduling over exactly four hardware threads, even with non-active threads. The hardware-based isolation is beneficial for hard real-time systems but inflexible for achieving high overall efficiency: cycles are wasted if any hardware thread is not active and a single hardware thread only executes once every four cycles, regardless of throughput requirements.

XMOS is a commercial fine-grained multithreaded processor. It uses active round-robin scheduling for up to 8 hardware threads, but can also be configured to use fixed-round robin scheduling. XMOS cannot interleave fewer than 4 hardware threads, so it will efficiently use every cycle when between 4 and 8 hardware threads are active, with non-active threads ignored. This scheduling approach reduces isolation because the logical execution frequency of a hardware thread depends on the number of active hardware threads, which likely varies over time.

Multithreaded, Multi-issue

In SMT processors, high overall throughput is obtained by using multiple execution pipelines to complete multiple operations every cycle and multiple hardware threads to effectively provide more instructions with fewer dependencies. This type of processor has high overall throughput, but WCET analysis must handle the high hardware complexity, dynamic behavior, and susceptibility to timing anomalies [62].

El-Haj-Mahmoud et al. [84] proposed modifying an in-order issue, 4-way superscalar processor to support isolated scheduling in both time and space of multiple hardware threads; the scheduler allocates each hardware thread combinations of ways during certain segments of time using an offline bin-packing algorithm. For predictability, each hardware thread uses predict not-taken branching and instruction and data SPMs. To prevent the high hardware cost of an instruction SPM port per hardware thread, a static schedule selects one hardware thread to fetch from the instruction SPM port to its instruction buffer every cycle.

Each execution pipeline does not support all operations—it would be prohibitively expensive to have 4 floating-points units, 4 read-write ports to data memory, and 4 integer multiplier and dividers—so the scheduler also interleaves priority to each execution pipeline between hardware threads. The WCET bound for a task is increased by instructions waiting for interleaved priority on the instruction SPM port on a branch or jump or any of the execution pipelines. Execution pipelines are also idle when a scheduled hardware thread finishes early or is waiting for priority on another execution pipeline, another example of reduced efficiency when all hardware threads are isolated.

Another scheduling approach is prioritizing only one hardware thread for isolated behavior and sharing the unused cycles between the remaining hardware threads. Hardware isolation for multiple tasks requires multiple processors, but non-isolated tasks efficiently share the multiple execution units. The CarCore SMT processor by Mische et al. [32] in the MERASA project uses this approach. They also use in-order issue to reduce hardware complexity and improve predictability. There are two execution pipelines: one for arithmetic operations and the other for address calculation and memory access to an off-chip or scratchpad memory. Fetching instructions to buffers for each hardware thread and in-order instruction issue to the two pipelines occurs in priority order of the hardware threads. To prevent a memory operation of a lower priority hardware thread from blocking the highest priority thread, a load instruction is split into a request microinstruction to an address buffer and a writeback microinstruction to store the result.

Comparison to FlexPRET

The processor presented in this dissertation, FlexPRET, is multithreaded and single-issue. For an isolation-bound task set—where hardware isolation ability and high overall throughput is more important than single-threaded throughput—a multithreaded and multi-issue processor is more difficult to model and analyze for WCET bounds and not as efficient with hardware area. The additional hardware area required to implement more complex control

logic, additional bypass paths, and multiple functional units could instead be used for a second processor, as the ILP gains are unlikely to scale linearly with hardware cost and are not needed to meet any task throughput requirements. For this type of task set, a single-threaded processor cannot provide any hardware-based isolation. As discussed in Chapter 1, software-based isolation requires high confidence in RTOS implementation correctness, both functionality and timing, and consideration of preemption effects.

Other Approaches

A single processor executing assembly code generated by a C language compiler is not the only way to deploy real-time applications. The Java language has some advantages over C: well-defined syntax and semantics simply writing portable code that includes concurrency, preventing many common programming errors. Some disadvantages are additional memory overhead and performing predictable garbage collection. The JOP (Java Optimized Processor) by Schoeberl [31] is a predictable processor designed to execute Java bytecode for embedded real-time systems. It uses a *method cache* for instructions that caches entire methods, so misses can only occur on method calls and returns. The fetched Java bytecode is converted to microcode that executes on a 3 stage, stack-based pipeline. Operations are performed on a *stack cache* with the top two elements in registers and the rest in on-chip memory, with push and pop moving data between them. The execution of Java bytecode is exactly predictable without dependencies because of these caches and the short, predictable pipeline.

2.4.2 Timing-Predictable Memory Hierarchy

A common memory hierarchy is several levels of caches on chip connected to a DRAM main memory, with each level cache progressively larger but slower. For small, embedded processors at lower clock speeds, there may only be one level of cache with code and some data stored in flash or EEPROMs. In a *Harvard memory architecture*, there are separate storage components and associated connections for instruction and data. Separate instruction and data caches support both fetching an instruction and performing a memory operation in the same cycle, instead of stalling fetch during a memory operation, and reduce interference. Caches affect the timing behavior of memory instructions because cache contents, which depend on execution history, determine whether an access is a hit or miss. Instead of analyzing cache behavior during WCET analysis, the memory hierarchy can use scratchpad memories; all accesses always succeed, but the programmer or compiler must explicitly manage SPM contents. FlexPRET uses scratchpad memories but could be adapted to use other predictable memory hierarchies.

Cache Memory

Even though contents depend on execution history, caches are still analyzable for WCET. Many techniques focus on instruction caches, perhaps because WCET analysis is often easier for instructions than data: the program counter is the memory address for the instruction cache, but the address for the data cache can be data-dependent and dynamic. A basic analysis concept used by Ferdinand and Wilhelm [85] is finding *must* and *may* cache information, where the cache is guaranteed to contain must-cache blocks and could contain may-cache blocks, at different program locations. This information is conservatively merged at locations where control flows meet. For predictability, fast recovery of cache information is important, as merging often reduces known information. The associativity, replacement policy and write policy all influence the predictability of a cache, as observed by Heckmann et al. [21]. Reineke et al. [22] quantitatively investigated the Least Recently Used (LRU), First-In First-Out (FIFO), Most Recently Used (MRU), and Pseudo-LRU (PLRU) replacement policies and demonstrated the LRU is most predictable for the selected metric.

More control over cache usage and operation can also improve predictability. *Cache partitioning* reduces or removes preemption-related interference between tasks by privatizing sections of the cache; each task has an effectively smaller cache but its private contents will not change during preemptions. *Cache locking* uses hardware support to freeze the contents of certain lines, with cache misses retrieving data from main memory instead of replacing the cache line. Puaut [28] proposed several algorithms to improve WCET by selecting instruction cache contents in different regions of operation, then load and lock the correct lines during run-time. Whitham and Audsley [29] also proposed a technique to remove preemption-related interference between tasks, but the technique allows a task to use the entire contents of the local memory instead of just a private section. These techniques can reduce average case-performance, but they can also simplify WCET analysis or improve WCET bounds. Schoeberl [37] proposed using separate caches to remove interference between different types of memory operations, with caches for instruction methods, the stack, static data, heap data, and constants.

Scratchpad Memory

Scratchpad memories were originally proposed to reduce latencies [34] or energy usage [35] by statically placing frequently used instruction and data blocks in the scratchpad memory with other accesses going to main memory. Most techniques optimize for average-case execution time or energy consumption and not for WCET. Scratchpad allocation can also be static or dynamic, depending on if SPM contents change during run-time. For large applications, which could consist of many small tasks, static approaches are limited by the size of the scratchpad memory and suffer many slow main memory accesses. Dynamic approaches change the contents during run-time, but the algorithm must consider SPM modification delays. As scratchpad memory management has many similarities to cache locking, with memory addressing the primary difference, Puaut and Pais [86] proposed a dynamic algo-

rithm to manage code for both types of memory and observed similar WCET estimates. Kim et al. [36] presented two WCET-aware dynamic SPM code management techniques that reduce WCET estimates, one optimal and the other a heuristic for better scalability.

DRAM Memory

Unless a program fits entirely in the local memory, WCET analysis for both caches and SPMs needs bounds on main memory latency, as this determines the delay of moving data into the cache or SPM. Load and store operations to a DRAM array require a series of DRAM commands over the command, address, and data interface buses, with minimum delays between different commands. To hide the delay of accessing a DRAM array, there are multiple parallel banks of DRAM arrays that can each perform operations in parallel but contend for bus access. The DRAM controller converts the high-level load and store requests for blocks of data to sequences of commands to the banks of DRAM arrays.

For each memory request, the DRAM controller has a policy for mapping an address and data length to bit locations in DRAM arrays, generates the sequence of commands to perform the operation, and manages scheduling between multiple memory requests. If a contiguous segment of data is interleaved across banks, the parallelism provides low latency for a single request, but interference from other requests can increase the latency. For less interference and more predictability, each requester could have private banks, as proposed by Reineke et al. [87]. The latency and throughput for each requester is isolated and fixed, but it cannot be shared.

Without discussing details of DRAM operation, an *open-page* policy exploits temporal locality by decreasing latency to nearby memory locations but increasing latency to other locations. The increase in worst-case latency motivates many predictable DRAM controllers to use a *close-page* policy for more consistent latencies. Commercial DRAM controllers often re-order requests to improve throughput, but this can increase the worst-case latency bound [88]. Several DRAM controllers have been designed for mixed-criticality systems. Paolieri et al. [89] propose a memory controller that prioritizes critical requests and shares remaining resources between non-critical requests, but bank interleaving causes interference, and the close-page policy limits throughput. Any DRAM controller used for FlexPRET should provide tightly bounded worst-case latency for HRTT requests but could have looser bounds for average-case latency for SRTT requests. The DRAM controller proposed by Kim et al. [90] meets these requirements by prioritizing critical requests to private banks while using interleaved banks with an open-page policy for high throughput to non-critical requesters.

Chapter 3

FlexPRET Processor Design

Designing and verifying cyber-physical or real-time embedded systems with safety or precise timing constraints is challenging, particularly while maintaining efficient resource usage and confidence in correctness. Programmers and software abstractions use assumptions about the timing behavior of the underlying processor that are difficult to guarantee. Instead of restricting programming methods or performing complex execution time analyses that cannot scale, the underlying processor could better support the requirements of these systems. This chapter discusses microarchitectural techniques for such a processor and describes a specific processor design, called FlexPRET, deployable as a soft-core on a field-programmable gate array (FPGA). The processor design has some improvements from the original version [91]. Some techniques are inspired by previous work, such as PRET machines [33], [74] and XMOS processors [44], while others are unique to this processor design.

Different microarchitectures are most suitable for different classes of applications. Our design decisions target isolation-bound task sets, which consist of many concurrent tasks that prioritize timing correctness guarantees through isolated and predictable behavior over single task latency and throughput. For isolated timing behavior, known and coordinated dependencies should be the only source of interference between tasks. Task sets in real-time systems often possess these properties; if not, a high-performance processor that uses additional hardware to exploit instruction-level parallelism may be more suitable, such as predictable superscalar [32] or VLIW [47] processors.

The highest-level design decision is the general complexity of the processor, as an 8-bit non-pipelined processor is simpler and cheaper than a 64-bit superscalar processor with over a dozen pipeline stages but has lower performance. In the real-time embedded space, a common trade-off between cost and performance is a 32-bit single-threaded processor operating at a few hundred MHz with 3-6 pipeline stages, such as an ARM Cortex-M3 or Cortex-M4. At this design point, instruction and data caches are optional, and flash memory is fast enough for code storage with minimal prediction and buffering. FlexPRET is at this level of complexity, although its architectural techniques are adaptable to more complex processor designs.

FlexPRET is a 32-bit, 5-stage, fine-grained multithreaded processor implementing the base RISC-V ISA [92]. RISC-V is a completely open (not proprietary) ISA, originally designed to support computer architecture research but gaining industry attention. The base ISA is small but usable, enabling an efficient minimal hardware implementation. Optional extensions add target-specific functionality, such as floating-point operations. For smaller code and hardware size, FlexPRET uses the 32-bit version without any extensions (RV32I), and a future implementation could support a more compact version of RISC-V with variable-length instructions [93].

The standard approach of executing a task set using a real-time operating system (RTOS) on a single-threaded processor presents challenges for worst-case execution time (WCET) analysis: hardware prediction mechanisms, such as branch prediction and caching, introduce variable latencies that are difficult to predict; and interrupts, which perform preemptive task scheduling and often I/O, interfere with these latencies. Removal of hardware prediction mechanisms and interrupts in a single-threaded processor reduces performance and restricts the programming model but has less effect on a fine-grained multithreaded processor. Instead of the pipeline stalling (waiting) during a branch or jump instruction, the pipeline executes instructions from other hardware threads. Particular hardware threads handle each interrupt and isolate their effects. An additional benefit with multithreading is a context switch between tasks on different hardware threads occurs every clock cycle, enabling low-latency responses to multiple events; a single-threaded processor requires multiple clock cycles to context switch with software saving and restoring program state.

By using fine-grained multithreading, FlexPRET removes dynamic branch prediction and hides branch latency with hardware thread concurrency, and it isolates interrupts to particular hardware threads (Section 3.1). Temporal isolation of each hardware thread depends on how it is scheduled. The most novel and compelling technique used in FlexPRET is its flexible, software-controlled hardware thread scheduler (Section 3.2). By classifying hardware threads as either hard real-time threads (HRTTs) or soft real-time threads (SRTTs), the processor can trade-off predictability, isolation, and throughput for each thread. Extensions to the ISA, called timing instructions [43], [45], [46], allow programs to express real-time semantics and are also leveraged by the hardware thread scheduler to improve overall throughput (Section 3.3).

The most contentious design decisions involve the memory hierarchy (Section 3.4). HRTTs use scratchpad memories (SPMs) instead of caches to provide constant latency memory operations, but the burden is on software to explicitly manage SPM contents. We assume programs fit entirely within the SPMs or use SPM management [36]; applications with more demanding memory requirements should use caches and accept the variable latency. FlexPRET's hardware thread scheduling does not require SPMs, so a future variation could use caching if supported by WCET analysis.

Unless it interacts externally, an application is useless. The pipeline communicates with external hardware peripherals using a simple peripheral bus that does not affect the isolation or predictability of hardware threads (Section 3.5). A gateway component can interface with more complex bus protocols. The combination of predictable instruction latencies and timing

instructions supports controlling the time of an instruction's execution within several clock cycles, so FlexPRET supports some simple mechanisms for digital I/O operations on pins (Section 3.6). More precise mechanisms with less software overhead are possible with external hardware peripherals.

High-level comparisons with other processors only require the cycle-level design details in this chapter, but a more rigorous area, timing, and power evaluation requires implementation in hardware. The relative cost in program performance and resource utilization of these microarchitectural techniques is evaluated in Chapter 4 by comparing against other similar implementations through parameterization.

3.1 Datapath and Control Unit

The hardware thread scheduler decides which hardware thread to fetch each cycle. The pipeline supports an arbitrary interleaving of hardware threads and enforces the scheduling decisions; once an instruction is fetched, its execution through the pipeline is isolated from the behavior of other hardware threads. This section briefly summarizes each stage and the control logic used to execute different types of instructions. The design targets a soft-core implementation on FPGA, with many decisions influenced by exploratory synthesis and place-and-route results. For deployment as an application-specific integrated circuit (ASIC), some modifications to the design could improve area, critical path, and energy usage.

An overview of the datapath is shown in Figure 3.1. The tall rectangles with small triangles represent pipeline registers that store signals between pipeline stages, and the signal names at start and end points are used instead of long lines for unregistered connections between pipeline stages (e.g. *Addr* for ALU result connection to next PC multiplexer). Control signals are also not shown.

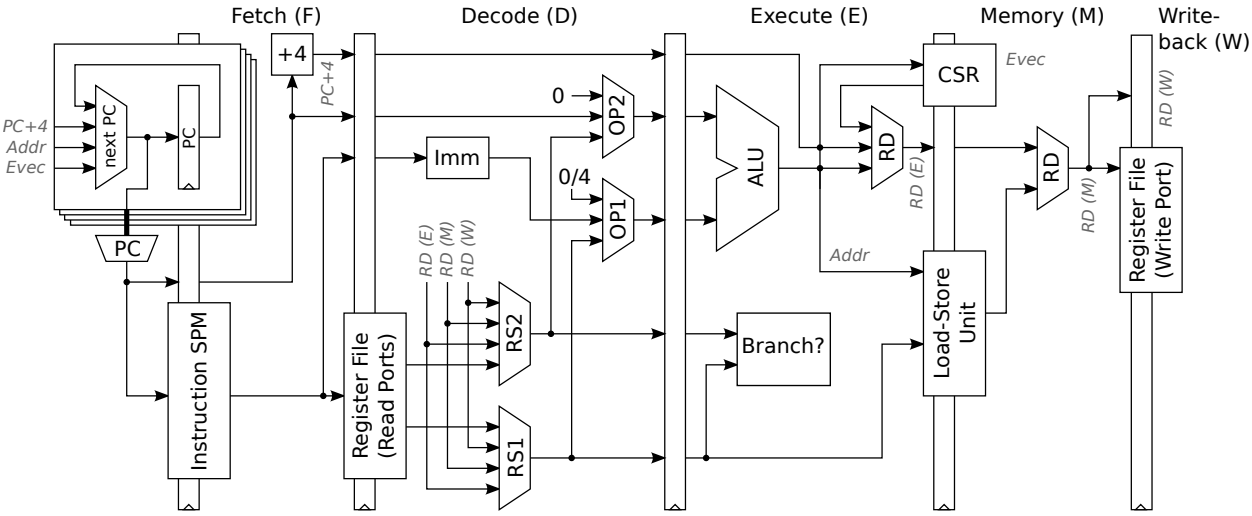


Figure 3.1: A high-level diagram of FlexPRET's datapath.

The design assumes all memories are sequential with the processor clock: the inputs are provided in one clock cycle and the outputs are available in the next clock cycle. In the selected FPGA, inputs only need to arrive just before the rising clock edge, and most of the access latency occurs in the next clock cycle. Consequently, as seen in Figure 3.1, the datapath provides inputs to the scratchpad memories and register file at the end of the previous stage.

For isolation between hardware threads, every fetched instruction goes directly through the pipeline, but for correct program execution, not every instruction will complete. An instruction does not modify processor state until the end of the execute stage, which is the *commit point*. Before this point, the control unit can kill any instruction, but after this point, all instructions must complete. The control unit kills instructions to handle data dependencies, control flow, interrupts, and exceptions and will only modify the necessary control signals.

3.1.1 Pipeline Stages

The *fetch* stage retrieves the instruction stored in the *instruction scratchpad memory (I-SPM)* at the program counter (a memory address) of the scheduled hardware thread and produces a copy of the program counter incremented by 4 (the address of the next sequential instruction in memory). The source operand register addresses are at fixed bit locations in the instruction, so the datapath directly provides requests to the two read ports of the register file without any decoding required. In contrast to a single-threaded processor, the control unit may need to update the program counter of multiple hardware threads each cycle, with new values coming from the execute stage with a branch or jump, the memory stage with an exception handler address, or the fetch stage itself with the incremented program counter. The possibly updated program counter of the next scheduled hardware thread is connected to the read port of the I-SPM for instruction output in the next fetch cycle.

In the *decode* stage, the control unit determines the control signals required for the datapath to properly execute the instruction. Registers store control signals needed for later pipeline stages and may be modified in later cycles depending on program behavior, such as preventing a write operation from a killed instruction. The stage also prepares operands for components in the next stage: source register values from the register file, forwarded values, immediate values, the program counter, or constants. Logic decodes the immediate value, if present, from the instruction.

Because of pipelining, the correct source register value can exist in a later stage but not yet be written into the register file. The control unit must preserve data dependencies by using the correct value; using the incorrect value is known as a *data hazard*. Bypassing (operand forwarding) adds paths to enable using available data without always stalling a hardware thread's execution until the value is written into the register file. The control unit uses information about instructions in the pipeline to select data from the execute, memory, or writeback stage instead of the register file as the source register value. We decided the throughput improvement for frequently scheduled hardware threads is worth

the added complexity of bypassing; otherwise at least four hardware threads need to be constantly executing to fully utilize the pipeline.

The *execute* stage is the most complex in this pipeline. The arithmetic logic unit (ALU) computes the specified operation on the operands, either producing the destination register data, input to the control status register (CSR) unit, or an address used as a program counter or memory location. The ALU can only perform one operation, but the branch instruction must both compare source register values to evaluate the condition and compute an address for the destination, so a dedicated unit is required for one of these operations. We decided to use a dedicated branch condition checker because the result is used by one of the longer paths in the processor and ALU complexity does not increase.

The *load-store unit* handles memory operations to the I-SPM read-write port, the *data scratchpad memory (D-SPM)* read-write port, or the peripheral bus, depending on the address provided. Data manipulation enables subword memory operations for the D-SPM but not the I-SPM or peripheral bus. As with other memories, the request occurs in the execute stage, and the response occurs in the next stage.

The execute stage also contains the *control and status register (CSR) unit* that handles interrupt handling (Section 3.1.5), thread scheduling configuration (Section 3.2), and timing instructions (Section 3.3). A multiplexer selects destination register data from either the ALU, a CSR, or program counter. The commit point of the processor, where instructions must always complete, is at the end of the execute stage because an instruction could modify either a CSR or memory location, which cannot easily be undone. The control unit can kill instructions in the fetch, decode, or execute stage if required, preventing any of these modifications.

In the *memory* stage, the load-store unit returns data for any load instruction, including subword operations from the D-SPM. The destination register data, either from the load-store unit or the execute stage, is connected to the write port of the register file. The *writeback* stage registers provide an accessible copy of data that is in the process of being stored in the register file as a bypass path.

3.1.2 Computational Instructions

Many instructions perform a computation on one source register value and either a second source register value or a decoded immediate value, and then store the result in a destination register. Computations include arithmetic, shifting, bitwise logic, and comparisons. These instructions increment the program counter by four and produce a destination register value in the execute stage. If the subsequent instruction is scheduled within three or fewer cycles and uses the preceding instruction's destination register value, the control unit will enable the correct bypass path. The only difference from bypassing in a single-threaded processor is the thread IDs are also compared to prevent data usage from the wrong hardware thread.

The following table lists RISC-V instructions of this type and their latencies under different operating conditions. The instruction format column shows the required operands: `rd` for a destination register, `rs1` for a first source register, `rs2` for a second source register, and

`imm` for an immediate value. The number of *thread cycles* is how many cycles a hardware thread must be scheduled before the next instruction executes without any dependencies with the current instruction. If the *scheduling frequency* f of a hardware thread is $1/X$, it is scheduled exactly once every X cycles. For example, $f = 1/3$ when the scheduler interleaves three hardware threads in round-robin order. The RISC-V specification [92] provides more detail on instruction syntax and semantics than is provided by this chapter.

Instruction Format	Instruction	Thread cycles at $f =$			
		1/1	1/2	1/3	1/4
`INST rd, imm`	`LUI, AUIPC`	1	1	1	1
`INST rd, rs1, imm`	`ADDI, SLTI, SLTIU, XORI, ORI, ANDI`	1	1	1	1
`INST rd, rs1, imm`	`SLLI, SRLI, SRAI`	1	1	1	1
`INST rd, rs1, rs2`	`ADD, SUB, SLL, SLT, SLTU, XOR, SRL, SRA, OR, AND`	1	1	1	1

Table 3.1: Thread cycle count for computational instructions at different hardware thread scheduling frequencies.

3.1.3 Control Instructions

Jump and link instructions unconditionally change control flow by modifying the program counter to a computed address, which is not available until calculated by the ALU in the execute stage. Branch instructions conditionally do the same, based on a comparison that also occurs in the execute stage. During the execute stage, the hardware thread's program counter is modified to the computed address, which can then be fetched in the next cycle.

The default operation for a hardware thread is to speculatively fetch the next contiguous instruction in memory whenever scheduled, which is correct behavior unless there is a branch taken, jump, or exception. When the new program counter is available in the execute stage, depending on hardware thread scheduling, incorrect instructions from the same hardware thread can be in the fetch or decode stage. The control unit then kills these instructions by modifying control signals to prevent modification of any processor state (such as I-SPM, D-SPM, register file, PC, and CSRs).

Consequently, a branch taken or jump instruction takes 1-3 thread cycles to complete, depending on if the hardware thread was scheduled for the fetch or decode stage. A not-taken branch takes 1 thread cycle because any instructions in the pipeline from the same thread are correct. Ignoring system and timing instructions, branch instructions are the only instructions without a constant cycle latency. However, latency is only dependent on program flow, which is already part of program analysis. Conditional branches could always wait, but there is a performance benefit of predicting branches will not be taken.

Instruction Format	Instruction	Thread cycles at $f =$			
		1/1	1/2	1/3	1/4
`INST rd, imm`	`JAL`	3	2	1	1
`INST rd, rs1, imm`	`JALR`	3	2	1	1
`INST rs1, rs2, imm`	`BEQ, BNE, BLT, BGE, BLTU, BGEU`	3 or 1	2 or 1	1	1

Table 3.2: Thread cycle count for control instructions at different hardware thread scheduling frequencies.

3.1.4 Data Transfer Instructions

Store instructions are similar to computational instructions but use the ALU for address calculation and store data into the D-SPM, I-SPM, or peripheral bus instead of the register file. All stores start at the end of the execute stage and complete in the memory stage. Both the D-SPM and I-SPM have a second port for scratchpad memory management without stalling the pipeline, and this second port on the I-SPM is shared with the load-store unit. For subword stores to non-word-aligned addresses in the D-SPM, the load-store unit uses data manipulation and byte write masks. Unaligned memory accesses, such as a word store to a non-word-aligned address, are either disallowed or cause a trap.

Load instructions use the ALU for address calculation and use the memory stage to load data from the D-SPM, I-SPM, or peripheral bus. For subword loads from the D-SPM, the load-store unit shifts and zero or sign extends the data. Because the destination register data is not available until the memory stage instead of the execute stage, a load-use data hazard can occur that is not preventable using bypass paths. This situation only occurs when the same thread is scheduled for multiple consecutive cycles, such as in single-threaded mode, and the processor is purposely not optimized for this operating mode. Therefore, all load instructions are forced to take two thread cycles in single-threaded mode instead of adding additional control logic.

Instruction Format	Instruction	Thread cycles at $f =$			
		1/1	1/2	1/3	1/4
`INST rs1, rs2, imm`	`SB, SH, SW`	1	1	1	1
`INST rd, rs1, imm`	`LB, LH, LW, LBU, LHU`	2	1	1	1

Table 3.3: Thread cycle count for data transfer instructions at different hardware thread scheduling frequencies.

3.1.5 System Instructions

There are two main types of system instructions: CSR modification instructions and instructions for changing between the two privilege levels, *user mode* and *privileged mode*. System instructions control thread scheduling, timing instructions, I/O mechanisms, and handling of exceptions, interrupts, and traps. To reduce hardware overhead and complexity, only a minimal subset of RISC-V privileged specification [94] is supported.

The CSR unit contains registers with various properties: private or shared between hardware threads; read-write, read-only, or unaccessible from user mode; and read-write or read-only from privileged mode. All CSRs use an atomic read-modify-write operation because certain CSRs are modified by other hardware threads or external events. To obey the commit point at the end of the execute stage, the read and modify occurs during the execute stage and the write at the clock edge.

CSR instructions support reading or writing all bits and setting or clearing specific bits, although each CSR may not allow all these operations (e.g. read-only). The `CSRRW` and `CSRRWI` instructions atomically read the current value and write a new value, specified by either a register or immediate, respectively. The `CSRRS` and `CSRRSI` instructions also atomically read the current value, but the register or immediate value specifies which bits are atomically set high, with other bits unaffected. The `CSRRC` and `CSRRCI` instructions are similar, but atomically clear low instead of setting high.

Instruction Format	Instruction	Thread cycles at $f =$			
		1/1	1/2	1/3	1/4
`INST rd, csr, rs1`	`CSRRW, CSRRS, CSRRC`	1	1	1	1
`INST rd, csr, imm`	`CSRRWI, CSRRSI, CSRRCI`	1	1	1	1

Table 3.4: Thread cycle count for CSR modification instructions at different hardware thread scheduling frequencies.

Handling traps, exceptions, and interrupts complicate the design of the control unit and datapath. We use RISC-V terminology, where an *exception* is an unusual condition at runtime, a *trap* is a synchronous transfer of control, and an *interrupt* is an asynchronous transfer of control. In FlexPRET, both exceptions and interrupts are handled by the trap mechanism. When an instruction causes an exception, all previous but no following instructions will complete, and the faulting instruction can be restarted after exception handling, referred to as *precise exceptions*. Without precise exceptions, software must try (if even possible) to recover state and restart execution at the correct point.

The trap mechanism is triggered by the `SYSCALL` instruction, an exception, or an interrupt. To transfer control to the trap handler routine that runs in privileged mode, the program counter is set to the routine's address, stored in the *evec* CSR for that hardware thread. By storing the address of the first uncompleted instruction in the hardware thread's

epc CSR, software can handle an interrupt or diagnose an exception and correctly restart execution. For further information, the cause of the trap is also classified and stored in the hardware thread's *cause* CSR. Some of the optionally enabled causes are illegal or system instructions, misaligned memory accesses, and external or timing interrupts.

To support precise exceptions, the trap mechanism must be carefully designed. Exceptions cannot always be handled immediately, as previous instructions may not have completed and could also cause an exception. Out-of-order exception handling is prevented to avoid implementation-specific program behavior. In FlexPRET, all exceptions do not cause a trap until the execute stage, when the corresponding instruction is the next to complete, and must be stored until that stage. When a trap occurs, relevant CSRs store data and the control unit is notified to update the program counter in the next cycle. This cycle delay removes a long path in the design because an exception decision does not affect any program counter update or I-SPM fetch in the same cycle. The `SRET` instruction returns privilege level to user mode.

Instruction Format	Instruction	Thread cycles at $f =$			
		1/1	1/2	1/3	1/4
`INST`	`SCALL`	4	2	2	1
`INST`	`SRET`	1	1	1	1

Table 3.5: Thread cycle count for system instructions at different hardware thread scheduling frequencies.

3.2 Hardware Thread Scheduling

The purpose of FlexPRET's hardware thread scheduler is to provide predictable and isolated execution to hard real-time threads (HRTTs) while allowing soft real-time threads (SRTTs) to efficiently utilize spare cycles. The datapath and control unit support an arbitrary interleaving of hardware threads, unlike other predictable fine-grained multithreaded processors, so the scheduler must only meet the HRTT and SRTT property requirements. There are multiple scheduler implementations that meet these requirements, and our implementation is designed for flexibility with minimal complexity. Scheduling must be implemented in hardware to make decisions at the cycle granularity, but it is configurable by software for more flexibility.

3.2.1 Constraints on Scheduling

Careful processor design only facilitates isolated and predictable HRTTs. For isolation, the specific cycles each HRTT is scheduled must be independent of the execution of the other hardware threads. In other words, no information from other hardware threads is used to schedule an HRTT. In theory, each HRTT could request its own scheduling sequence based on

program flow. Unfortunately, this would require complex hardware to construct a schedule that fulfills all requests—which may not be feasible.

Example 3.1. If T_1 requests scheduling every 2nd cycle (scheduling frequency 1/2) and T_2 every 3rd cycle (scheduling frequency 1/3), no feasible schedule exists even though utilization is under one. The diagram below shows an attempt at scheduling, where T_1 and T_2 represent requests or granted requests and ? for conflicts.

T_1 *requests*	T_1	_	T_1	_	T_1	_	T_1	_	T_1	_	T_1	_
T_2 *requests*	_	T_2	_	_	T_2	_	_	T_2	_	_	T_2	_
schedule	T_1	T_2	T_1	_	?	_	T_1	T_2	T_1	_	?	_

Alternatively, the hardware scheduler can use a predefined schedule where each cycle is reserved for some hardware thread. To limit the amount of memory needed to store this schedule, a short scheduling sequence is periodically repeated. For example, T_1 T_2 T_1 T_3 could be periodically repeated to have T_1 execute every 2nd cycle, T_2 every 4th cycle, and T_3 every 4th cycle (scheduling frequencies of 1/2, 1/4, and 1/4). Depending on program flow, a hardware thread may not use its reserved cycle, but provided an HRTT only uses its reserved cycles, its execution is isolated by the hardware thread scheduler.

The thread cycle latency of each instruction along a program path depends on pipeline spacing between instructions from the same hardware thread, which is determined by scheduling. For periodic scheduling, the latency of each instruction along a program path is known and constant—determining execution time is trivial for any program segment. A non-periodic scheduling with a large minimum spacing, such as over four cycles, also provides this property.

When scheduling produces short, non-periodic spacing between instructions from the same hardware thread, the thread cycle latency of each instruction varies. Determining the latency of each instruction would require knowing the phase relation between the schedule and instruction sequence, unlikely for programs that have more than a single path.

Example 3.2. Consider the following non-periodic scheduling of hardware thread T_0 over a period of 6 cycles, which demonstrates variable latency for a branch taken instruction.

Cycle	0	1	2	3	4	5
Schedule	T_0	_	T_0	_	_	T_0

If a branch instruction starts executing in cycle 0 and is taken, the address and condition are not known by cycle 2, so the next instruction starts executing in cycle 5. If the same instruction starts executing in cycle 2, the address and condition are known by cycle 5. The same instruction takes 2 thread cycles in the first case and 1 thread cycle in the second case.

For efficiency, if it is known a priori that a scheduled thread will not fetch or complete its next instruction, that cycle should be allocated to a different hardware thread. Examples of

this include waiting for a timer to expire or an external interrupt. Which cycles are available depends on the execution of each thread, so HRTTs cannot use them without sacrificing isolation. SRTTs use these available cycles to increase both thread and overall processor throughput, at the cost of isolation and predictability. The general idea is each HRTT has a constant scheduling frequency, and SRTTs share unused cycles.

Each hardware thread, either a HRTT or SRTT, is always in one of two states: *sleeping* if it does not need to be scheduled until some later cycle, *active* otherwise. The hardware thread scheduler will only schedule active threads. By allowing both software and hardware to change the status of a hardware thread, the processor can efficiently implement common mechanisms. For example, a program running on a HRTT can put itself to sleep, allowing its reserved cycles to be used by any SRTT until a timer or I/O peripheral activates the HRTT again—a rapid, event-driven response without wasting cycles polling.

Thus far, to satisfy the requirements of HRTTs and SRTTs, we have proposed several constraints on hardware thread scheduling but have not described a concrete implementation. Before the selected implementation is described, consider some examples of valid schedules that meet different contrived requirements.

Example 3.3. 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 the desired scheduling frequency of T_0 is 1/4, and T_1 is 1/2, then a resulting schedule could be as follows:

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

T_0 is the only active HRTT and is scheduled every forth cycle to satisfy its scheduling frequency. In this example, the rest of the cycles are shared round-robin between active SRTTs T_2 and T_3. For T_2, as well as T_3, the spacing between subsequent instructions varies.

Example 3.4. Consider the same system as Example 3.3, but HRTT T_1 and SRTT T_4 are now active. Here are two possible schedules:

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

Now only one out of every four cycles is used for SRTTs. In the first schedule, a round-robin policy is used, and in the second, T_2 has priority; the choice has performance implications for SRTTs but doesn't affect HRTTs. An SRTT could also be scheduled like an HRTT to guarantee a minimum scheduling frequency. The overall throughput of a fine-grained multithreaded processor is higher when more hardware threads are interleaved in the pipeline, so a round-robin schedule will provide higher overall throughput than a priority-based schedule.

This scheduling technique is not possible in other fine-grained multithreaded processors. PTARM [43] supports a restricted version of only HRTTs by a round-robin rotation through a fixed number of hardware threads, but cycles are unused if any thread is sleeping. In addition to scheduling similarly to PTARM, XMOS [44] can support a restricted version of only SRTTs by a round-robin rotation through all active hardware threads, efficiently using cycles but with reduced isolation.

3.2.2 Scheduler Implementation

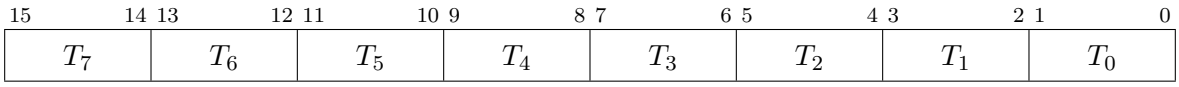
We designed a hardware thread scheduler to provide flexible scheduling options for both HRTTs and SRTTs while minimizing complexity. The design is intended to keep scheduling out of the critical timing path, leaving the maximum processor frequency unaffected, with a small area footprint. Certain cycles in a repeating sequence are reserved for specific hardware threads, set by a configuration register. If a cycle would be unused, such as when the specified hardware thread is not active, an active SRTT is selected in round-robin order.

By using configuration registers that are modifiable by software, many different schedules are possible, although limited by the length of the repeating sequence. In many cases a single configuration at startup would be sufficient, but runtime changes are also supported. To protect the processor against accidental or malicious schedule modifications, a hardware thread requires privileged mode to modify the scheduling CSRs.

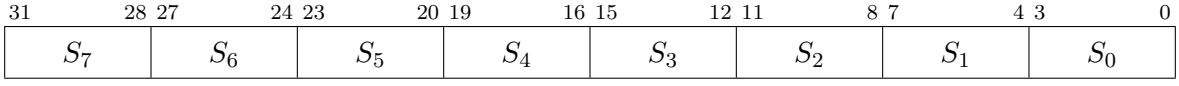
The *thread mode CSR* maintains the scheduling status of each hardware thread. A thread can be in one of four modes: active HRTT (HA), sleeping HRTT (HZ), active SRTT (SA), or sleeping SRTT (SZ). This requires two bits to encode the mode per hardware thread, which is 16 bits for an 8 hardware thread processor, as shown in Figure 3.2a.

The *slots CSR* has slots that specify cycle reservations within the repeating sequence. Each slot can have one of the following values: T_x where x is a hardware thread ID, S to indicate any active SRTT, or D to disable. Provided the processor contains less than 14 hardware threads, each slot can be encoded in 4 bits. For single cycle operation to a 32-bit register, 8 slots are used, as shown in Figure 3.2b. Both of these configuration registers are not modified until the execute stage, so the schedule change is not immediate.

The scheduling logic can be conceptually viewed as a combination of two schedulers. The higher priority scheduler round-robins through each non-disabled slot in the slots CSR, observing the value of the current slot. If the value corresponds to a hardware thread that is active, that hardware thread is scheduled the next cycle; otherwise the scheduling decision is delegated to the lower priority scheduler. Only upon delegation, the lower priority scheduler round-robins through active SRTTs. Pseudocode for scheduling is shown in Algorithm 3.1, even though the implementation is in hardware and not software.



(a) The thread mode CSR where $T_i \in \{HA, HZ, SA, SZ\}$.



(b) The slots CSR where $S_i \in \{T_0, T_1, ..., T_7, S, D\}$.

Figure 3.2: Thread scheduling is configured with two CSRs.

Algorithm 3.1 Pseudocode for FlexPRET's hardware thread scheduler.

```
1: procedure NEXT_TID(slots, slots_last, tmodes, tmodes_last)
2:     tid ← 0
3:     valid, s ← NEXT_ENABLED_SLOT(slots, slots_last)              ▷ round-robin
4:     if valid then
5:         slot ← slots(s)
6:         if slot ≠ S and (tmodes(slot) == HA or tmodes(slot) == SA) then
7:             tid ← slot
8:         else
9:             valid, t ← NEXT_ACTIVE_SRTT(tmodes, tmodes_last)     ▷ round-robin
10:            if valid then
11:                tid ← t
12:                tmodes_last ← t
13:            end if
14:        end if
15:        slots_last ← s
16:    end if
17:    return tid, valid, slots_last, tmodes_last
18: end procedure
```

Scheduling using periodic slots enables HRTTs to have a constant scheduling frequency, but it is still the responsibility of the programmer or compiler to assign the slots accordingly. SRTTs are also allowed to have reserved slots to guarantee minimum scheduling requirements. The D value enables repeating sequences with lengths fewer than 8. For example, a scheduling frequency of 1/7 is not achievable unless one of the eight slots is disabled.

A possible assignment of the slots CSRs that implements the first schedule in both Example 3.3 and 3.4 is shown in Figure 3.3; 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 second schedule in Example 3.4 could also be implemented by a software scheduler that sleeps all SRTTs without the highest priority.

The hardware implementation uses round-robin arbiters and other logic. The round-robin arbiter uses a mask-based approach, but other implementations are possible. In the mask-based approach, the requesters are in a fixed order, and both masked and unmasked requests are provided to priority arbiters. All the requesters before and including the last one granted are masked out, and the highest priority remaining request is granted; if the masked request is empty, then the unmasked request is used instead, restarting at the beginning.

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

(a) The thread mode CSR for Example 3.3.

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

(b) The thread mode CSR for Example 3.4.

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

(c) The slots CSR for Examples 3.3 and 3.4.

Figure 3.3: Thread scheduling CSRs for Examples 3.3 and 3.4.

The examples presented only set the slots CSR once. By changing either the slots or thread mode CSR during run-time, more flexible scheduling sequences are possible—useful for applications with varying concurrency and deadlines. For example, an HRTT's scheduling frequency could be increased to meet a deadline it would otherwise miss, or a processor could switch modes of operation.

3.3 Timing Instructions

We extend the RISC-V ISA with timing instructions, introduced in Section 2.3, to express real-time semantics. The supported pseudo-instructions are `get_time` to access the current time, `delay_until` to stall the program until some time, `interrupt_on_expire` to interrupt the program at some time, and `deactivate_exception` to disable the interrupt mechanism. Compared to previous PRET architectures supporting timing instructions [43], [46], the flexibility of FlexPRET complicates implementation but improves performance: a hardware thread will sleep instead of stalling execution to let active SRTTs use spare cycles. In addition to specifying the timing behavior of a single hardware thread, time can also synchronize operations between hardware threads.

In FlexPRET, time is represented by a nanosecond value instead of counting clock ticks, starting at zero when powered on. At 100 MHz, ten nanoseconds per clock tick means any time interval requires approximately three extra bits. We accept this overhead for a few reasons. First, a time value in software is independent of the processor's clock frequency, improving binary code portability and supporting dynamic frequency scaling of the processor clock. Second, time could be synchronized in a distributed system by adjusting the timer increment value and maintaining sub nanosecond counts. By restricting the increment to a narrow range of values and not allowing any other modification, relative timing specified by timing instructions cannot experience much variation. For clock synchronization pro-

tocols such as IEEE 1588 [72], software converts relative to absolute real-time by adding an offset, which provides a mechanism for discrete clock jumps, sometimes required when synchronization is far off.

The width of the clock is set to 32 bits but configurable to less bits as well to reduce area. The timer overflows every $2^{32} \approx 4.29$ seconds, which means longer relative timing behavior must be supported with software. In the first version of FlexPRET [91] we supported up to 64 bits, which only overflows every 584 years, but decided the added complexity is too much. With a 32-bit datapath and one register write port, retrieving the clock value or calculating a compare value would require multiple instructions and atomicity concerns for 64-bit time values. Also, additional hardware is required to store and compare more bits of time, and it scales with the number of hardware threads.

Software accesses the current clock value with the `get_time` pseudo-instruction. FlexPRET defines this instruction as `GT`, storing the current clock value in a destination register. The value returned is the time of instruction commit relative to processor power on or reset. Time could be defined as another point (e.g. instruction fetch); the important part is consistency. The commit point is when the instruction irreversibly modifies either processor state or external I/O, a natural choice.

Each hardware thread has a register that is compared against the current time every clock cycle, set using the `CMP` instruction. The comparison result is used to implement the `delay_until` and `exception_on_expire` pseudo-instructions, triggering a wake, interrupt, or exception for the hardware thread. Setting this register will also disable any triggers (e.g. `CMP x0`) to implement the `deactivate_exception` pseudo-instruction. To reduce processor area, this register is shared between all timing instructions on its hardware thread, but the processor could be extended in the future to support multiple comparisons per hardware thread. Also, using a separate `CMP` instruction makes the sharing explicit and visible to the programmer.

The comparison operation must carefully handle the periodic overflows resulting from the finite bit lengths of both the time and compare registers. An equality comparison is not used because the `delay_until` and `interrupt_on_expire` pseudo-instructions do not always provide a future time. Correct operation would require setting the compare time before the clock time reaches it (else wait another 2^{32} ns), and the compare time would need to be equal to one of the incremented values of the clock time.

A direct 32-bit unsigned comparison would also be problematic whenever a computed compare time overflows. A simple example using a 4-bit time value (0-15) can demonstrate this problem using expiration defined as ($time \geq compare$). To expire 3 ticks in the future for current time of 1, the compare time is set to 4, and expiration occurs at time 4. If the current time is instead 14, the compare time is set to 1 because of overflow, and expiration incorrectly occurs immediately. Although the overflow information could be used to wait on comparison until the time value overflows as well, this would require additional software or hardware support.

Instead, the comparison operation used for expiration of the value set by `CMP` is ($time - compare \geq 0$) with unsigned subtraction and considering the result as signed, which is a

technique used in some existing hardware timers. The comparison is correct whenever the intended comparison time is within 2^{31} nanoseconds of the current time, either before or after. By using unsigned subtraction instead of unsigned comparison, software and hardware can ignore addition overflow and functionality is still correct.

Instruction Format	Instruction	Thread cycles at $f =$			
		1/1	1/2	1/3	1/4
`INST rd`	`GT`	1	1	1	1
`INST rs1`	`CMP`	1	1	1	1

Table 3.6: Thread cycle count for clock interaction instructions at different hardware thread scheduling frequencies.

3.3.1 Timed Delay

A `delay_until` pseudo-instruction stalls program execution until time has passed the specified value, supported by FlexPRET with `DU` and `WU` instructions. A simple implementation would operate as a conditional branch back to itself unless the specified time value has passed, but this wastes cycles that in FlexPRET could be used by another thread. An alternative approach is to always increment the program counter to the next instruction but sleep the hardware thread if time has not passed the specified value. The pipeline is used by other threads so an external comparison unit needs to wake the hardware thread once time has passed the specified value, which is set using the `CMP` instruction, and then the next instruction executes.

This approach is problematic if interrupts are enabled. If a hardware thread is sleeping, an interrupt will wake the hardware thread, execute the interrupt handler routine, and return to the program counter that was interrupted. If the expiration time has not occurred, execution is incorrect because program execution will immediately continue after the `delay_until` pseudo-instruction, unless software is forced to prevent this problem by modifying the return point. The solution we use is not to update the program counter but still sleep the hardware thread; the instruction is replayed on wake and will either proceed or put the hardware thread back to sleep depending on the current time.

For the `delay_until` operation, FlexPRET uses a `CMP` instruction followed by a `DU` instruction. At the commit point, `CMP` will disable any existing timer triggers and set the provided value to be compared against the clock. When the `DU` instruction reaches the execute stage, a conditional branch occurs. If time has expired, using the result of the comparison, execution proceeds as normal. If time has not expired, the hardware thread sets the program counter to that of the `DU` instruction and flushes the pipeline, just like a branch instruction.

This is enough to correctly execute `delay_until` semantics, but to share cycles the hardware thread must sleep. If time has not expired, the hardware thread is put to sleep and

the timer is set to trigger a wake upon expiration. When the hardware thread wakes, the `DU` instruction executes again, which prevents an interrupt from causing execution to proceed prematurely. The `WU` instruction is identical to `DU` but does increment the program counter, providing a mechanism for the processor to wait until an interrupt occurs with a timeout value. If the comparison time set by `CMP` has expired when the `DU` or `WU` instruction reaches the commit point, the instruction uses one thread cycle; otherwise the pipeline flush and thread mode change uses multiple thread cycles.

Instruction Format	Instruction	Thread cycles at $f =$			
		1/1	1/2	1/3	1/4
`INST`	`DU`	5 or 1	3 or 1	2 or 1	2 or 1
`INST`	`WU`	5 or 1	3 or 1	2 or 1	2 or 1

Table 3.7: Thread cycle count for timed delay instructions at different hardware thread scheduling frequencies.

3.3.2 Timed Interrupt

An `interrupt_on_expire` pseudo-instruction will interrupt program execution once a clock time is reached unless disabled by a later instruction, supported by FlexPRET with `IE` and `EE` instructions. All exceptions, interrupts, and software traps are handled during the execute stage. For `interrupt_on_expire`, a timer can use the existing trap mechanism and trigger an interrupt by the execute stage. As previously mentioned, the trap mechanism will store the current program counter and cause, flush the pipeline, and set the program counter to the trap handling routine.

FlexPRET again uses a combination of two instructions: `CMP` followed by `IE`. As with delay until, `CMP` disables any existing triggers and sets the comparison value. When `IE` commits at the end of the execute stage, the timer is set to trigger an interrupt upon expiration. The `EE` instruction is identical to `IE` but uses a different cause identifier as an exception instead of an interrupt. The programmer should use `IE` for interrupts that should occur for correct program behavior, and `EE` for run-time detection of unexpected timing violations.

Instruction Format	Instruction	Thread cycles at $f =$			
		1/1	1/2	1/3	1/4
`INST`	`IE`	1	1	1	1
`INST`	`EE`	1	1	1	1

Table 3.8: Thread cycle count for timed interrupt instructions at different hardware thread scheduling frequencies.

3.3.3 Alternative Implementations

We used this implementation of timing instructions to provide flexible functionality without adding too much complexity, but other implementations are possible. The `CMP` instruction could become part of the `DU` and `IE` instructions, reducing code size, but a deactivate interrupt instruction would need to be added. Memory or a general-purpose register would need to store the future time value until used by the `DU` or `IE` instruction. Also, the branch condition checker would need to be used for the `DU` instruction because the result is needed during the execute stage but the compare register cannot be modified until the commit point at the end of the execute stage.

The current implementation only allows one timing instruction in operation at any one point. For example, if an `IE` instructions is monitoring a block of code for a deadline miss, this block of code cannot use the `DU` instruction (although could use the `GT` instruction and a conditional branch). Multiple timers per hardware thread could be added, but hardware complexity would increase. In addition to comparison storage and evaluation, an interrupt could occur while a hardware thread is in the sleep state, adding more control logic and paths both setting and reading the program counters. For further flexibility, each timer could be a shared resource available to any hardware thread, but resource availability would need to be guaranteed for correct program execution.

3.4 Memory Hierarchy

Memory hierarchy design affects functionality, predictability, and performance, and different trade-offs are made depending on the application area. FlexPRET has separate memories for instruction and data, uses software-managed scratchpad memories instead of caches, and only allows access to permissible regions of the scratchpad memories. As a consequence, all valid memory accesses always succeed and are single cycle.

Without separate instruction and data memories, only one memory operation could occur every cycle; during a load or store operation, a new instruction would not be fetched, reducing overall throughput of the processor. Although typically implemented with instruction and data caches instead of scratchpads, this is a standard design decision for processors with similar or higher complexity. For further justification, if only one scratchpad is shared between code and data, a hardware thread's memory operation can only occur during a scheduled cycle that may not align with the memory stage, requiring more control logic and data storage. For example, each hardware thread would need to store address and data for a memory operation until the next scheduled cycle, then prevent the fetch.

FlexPRET optionally allows memory operations to the instruction memory instead of data memory, improving the allowed functionality. A potential use case is a bootloader, where a small program initialized in instruction memory can retrieve instructions over the peripheral bus, then store them into instruction memory. These stores are not guaranteed to be visible to instruction fetching until the program executes a synchronization instruction,

called `FENCE.I`. In FlexPRET, this instruction just stalls the pipeline until all preceding instructions have committed. Other potential uses include just-in-time compilation and self-modifying code.

The removal of caches in favor of scratchpad memories is a contentious but important decision. The main justification is predictability: with a cache, the execution time of a memory instruction depends on cache state, whereas with a scratchpad memory, a memory instruction always succeeds. If the tasks are small and fit entirely within scratchpad, as occurs in many simple embedded applications, there are no complications. If the tasks do not fit entirely within scratchpad, there are several options.

One option is separating the safety- or time-critical sections of an application from the rest. FlexPRET can execute these sections as a coprocessor while the rest of the program runs on a high performance main processor with caching. Another option is scratchpad memory management, where the scratchpad contents are dynamically and explicitly controlled by software, written by the programmer or compiler, through direct memory access (DMA) operations.

If the unpredictability of caching is acceptable, FlexPRET could be modified to use caches, even if just for SRTTs. To maintain temporal isolation, caches would need to be private for each hardware thread. Also, each hardware thread would need to handle cache misses without disturbing thread scheduling.

Memory instructions can only access the scratchpad memories and have no direct connections to main memory. Because of the relatively large latencies of external memories, any direct connection would result in memory instruction latency depending on the address accessed, an additional burden on timing analysis. DRAM memories are suited for burst accesses, such as 64 bytes; a direct DRAM operation for a single 32-bit value would be quite slow and wasteful of throughput resources. Instead, all memories operations must be achieved using DMA operations on the scratchpad and main memory.

Virtual addressing is not supported to reduce datapath and control unit complexity and to maintain single cycle memories accesses, which only occurs on translation hits for virtual addressing. For isolation between threads, FlexPRET has a simple memory management unit. Each hardware thread can read anywhere in memory, but only write to certain regions. These regions can either be written by all hardware threads (shared) or only one hardware thread (private).

If a program dynamically manages the scratchpad contents with DMA operations, timing analysis requires the latency bound of each transfer. Flash memory is often used for instruction storage because data is persistent. DRAM memories are often used as main memory for both instructions and data, but are slower than SRAMs and have variable access times. If the latency of DRAM requests using a certain DRAM controller cannot be bounded or the bound is too large, a more predictable DRAM controller can be used. Much like the design of a predictable processor, predictable DRAM controllers can prioritize isolation [87], overall throughput [89], or a combination [90].

3.4.1 Implementation

The size of each scratchpad memory is configurable, set to 16 kB by default. The I-SPM has two ports: one read port for instruction fetch, and one read/write port both externally exposed and connected to the load-store unit. The load-store unit has priority for memory access, and the external connection can be to a bus for DMA operations. The D-SPM also has two ports: one read/write port connected to the load-store unit, and another read/write port externally exposed. This external connection also allows DMA transfer to the scratchpad memory over a bus. The load-store unit uses the address to determine which memory to access: I-SPM, D-SPM, or peripheral bus.

The I-SPM and D-SPM are divided into 8 equally sized regions for write protection using the upper 3 address bits. Each region is set as private to a specific thread ID or shared. The register format for the I-SPM protection CSR and D-SPM protection CSR is as follows, with $R_i \in \{T_0, T_1, ...T_7, shared\}$.

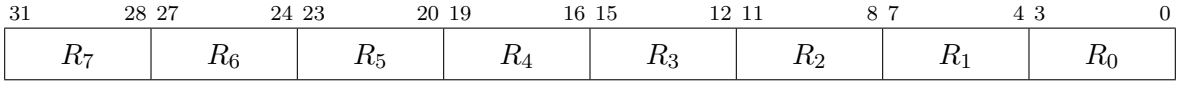


Figure 3.4: A CSR configures scratchpad memory regions as shared or private for a hardware thread.

Although not implemented, with multiple CSRs for each scratchpad, more than 8 regions could be supported. If a hardware thread attempts a store to a region without permission, an exception is thrown.

To order data and I/O memory accesses between multiple concurrent threads of execution, the RISC-V ISA requires a `FENCE` instruction. In FlexPRET, the `FENCE` instruction does not do anything—instructions commit in order and all memory operations to D-SPM or the peripheral bus are guaranteed to succeed in a single cycle. The `FENCE.I` instruction, which ensures previous stores to instruction memory complete before a fetch occurs from the same memory location, does require processor support. The `FENCE.I` instruction prevents a fetch from the same hardware thread until it commits at the end of the execute stage, killing instructions if needed, to ensure any preceding store instruction has written to memory.

Instruction Format	Instruction	Thread cycles at $f =$			
		1/1	1/2	1/3	1/4
`INST`	`FENCE`	1	1	1	1
`INST`	`FENCE.I`	4	2	2	1

Table 3.9: Thread cycle count for ordering constraint instructions at different hardware thread scheduling frequencies.

3.5 Peripheral Bus

The processor uses memory-mapped registers accessible through load and store memory operations to communicate with external hardware peripherals over the *peripheral bus*. External hardware could include a direct memory access (DMA) engine for scratchpad memory management, a network-on-chip in a multicore system, and common communication peripherals such as Ethernet or USB. Single cycle load and store operations to the bus maintain isolation and predictability for hardware threads.

The *bus protocol* defines how components communicate over the bus. Multiple components conceptually share the control and data lines of the bus to prevent point to point connections for every pairing of components. A *master* component starts all transactions, and *slave* components respond. With a single master, such as a processor, there is no contention for bus access, but the master must control every transaction. Multiple masters remove this requirement and allow flexible communication, but preventing conflicts requires an arbitration scheme. For higher throughput over long distances on or off chip, some bus protocols support pipelining bursts of data.

The challenge for FlexPRET is maintaining predictable communication while meeting bus throughput requirements. The processor cannot directly move data from the DRAM to either the I-SPM or D-SPM because the resulting throughput would be too low: it would require three instructions (address calculation, load, store) to move one new instruction into the I-SPM. Instead, a DMA engine on a high-throughput bus, such as AXI4, could manage data transfer with only high-level commands from the processor to move blocks of data.

One option is directly connecting FlexPRET to this bus that supports multi-cycle data transfers and multiple masters. A bus access only requires a single instruction, but this instruction has variable latency because of bus contention and arbitration, which use *handshaking* mechanisms. Consequently, timing analysis would need to both determine the address used in a memory operation and be provided with a timing bound for this address.

Another option, and the option used by FlexPRET, is to only connect the datapath to a simple synchronous bus with guaranteed single-cycle accesses, maintaining constant latency for all memory operations. This moves control mechanisms, such as waiting for valid data, to software. For example, if a peripheral does not always have valid data, a control register at a different address stores a valid bit. Instead of stalling on a single variable latency load instruction, software polls this control register until data is valid, then performs a single-cycle load instruction. Although this mechanism requires more code, all memory instructions have constant latency. By using a timeout on polling, timing bounds are part of the program itself and visible during timing analysis.

By moving handshaking to software and adding a gateway, the processor can still communicate on more complex bus protocols, such as AXI4. A *gateway* uses hardware logic and buffers to provide a bridge between different bus protocols. A possible configuration is shown in Figure 3.5. A DMA engine on a high-throughput bus moves data between the DRAM and the SPMs, and it also serves as a gateway from the processor's peripheral bus, which provides high-level commands for data movement.

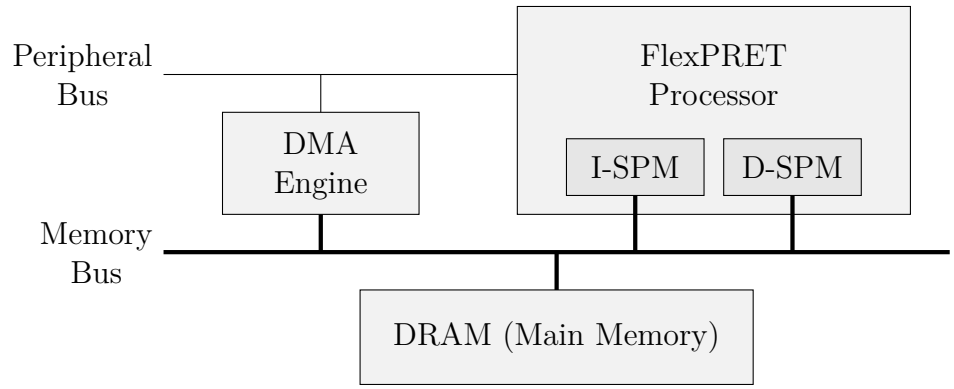


Figure 3.5: A possible configuration with a peripheral bus and memory bus.

The interface for FlexPRET's single-cycle access peripheral bus is defined as follows:

```
1 output [9:0] address,
2 output enable,
3 output write,
4 output data_to_bus,
5 input data_from_bus
```

For a peripheral bus read, the datapath sets the outputs towards the end of the execute stage, just as with a D-SPM read. The lower bits of `address` are determined by software and the upper bits contain the thread ID, `enable` is set high, and `write` is set low. After the positive clock edge, the processor data input `data_from_bus` must be set by the bus before the next positive clock edge, as it is registered at the end of the memory stage. To not increase the critical path of the processor, the bus latency should be less than D-SPM latency. For a peripheral bus write, the datapath also sets the outputs towards the end of the execute stage, but with `write` set high and `data_to_bus` set with write data. At the positive clock edge, the bus must store this data, as outputs can change in the next clock cycle.

Multiple bus implementations support this custom interface. In one possible implementation, the bus provides `address` and `data_to_bus` to all peripherals, but only provides the `enable` and `write` signals to the peripheral in the correct address range. The `data_from_bus` is set using a multiplexer connected to all peripherals, with selection depending on the address range of the previous clock cycle.

3.6 I/O

With predictable and controllable timing behavior, FlexPRET is well suited for interacting with digital I/O pins. Fine-grained multithreading with flexible hardware thread scheduling supports multiple independent I/O operations, whereas a single-threaded processor requires a context switch for multiple I/O operations, and only one operation has priority. Possible

applications include using software to replace hardware peripherals and performing low-level I/O processing to offload a main processor, both discussed further in Section 5.2. We designed FlexPRET's I/O mechanisms to support these applications, either directly or through additional hardware peripherals.

I/O interaction can be implemented almost entirely in hardware, software, or a hybrid of both. To keep the base implementation small and simple, but at the cost of accuracy and precision, FlexPRET uses minimal hardware support for I/O mechanisms. Every processor clock cycle, general-purpose input registers read the value on the corresponding input pins, and general-purpose output registers write the value on the corresponding output pins. These input and output registers are implemented as CSRs instead of memory-mapped registers for faster access: setting and clearing bits is a single instruction atomic operation instead of a three-instruction read-modify-write operation. Further hardware support for I/O can be added as a hardware peripheral, controlled using the peripheral bus.

For flexibility and isolation, there must be a mechanism to assign pins to a specific hardware thread; otherwise, pin allocation is fixed or a hardware thread can disrupt another's I/O operations. There is overhead for storing assignments, so pins are grouped into either input or output *ports* that share an assignment, with a configurable number of pins in each port (up to 32). A CSR stores the hardware thread assignments for each port. Each port has one CSR for access to the pins, which is read-only for an input port or writable for an output port. The combination of pins and ports is configurable in the design, and a CSR that controls permission to up to 8 output ports is as follows, with $P_i \in \{T_0, T_1, ...T_7, \mathit{shared}\}$.

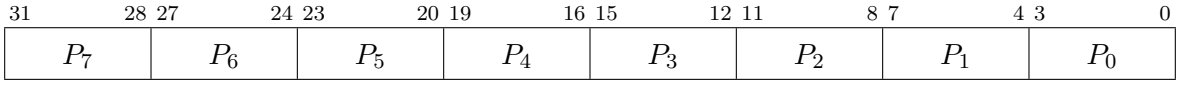


Figure 3.6: A CSR configures output ports, which consist of one or more output pins, as shared or private for a hardware thread.

With timing instructions, I/O can be read or written at specific times. For example, a `DU` instruction followed by an I/O instruction would sleep the hardware thread until a specific time, then wake and perform the I/O operation. The main limitation of the selected implementation is precision and accuracy—the I/O operation does not immediately occur at the specified time. Hardware thread scheduling determines how many clock cycles after time expiration that the I/O instruction is fetched, and this instruction does not affect the I/O pin until after the execute stage. The accuracy and precision are further discussed in Section 4.1. Even if I/O operations are cycle accurate, the update rate of I/O is limited by software execution time to compute the next values. For a 100 MHz processor, I/O operations can occur on the order of every thousand nanoseconds with a precision under one hundred nanoseconds, which is further described in Section 4.1.

3.6.1 Alternative Implementations

Clock cycle accurate I/O is possible with additional hardware support, controlled either by custom instructions or as a memory-mapped peripheral. For example, a port could read or write directly at a specified time instead of when an instruction executes. To support this, each port would need additional registers to store the specified time and write value, as well as logic to compare against the current time. The bit width of the comparison would determine how long in advance the port operation could be set, which may need to be checked during timing analysis.

As demonstrated by the I/O functionality of XMOS [44], there are many options for hardware-supported software-controlled I/O. In addition to reading or writing at a specific clock cycle, ports on XMOS processors can use different clocks, wait for specific input values then timestamp, and perform serialization and deserialization. We envision this level of hardware support being implemented as a peripheral for specific precision-timed I/O applications but decided not to add this complexity to the base version of FlexPRET.

Chapter 4

Implementation and Evaluation

We implement and evaluate the RISC-V FlexPRET processor using both cycle-accurate simulation and FPGA deployment. The cycle-accurate simulator is useful for prototyping, debugging, and benchmarking, and the FPGA deployment demonstrates feasibility and provides quantitative hardware resource costs of the proposed architectural techniques. The design is coded in Chisel [95], a hardware construction language that generates both Verilog code and a cycle-accurate C++ based simulator. This chapter evaluates the application-independent properties of FlexPRET: the isolation and predictability of hardware threads (Section 4.1), average-case performance for different thread scheduling frequencies (Section 4.2), FPGA resource utilization (Section 4.3), and power implications (Section 4.4). Chapter 5 evaluates application-specific properties of FlexPRET.

Chisel allows us to parameterize the code to produce different processor variations, including two baseline processors that remove functionality for comparison purposes. We implement baseline processors instead of comparing to existing processors to negate discrepancies caused by instruction set architecture (ISA) selection, microarchitecture design decisions, hardware device properties, and optimization techniques. The intent is to evaluate the incremental cost of FlexPRET's hardware thread scheduler, isolation mechanisms, and timing instructions, as these techniques are applicable to other microarchitectures.

The first baseline processor, called *Base-1T*, is a variation without multithreading. Base-1T is a classical 5-stage processor with predict not-taken branching, scratchpad memories (SPMs), and forwarding paths. It has the lowest hardware complexity of all processor variations. When FlexPRET schedules the same hardware thread every cycle, its run-time behavior is identical to this baseline processor.

The second baseline processor, called *Base-4T-RR*, is a variation without flexible hardware thread scheduling. Base-4T-RR schedules 4 hardware threads in a fixed round-robin order, similar to both PTARM [33] and a configuration of XMOS [44]. It does not use forwarding paths or flush logic because hardware threads have a fixed minimum spacing in the pipeline. When FlexPRET schedules four hard real-time threads (HRTTs) in a fixed round-robin order, its run-time behavior is identical to this baseline processor.

From the perspective of cycle-accurate functionality, FlexPRET is a superset of both baseline processors: run-time software configuration of the hardware thread scheduler allows identical scheduling to either baseline processor. Both FlexPRET and the baseline processors implement RV32I [92], which does not include ISA extensions for floating point arithmetic, atomic memory operations, or integer multiplication and division instructions. Future variations of FlexPRET could support these application-specific RISC-V extensions, but these extensions are not needed to demonstrate the proposed microarchitectural techniques.

To verify implementation correctness for these processors, both the cycle-accurate simulator and FPGA deployment execute an assembly test suite and C benchmark programs, with results examined and compared. The simulator is useful for debugging, as generated traces contain the logical values of each wire every cycle. To execute different programs, the simulator initializes the instruction and data memory accordingly. Programs execute faster on FPGA, but debugging is more difficult because only data at the processor interface is visible (without adding additional debug hardware). On FPGA, different programs are executed by modifying the memory initialization data in the binary FPGA image.

4.1 Isolation and Predictability

As described in the previous chapter, with isolated pipeline operation, the timing behavior of each hardware thread only depends on its scheduling and program flow. Dependencies between hardware threads, such as shared data or communication, only affect program flow. This section summarizes the timing behavior of each instruction type and further analyses the timing behavior of timing and I/O instructions with respect to real-time.

Table 4.1 lists the latencies of each instruction under different hardware thread scheduling conditions, merging multiple tables from Chapter 3. For each constant scheduling frequency, a hardware thread must be scheduled a number of *thread cycles* to complete and enable execution of the next instruction. These scheduled cycles are unusable by other hardware threads. The number of thread cycles remains constant or decreases for slower scheduling frequencies because other hardware threads are hiding the latency of operations performed by the instruction. Except for branches and timing instructions, which conditionally stall, all instructions have constant thread cycle latencies at each scheduling frequency.

For a conditional branch, the latency depends on whether the branch is taken (3 thread cycles at $f = 1$) or not taken (always 1 cycle). The latency is known for a program path, however, as branch decisions differentiate one path from another. Timing instructions relate program execution to real-time at specific points, which is a goal of WCET analysis, but through direct control instead of analysis. The higher latencies for `DU` and `WU` instructions (5, 3, 2, and 2 thread cycles) occur when the instruction delays execution until a later time; execution time elapses but thread cycles are constant because the hardware thread sleeps and is ignored by the scheduler. If the scheduling frequency is not constant, often the case for soft real-time threads (SRTTs), instruction latencies are variable but bounded by the latencies for single-threaded scheduling ($f = 1$).

Instruction	Thread cycles at $f =$			
	1/1	1/2	1/3	1/4
`LUI, AUIPC`	1	1	1	1
`ADDI, SLTI, SLTIU, XORI, ORI, ANDI, SLLI, SRLI, SRAI`	1	1	1	1
`ADD, SUB, SLL, SLT, SLTU, XOR, SRL, SRA, OR, AND`	1	1	1	1
`JAL, JALR`	3	2	1	1
`BEQ, BNE, BLT, BGE, BLTU, BGEU`	3 or 1	2 or 1	1	1
`SB, SH, SW`	1	1	1	1
`LB, LH, LW, LBU, LHU`	2	1	1	1
`CSRRW, CSRRS, CSRRC`	1	1	1	1
`CSRRWI, CSRRSI, CSRRCI`	1	1	1	1
`SYSCALL, FENCE.I`	4	2	2	1
`SRET, FENCE`	1	1	1	1
`GT`	1	1	1	1
`CMP`	1	1	1	1
`DU, WU`	5 or 1	3 or 1	2 or 1	2 or 1
`EE, IE`	1	1	1	1

Table 4.1: Summary of thread cycle counts for instructions at different hardware thread scheduling frequencies.

Constant instruction latencies—independent of execution history—provide predictable behavior. For a program path of any length, such as a basic block, adding the latencies for each instruction on the path determines the exact number of thread cycles used to execute the path. Dividing thread cycles by a constant scheduling frequency then determines processor clock cycles. This greatly simplifies the hardware analysis and longest path portions of WCET analysis because assigning timing costs to path segments is trivial. Without constant latencies, the latencies depend on processor and program state when arriving at a path segment, which must be modeled for tight and safe bounds. Furthermore, if allowed, interrupts can also affect processor state and consequently instruction latency.

The processor clock speed relates execution time in processor cycles to execution time in seconds. Timing instructions enforce relationships between real-time in seconds and program locations, and execution time analysis infers timing information at other program

points. One application of timing instructions is precision-timed I/O operations. Because the simple I/O mechanism uses a separate instruction that modifies I/O pins only at the end of the execute stage, the accuracy and precision depends on thread scheduling. Most precise operation occurs if a hardware thread is fetched during a specific cycle, but its next scheduled cycle may be multiple cycles later; this undesirable phase offset is referred to as *scheduling delay*.

The following analysis of I/O timing behavior for a hardware thread assumes:

- constant scheduling frequency $f = 1/p$
- scheduling delay $d \in \{0, ..., p-1\}$
- processor clock period c
- register r_t containing time value t

Example 4.1. Consider the following code segment where the `IO` instruction performs an I/O operation.

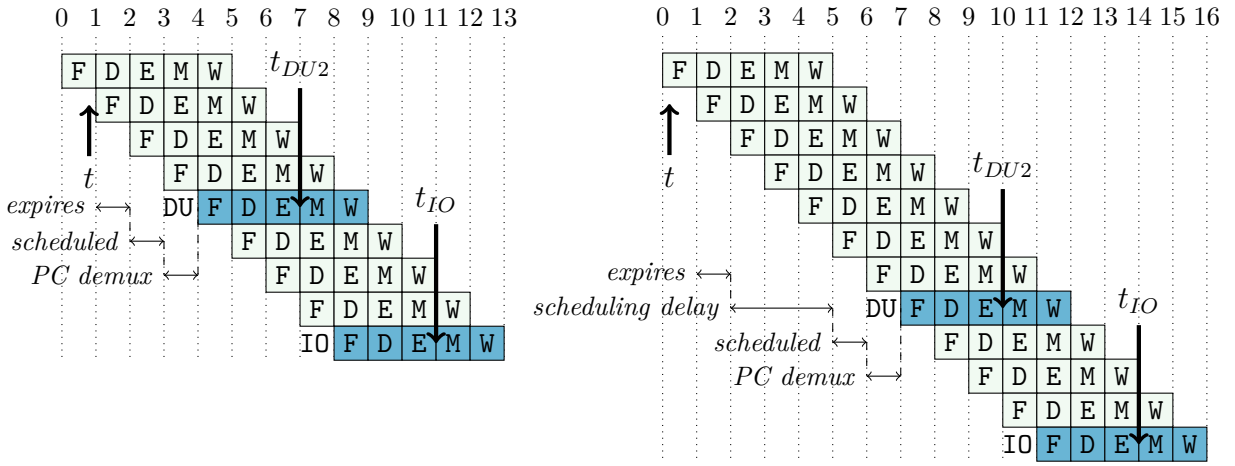
time at commit point	instruction
	`CMP` r_t
$t_{DU}, (t_{DU2})$	`DU`
t_{IO}	`IO`

Two possible situations exist depending on whether the comparison time expires before the `DU` instruction reaches the commit point: *wait* ($t_{DU} < t$) or *proceed* ($t_{DU} \geq t$). In the *proceed* situation, the `DU` instruction executes as a NOP, and the next instruction executed for the thread is the `IO` instruction, which is spaced in the pipeline by p. The I/O operation occurs at:

$$\begin{aligned} t_{DU} &\geq t \\ t_{IO} &= t_{DU} + c \times p \end{aligned} \tag{4.1}$$

In the *wait* situation, the hardware thread transitions to sleep mode, and the control unit flushes the pipeline and sets the program counter to the `DU` instruction. When the comparison expires (within a clock cycle after time t), it takes at least 6 clock cycles for the hardware thread to transition to active mode, be scheduled, and have the replayed `DU` instruction in the execute stage, at time t_{DU2}. In the worst-case, there is also a scheduling delay, and the expiration time is at the beginning of a clock cycle. Figure 4.1 illustrates both the minimum and maximum delay from time expiration until the `DU` instruction is in the execute stage. The `IO` instruction executes p cycles after the `DU` instruction, and the I/O operation occurs at:

$$\begin{aligned} t + c \times 6 \leq t_{DU2} &< t + c \times (6 + (p-1) + 1) \\ t + c \times 6 \leq t_{DU2} &< t + c \times (6 + p) \\ t_{IO} &= t_{DU2} + c \times p \end{aligned} \tag{4.2}$$



(a) Minimum delay

(b) Maximum delay

Figure 4.1: Pipeline behavior for the delay from time expiration (at t) to replay of `DU` instruction (at t_{DU2}) at scheduling frequency of 1/4, with lighter green indicating instructions from other hardware threads.

The `DU` instruction only provides a lower bound on timing behavior because the `DU` instruction executes when the program reaches it, with time t_{DU} possibly well past the set comparison time t. If t_{DU} is bounded, by timing analysis for example, the range of t_{IO} is bounded as well. If the relative comparison time is greater than the longest program path from a point, the *wait* situation always occurs, and Equation 4.2 can be rearranged using a new variable t_{adj}:

$$\begin{aligned} t + c \times (6 + p) \leq t_{IO} &< t + c \times (6 + p + p) \\ t_{adj} &= t + c \times (6 + p) \\ t_{adj} \leq t_{IO} &< t_{adj} + c \times p \end{aligned} \tag{4.3}$$

Except for variation caused by the scheduling delay d and the clock cycle granularity of expiration ($d + 1 = p$), all the other terms are known. Therefore, the programmer or compiler can adjust t, which is used by the `CMP` instruction, to compensate for the delay of waking the thread and executing the `IO` instruction. For example, if $f = 1/p = 1/4$ and $c = 10\ ns$, t is calculated from the desired t_{adj} by subtracting 100 ns, and the variation of t_{IO} is 40 ns ($c \times p$). For t_{IO} in a SRTT, p is not always constant and is replaced by the maximum spacing between scheduled cycles.

I/O applications with periodic and relative timing constraints are concerned with relative time duration between I/O mechanisms and not the absolute time.

Example 4.2. Consider the following code segment where the `DU` instruction executes before the comparison expires, and the I/O operations occur at periodically at t_{IOi}, where i is the loop iteration count.

time at commit point	instruction
	loop: `CMP` r_t
t_{DUi}, t_{DU2i}	`DU`
t_{IOi}	`IO`
	`ADDI` r_t, r_t, period
	`JAL` loop

The minimum and maximum relative time differences between I/O operations can be computed using the results from Equation 4.3.

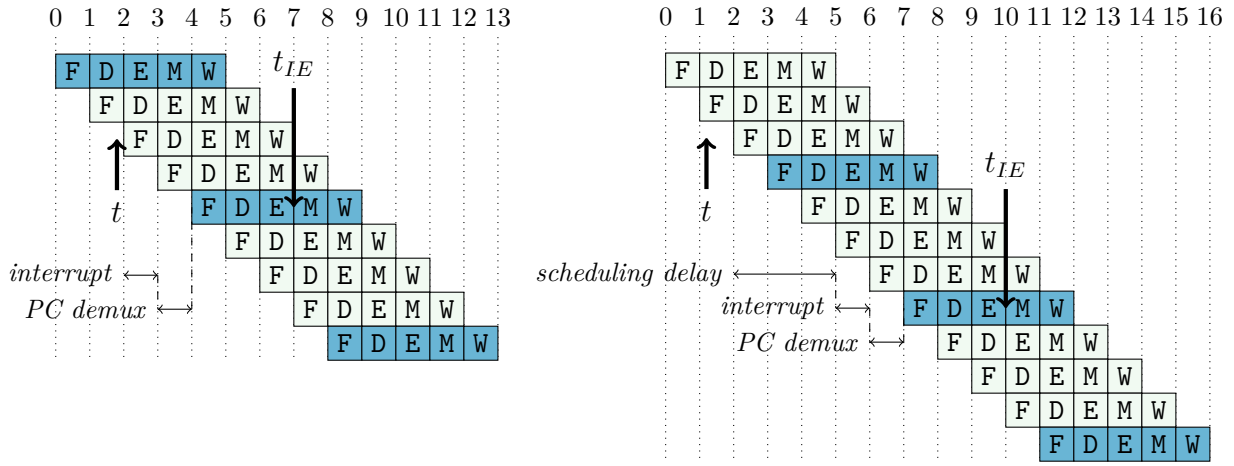
$$\begin{aligned} t_2 &= t_1 + \text{period} \\ t_1 \leq t_{IO1} &< t_1 + c \times p \\ t_2 \leq t_{IO2} &< t_2 + c \times p \\ \min(t_{IO2} - t_{IO1}) &= \text{period} - c \times p \qquad (4.4) \\ \max(t_{IO2} - t_{IO1}) &= \text{period} + c \times p \qquad (4.5) \end{aligned}$$

For $f = 1/p = 1/4$ and $c = 10\,ns$, the relative timing between two consecutive I/O operations is within 40 nanoseconds of the specified period. The error does not accumulate because r_t is incremented and not t_{IOi}. Also, if $p \times c$ is a multiple of the period value, then there is no phase shift in thread scheduling and consequently no variation in the relative timing of t_{IO}.

The other timing instruction usable for I/O operations is the `IE` instruction, which interrupts program execution when the current clock time reaches the comparison value set by the `CMP` instruction. Unless there are no other interrupts or exceptions allowed, the handler routine must first check that the cause of the interrupt was the `IE` mechanism before performing any I/O operations.

Example 4.3. Consider the following code segment where `IO` performs an I/O operation, and the handler must first execute code that requires h thread cycles.

time at commit point	instruction
	`CMP` r_t
	`IE`
	...
t_{IE}	handler: ...
t_{IO}	`IO`



(a) Minimum delay

(b) Maximum delay

Figure 4.2: Pipeline behavior for the delay caused by an earlier `IE` instruction from time expiration (at t) to the first instruction of handler code (at t_{IE}) at scheduling frequency of 1/4, with lighter green indicating instructions from other hardware threads.

The timing behavior of this code segment is similar to the `DU` instruction used in Example 4.1. Since the hardware thread is already active, an instruction from the thread is interrupted in the execute stage (and not earlier for precise exceptions). Multiple interrupt handler instructions then execute instead of a single replayed `DU` instruction, increasing the latency until the I/O operation. The scheduling delay affects when an instruction is interrupted. Figure 4.2 illustrates both the minimum and maximum delay from time expiration until the first handler code instruction is in the execute stage. The p_m term is the minimum delay caused by interrupt, pipeline flush, and next fetch for each scheduling frequency.

$$
\begin{aligned}
p_m &= \begin{cases} 5 & f = 1, 1/2, 1/4 \\ 7 & f = 1/3 \end{cases} \\
t + c \times p_m \leq t_{IE} &< t + c \times (p_m + (p-1) + 1) \\
t + c \times p_m \leq t_{IE} &< t + c \times (p_m + p) \\
t_{IO} &= t_{IE} + c \times (p \times h)
\end{aligned} \tag{4.6}
$$

As with the delay until mechanism, variation only occurs from the scheduling delay and the clock cycle granularity, and t can be adjusted to compensate for the other terms.

$$
\begin{aligned}
t + c \times (p_m + p \times h) \leq t_{IO} &< t + c \times (p_m + p + p \times h) \\
t_{adj} &= t + c \times (p_m + p \times h) \\
t_{adj} \leq t_{IO} &< t_{adj} + c \times p
\end{aligned} \tag{4.7}
$$

Interrupt on expire can also be used in a periodic loop, as done with the `DU` instruction in Example 4.2. The result is the same as Equations 4.4 and 4.5 provided the handler routine takes a constant number of thread cycles.

Because the I/O mechanism only modifies I/O pins when an instruction executes, the precision is limited by hardware thread scheduling. Between any two timing instructions, the variation is bounded by $c \times 2p$. For a HRTT on a 100 MHz processor, this is 80 nanoseconds at scheduling frequency 1/4. This level of timing accuracy and precision is much higher than on a single-threaded processor running an RTOS unless there is only one task running. Higher precision requires decoupling I/O from instructions using external hardware, as described previously in Section 3.6.

4.2 Performance

Average throughputs of the overall processor and of each hardware thread are common performance metrics for processors. High throughput provides fast and efficient execution for a program since each clock cycle performs useful work. *Cycles per instruction (CPI)* and its multiplicative inverse *instructions per cycle (IPC)* are average numbers for execution over a period of time. By measuring with respect to clock cycles instead of time, these metrics support comparisons that are independent of processor clock frequency but also ignore differences in achievable processor clock frequencies.

We define and use three types of throughput metrics: processor, thread, and scheduled thread CPI. *Processor CPI*, which provides a measure of overall processor throughput, is the average number of instructions completed from *all* hardware threads per clock cycle. *Thread CPI* is the average number of instructions completed from a specific hardware thread per clock cycle and is a measure of hardware thread throughput. *Scheduled thread CPI* is the average number of instructions completed from a specific hardware thread per *scheduled* clock cycle; this measures the efficiency of a hardware thread at completing instructions with provided cycle resources.

When a hardware thread's scheduling frequency is constant and some program execution properties are known, these metrics can be directly calculated. To evaluate the performance of just the pipeline, we assume programs fit entirely within the scratchpad memories, no peripheral bus or I/O operations, and no dynamic thread scheduling changes. This section provides an equation to calculate these metrics and measures scheduled thread CPI for benchmark programs executing in different hardware thread scheduling configurations. These results provide insights into the relationships between scheduling and hardware thread throughput, and concurrency and overall throughput.

Ignoring system and timing instructions, the only instructions that take more than a single thread cycle are jump, taken branch, and memory load. The *scheduled thread CPI* for different scheduling frequencies can be calculated if average percentages of these instruction

types are known:

$$\frac{\textit{thread cycles}}{\textit{inst. (1 thread)}} = \begin{cases} 3 \times \textit{(\% jump or branch taken)} + 2 \times \textit{(\% load)} + \textit{(\% rest)} & f = 1 \\ 2 \times \textit{(\% jump or branch taken)} + \textit{(\% rest)} & f = 1/2 \\ 1 & f = 1/3+ \end{cases} \tag{4.8}$$

For example, if a program path consists of 10% jump or branch taken and 25% load instructions, the scheduled thread CPI for scheduling frequencies 1, 1/2, and 1/3 is 1.45, 1.1, and 1. When multiplied by the corresponding scheduling frequencies, the *thread CPI* can also be calculated:

$$\frac{\textit{clock cycles}}{\textit{inst. (1 thread)}} = \begin{cases} 3 \times \textit{(\% jump or branch taken)} + 2 \times \textit{(\% load)} + \textit{(\% rest)} & f = 1 \\ 4 \times \textit{(\% jump or branch taken)} + 2\textit{(\% rest)} & f = 1/2 \\ 3 & f = 1/3 \\ 4 & f = 1/4 \end{cases} \tag{4.9}$$

Using the same example, the thread CPI for scheduling frequencies 1, 1/2, and 1/3 is 1.45, 2.2, and 3. The *processor CPI* depends on the scheduled CPI of each hardware thread.

$$\frac{\textit{clock cycles}}{\textit{inst. (all threads)}} = \sum_{i=0}^{n-1} f_{T_i} \times \textit{scheduled } T_i \textit{ CPI} \tag{4.10}$$

Extending the example to multiple hardware threads all the same instruction type percentages, the overall processor CPI for 1, 2, or 3 concurrent, interleaved hardware threads is 1.45, 1.1, and 1. As expected for a fine-grained multithreaded processor, more interleaved hardware threads decreases the throughput of each hardware thread (high thread CPI) but increases the overall throughput of the processor (low processor CPI).

The percentages of each instruction type vary between programs, often with more branching in control-intensive programs and more memory accesses in data-intensive programs. To investigate this variation and verify functionality and timing behavior, we executed and measured throughput for C programs from the Mälardalen [96] benchmark suite, which was created to compare WCET analysis tools and techniques. All programs from the benchmark suite that are executable with RV-32I, fit within the scratchpad memories, and require over 1000 cycles to execute were used.

The selected programs perform different types of tasks: *bsort100* and *insertsort* are sort algorithms, *cover* has a large number of possible paths, *crc* checks data for errors, *duff* copies data, and *statemate* is generated code for an automotive controller. The RISC-V port of gcc compiled the C programs with the "−02" optimization flag. The results are shown in Figure 4.3 and demonstrate the overall throughput advantage of fine-grained multithreading:

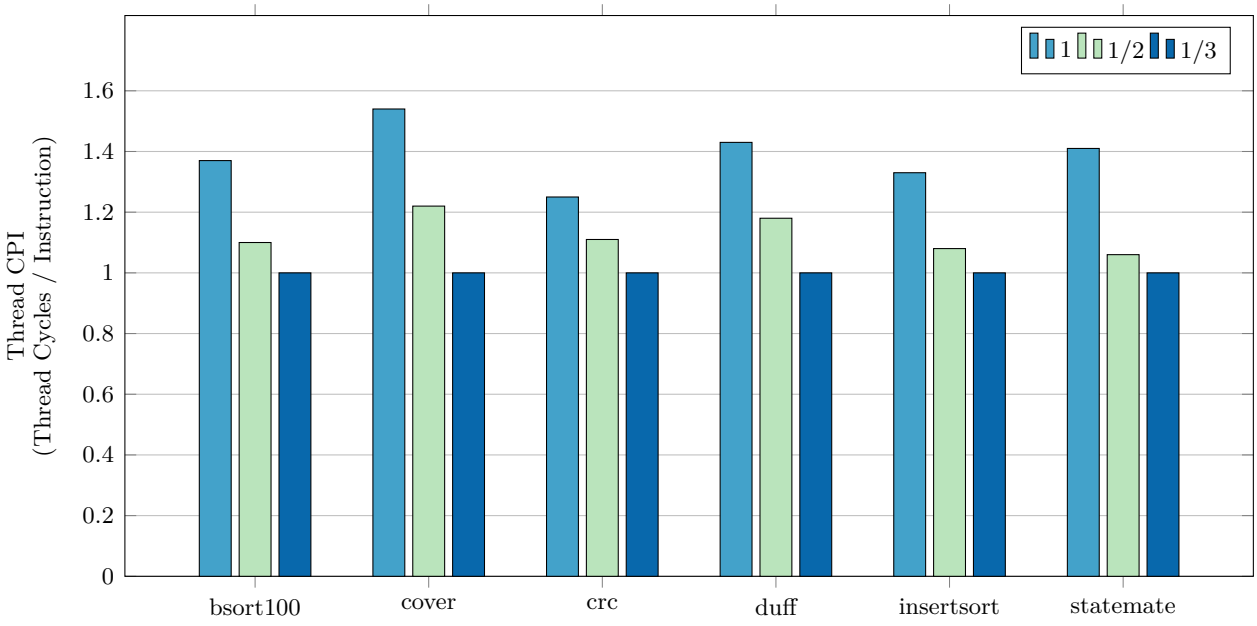


Figure 4.3: Thread throughput for different scheduling frequencies.

a scheduling frequency change from 1 to 1/2 decreases average thread CPI by 19%, and a scheduling frequency change from 1 to 1/3 decreases average thread CPI by 28%.

The overall processor throughput differences between FlexPRET and the baseline processors depend on the concurrency of the application. With only one task, the overall processor CPI of Base-1T is equal to FlexPRET with one hardware thread at scheduling frequency of 1 (every cycle). With four concurrent tasks deployable on separate hardware threads, the overall processor CPI of Base-4T-RR is equal to FlexPRET with 4 hardware threads each at scheduling frequency of 1/4.

One example relationship between concurrency and overall throughput is visualized in Figure 4.4 for both FlexPRET and the baseline processors. When concurrency of x exists, the Base-1T executes x identical tasks sequentially (one after another), FlexPRET executes x identical tasks on separate hardware threads that are interleaved round-robin, and Base-4T-RR also executes x identical tasks on separate hardware threads but always interleaves 4 hardware threads. This example assumes each task has a thread CPI equal to the average (mean) thread CPI from Figure 4.3, and the results are directly calculated instead of simulated.

When there is no concurrency, FlexPRET and Base-1T have an identical overall processor CPI, which is 1.39 in this example, while Base-4T-RR has an overall processor CPI of 4. The high processor CPI at low concurrency is the main disadvantage of Base-4T-RR's fixed thread interleaving, which FlexPRET avoids by allowing arbitrary thread interleaving. As concurrency increases, Base-1T's CPI remains constant, but FlexPRET's and Base-4T-RR's

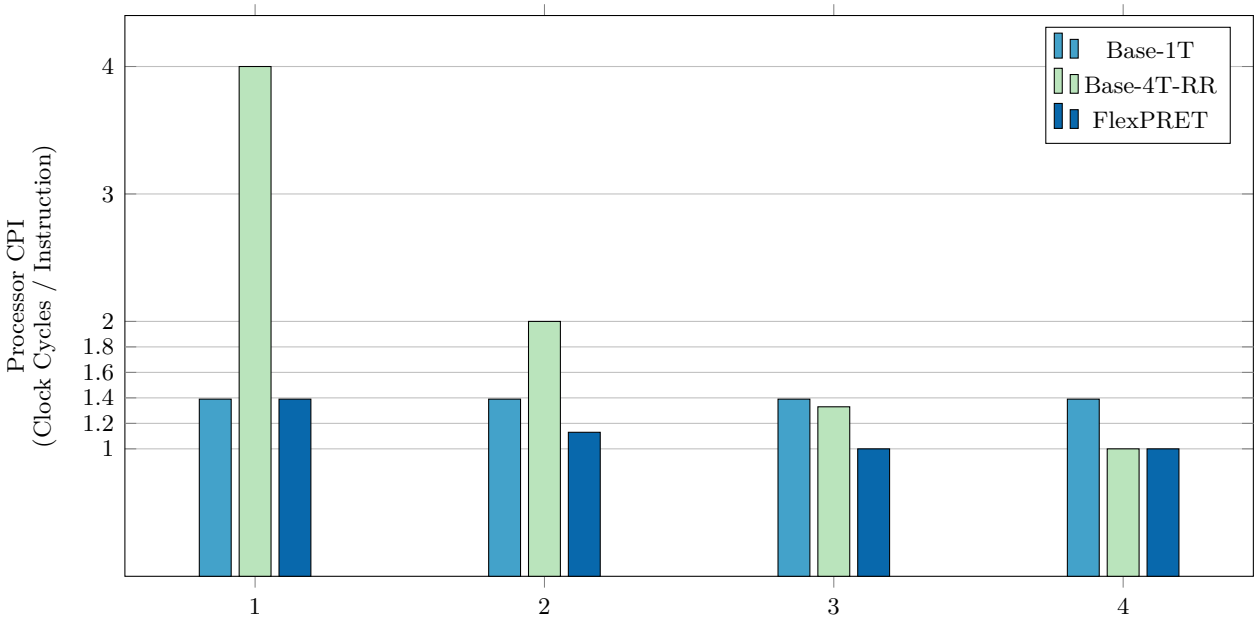


Figure 4.4: Overall processor throughput on FlexPRET and the two baseline processors for an example consisting of different numbers of independent concurrent tasks.

overall CPI improves. Base-4T-RR needs to execute at least 3 concurrent tasks for a lower CPI than Base-1T.

As a result, in addition to its predictability and isolation properties, FlexPRET provides higher overall throughput for varying concurrency than the baseline processors. A higher overall throughput does not imply a high overall efficiency, however, unless all processors operate at the same clock frequency and have the same cost. The hardware resource costs for the techniques used in FlexPRET will be evaluated in the next section.

4.3 FPGA Resources Usage

An FPGA contains configurable components, which include logic and memory, with configurable interconnect. The basic logic element in an FPGA is a *lookup table (LUT)* with some number of inputs and an output. For every possible input combination, the output is programmable, implementing combinational logic. Sequential logic requires memory, and the basic memory element in an FPGA is a *flip-flop (FF)*, which stores the input on the clock edge. *Block RAMs (BRAMs)* provide higher-density storage in an FPGA. All these elements are connected using different routing resources. Routing delays often constrain the maximum operating frequency of a design more than logic delays.

In the Xilinx Spartan-6 FPGA, 6-input LUTs and FFs are grouped into *slices*, which can also contain additional elements to improve area or speed, such as multiplexers, carry logic, and shift-registers. Composition of FPGA elements varies between manufacturer and part

family, so LUT and FF numbers are useful for general comparisons. For comparison between designs on the same hardware, such as between FlexPRET and the baseline processors, the number of slices is a useful measure of area that combines both LUTs and FFs.

Base-1T and Base-4T-RR would function correctly if FlexPRET's design is only modified with different schedulers but would have unnecessary logic and paths. For an objective comparison, we improve the design of both baseline processors to reduce area and increase maximum clock frequency. Figure 4.5 shows a high-level pipeline diagram for the three configurations.

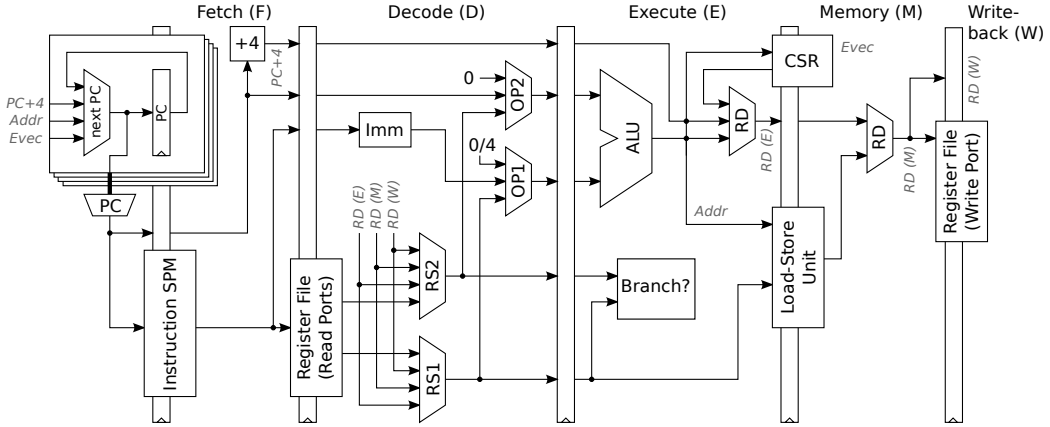
For Base-1T, the same paths and control logic execute instructions but for only a single thread. As opposed to a multithreaded pipeline that updates and selects between multiple program counters for instruction fetch every cycle, only one program counter is required. There is also only one set of CSRs, which reduces routing and multiplexing. The hardware thread scheduler does not exist, but the thread mode mechanism is used to indicate if each fetch cycle is valid. The modified pipeline is shown in Figure 4.5b.

In Base-4T-RR, fixed hardware thread scheduling simplifies datapath and control logic. Hardware thread scheduling only requires a 2-bit counter with validity based on thread mode instead of slot and thread mode CSRs generating requests for arbitration logic. When a hardware thread's current instruction is in the fetch stage, the hardware thread's previous instruction can only be in the writeback stage—with all processor state changes committed. Consequently, no bypass paths for source register values are needed from the execute, memory, or writeback stages. Also, the program counter address calculation in the execute stage can be registered to reduce timing path length.

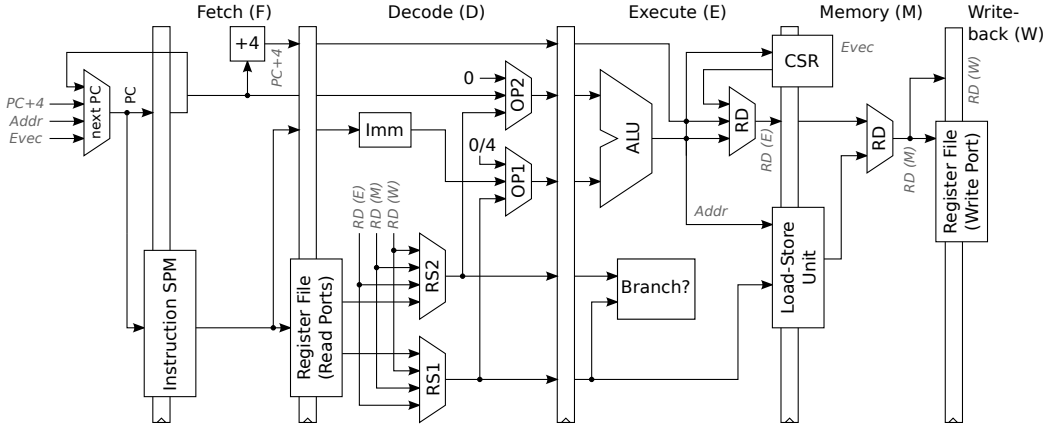
Figure 4.6a shows resource usage on a Xilinx Spartan-6 XCSLX45-3CSG324 for FlexPRET, Base-1T, and Base-4T under several configurations at a 90 MHz clock speed with a 16 KB I-SPM and 16 KB D-SPM. The suffix on the processor names indicates different configurations: *MIN* is a minimal implementation without trapping or timing instructions, *EX* has trapping added, and *TI* has both trapping and timing instructions added. The numbers were obtained by running synthesis, map, and place and route with high optimization effort for area but no manual floor planning. All versions require 16 Block RAMs for SPMs and 1 Block RAM for the register files, where each Block RAM in this device is 18 kilobits.

The resource differences between the Base-1T and Base-4T-RR configurations show the cost of fine-grained multithreading. Compared to Base-1T, Base-4T-RR does not need forwarding paths and control logic for flushing, which decreases required LUTs, but needs multiplexing for program counters and several CSRs, which increases required LUTs. The increase in LUTs is 1% for MIN, 31% for EX, and 39% for TI. Most CSRs and the program counters scale with the number of hardware threads, which causes the increase in FFs: 35% for MIN, 92% for EX, and 99% for TI. For this FPGA, the increase in FFs is less relevant than the increase in LUTs because FFs are underutilized in slices—the increase in slices is 22% for MIN, 39% for EX, and 57% for TI.

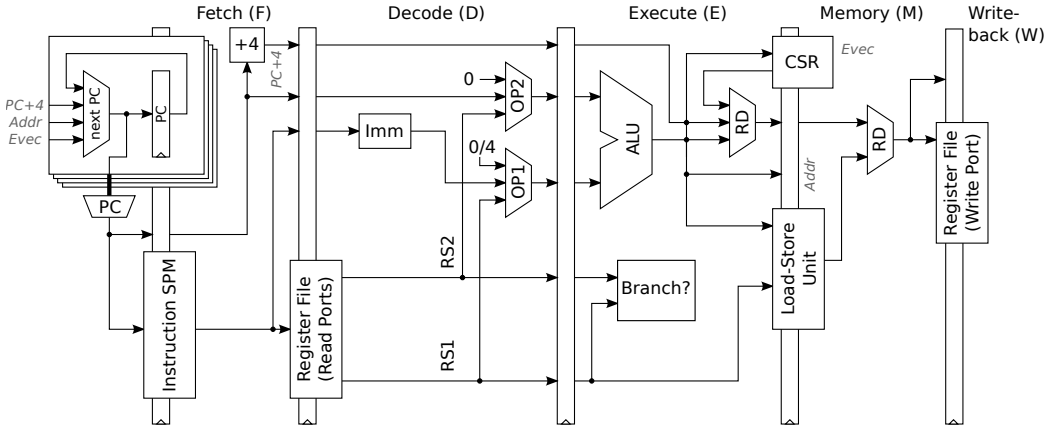
The relative cost of fine-grained multithreading is higher with trapping and timing instructions because of mechanisms that scale with number of hardware threads. For each hardware thread, trapping stores the cause, program counter, and address of handler rou-



(a) FlexPRET

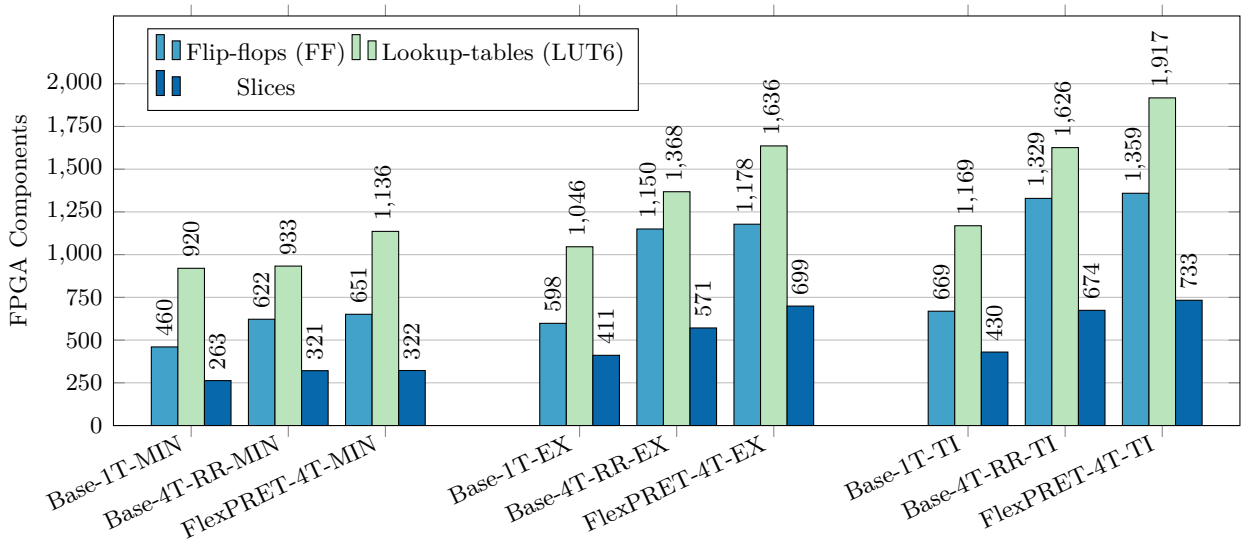


(b) Base-1T

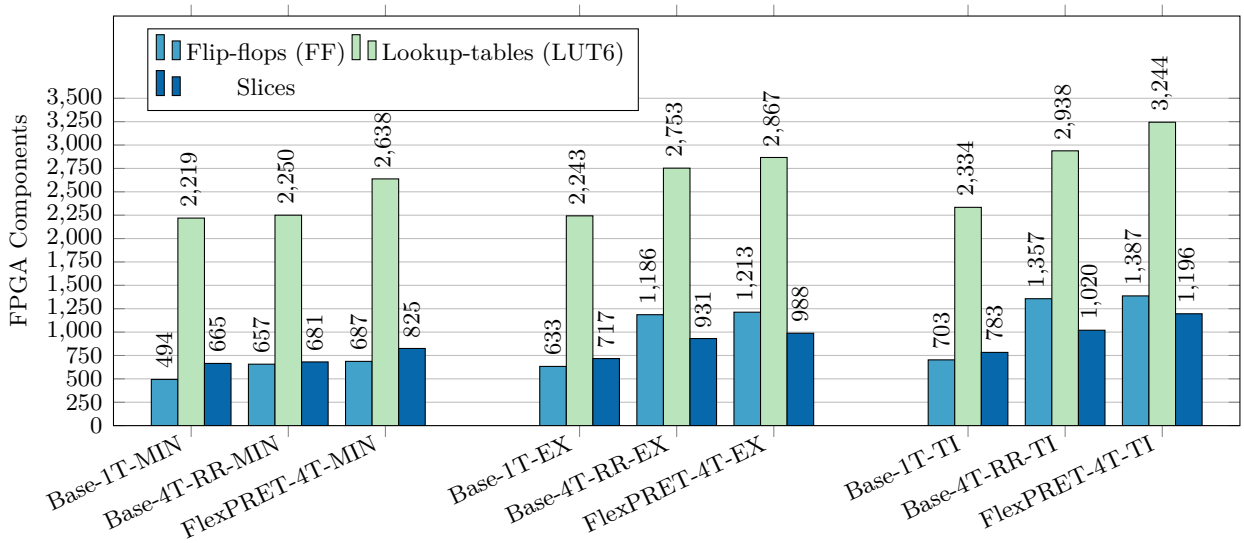


(c) Base-4T-RR

Figure 4.5: High-level datapath diagrams for FlexPRET and the baseline processors.



(a) Base ISA (RV32I)



(b) Base ISA (RV32I) extended with 2-cycle hardware multiplier implemented with lookup-tables

Figure 4.6: FPGA resource usage by FlexPRET and two baseline processors for three configurations: without trapping or timing instructions (MIN), with trapping but no timing instructions (EX), and with trapping and timing instructions (TI).

tine. For each hardware thread, timing instructions require a comparator between the shared clock and the thread's compare value in a CSR. Although these mechanisms increase resource usage, the Base-1T cannot independently and concurrently handle 4 external interrupts or timing operations—similar functionality requires 4 Base-1T processors.

The resource differences between the Base-4T-RR and FlexPRET configurations show the cost of flexible thread scheduling. The scheduling logic, forwarding paths, and control logic for flushing increase LUTs, and additional CSRs for the scheduler increase FFs. Compared to Base-4T-RR, the increase in LUTs for FlexPRET is 22% for MIN, 20% for EX, and 18% for TI; the increase in FFs is under 5%; and the increase in slices is under 1% for MIN, 22% for EX, 9% and TI. The overhead of less than 10% for the TI configuration justifies adding FlexPRET's thread scheduler, or a variation, to any fine-grained multithreaded processor; if necessary, the hardware thread scheduler can emulate existing scheduling approaches. In addition to the isolation of HRTTs and cycle sharing of SRTTs, overall processor throughput is higher whenever less than 4 hardware threads are active.

With more independent tasks than hardware threads, a collection of Base-1T cores provides a higher overall throughput per area than FlexPRET when the area costs are higher than the CPI benefits of fine-grained multithreading. For an isolation-bound task set, however, each FlexPRET processor can meet requirements that would traditionally require several (i.e. up to 4) Base-1T cores—a substantial savings. Furthermore, the percentage increases are based on a bare-minimum processor implementation and would decrease with more shared functionality added to the processor, such as a floating-point unit or peripherals. For example, Figure 4.6b shows the FPGA resource usage with an 80 MHz processor clock when a 2-cycle LUT-based multiplier is added to the configurations of Figure 4.6a.

4.4 Power and Energy Implications

Embedded devices require low power (energy consumed per unit time) for less heat dissipation and energy efficiency for longer battery life. Power and energy efficiency depend on both the microarchitecture and its implementation on a specific device process technology. This section focuses on only the high-level energy efficiency implications of the FlexPRET microarchitecture in comparison to the baseline processors, without any measurements. We define energy efficiency the energy required to perform a sequence of instructions, which is measured as average energy per instruction. Some of the techniques considered require additional circuitry on ASICs and cannot be done on current FPGAs.

There are two main sources of energy usage in a circuit: *dynamic* from transistor switching activity and *static* from transistor leakage. Dynamic energy is used every clock cycle to change the values at logic gates by charging or discharging associated capacitance. The energy used for a sequence of instructions can be calculated from the supply voltage V_{dd}, total circuit capacitance C_{total}, and switching activity factor α.

$$E_{dynamic} = \alpha \, C_{total} \, {V_{dd}}^2 \;\propto\; {V_{dd}}^2 \tag{4.11}$$

The switching activity varies every cycle, as it depends on the logical changes at every gate in the circuit, but an estimated factor can be used for a sequence of instructions. At a lower supply voltage, the maximum achievable clock frequency is also lower. For a fixed supply voltage, reducing clock frequency does not reduce energy efficiency, but it does reduce dynamic power because of less switching per unit time. However, reducing supply voltage quadratically reduces dynamic energy; this reduction is limited by clock frequency requirements or faulty operation at lower supply voltages. For minimal dynamic energy usage, the supply voltage should be as low as possible provided the corresponding clock frequency meets the real-time requirements of the application.

Static energy is used whenever the circuit is powered because transistors in gates do not turn completely off, allowing current to leak. To save energy, a design can power off parts of a circuit when not needed, but power up latencies and control logic overhead limit frequency and granularity of power toggling. At a high level, the energy usage for a sequence of instructions that has an execution time of t at supply voltage V_{dd} is

$$E_{static} = I_{leakage} \times t \times V_{dd} \tag{4.12}$$

where $I_{leakage}$ is the total leakage current for the powered transistors and scales with V_{dd}. The highest clock frequency for a given supply voltage will minimize t. Maximum clock frequency increases then saturates with higher supply voltages, and the minimal static energy point depends on this relationship. A high clock frequency with a low supply voltage allows the circuit to power down for longer periods of time. The supply voltage and clock frequency that minimizes the sum of dynamic and static energy for a sequence of instructions depends on all these parameters. To meet an application's timing requirements, the processor may also require a non-optimal operating point.

Low-level circuit design techniques affect the energy efficiency trade-offs for a particular design. *Clock gating* disables the clock signal to certain gates, which prevents them from switching and using dynamic energy. The *threshold voltage (V_t)* of a transistor affects its behavior: higher V_t transistors switch faster but also has a higher leakage current. In a *multiple threshold voltage (V_t)* process, higher leakage transistors are only used for gates that are on long timing paths, reducing overall static energy. These techniques are independent of the microarchitecture but affect the optimal operating point for energy efficiency. Within reasonable bounds, dynamic energy is minimized by a low supply voltage and finishing "just in time", but static energy is minimized by a high clock frequency to "race to finish" and power off. Historically, dynamic energy dominates overall energy usage until fairly low supply voltages.

For our high-level comparison, we assume the processors have a low-power standby state and can function at multiple discrete operating frequencies. The optimal energy efficiency occurs at some supply voltage and associated clock frequency, and higher suboptimal supply voltages and associated clock frequencies are used to meet timing constraints. If an application's WCET requires an operating frequency lower than the optimal frequency, the processor operates at the optimal frequency and goes into standby. If the required operating frequency is higher than the optimal frequency, however, there are several options. The

simplest method is a static selection of the lowest operating frequency that meets the application's WCET requirements. If an application's ACET is much less than the WCET, however, dynamically changing the operating frequency can further improve energy efficiency.

In both cases, a tighter bound on WCET improves energy efficiency. In the static case, a loose WCET bound requires operating at a higher, less efficient clock frequency than is necessary to meet the deadline. In the dynamic case, *dynamic voltage and frequency scaling (DVFS)* starts executing the program at the optimal operating frequency and only increases to suboptimal frequencies at program checkpoints for longer program paths (e.g. WCET path). For DVFS, a loose WCET bound requires frequency to increase faster than necessary, reducing energy efficiency.

Predictability enables tighter bounds on WCET, so FlexPRET should be a good candidate for these high-level approaches for energy efficiency. Furthermore, with enough concurrency, high processor throughput reduces cycles needed to complete a task. The baseline processors are also predictable, but throughput for varying concurrency is not as high. Base-1T requires more cycles to execute an application because latency is not hidden by other hardware threads. To match the throughput of FlexPRET, Base-4T-RR would need 4 hardware threads to always be all active or all not active.

Multithreading is both a benefit and a hindrance—it enables high throughout with predictability, but balancing hardware thread scheduling for energy efficiency is more difficult than in software. Software and hardware thread scheduling of tasks on FlexPRET for energy efficiency is a challenging future direction. The processor can only go into standby state if all hardware threads are sleeping, and there is some flexibility for trading-off the logical frequency of each hardware thread, provided full isolation is not required.

Chapter 5

Application to Real-Time Systems

This chapter demonstrates FlexPRET programming methodologies for two types of real-time applications: mixed-criticality systems and precision-timed input/output (I/O). FlexPRET supports existing techniques for implementing these applications by configuring hardware thread scheduling for single-threaded or interleaved multithreaded operation, but the example applications in this chapter demonstrate the benefits of more flexible thread scheduling. Also, the integration of timer functionality with the processor through timing instructions simplifies the specification of precise timing behavior.

Mixed-criticality systems consist of tasks with different verification requirements and timing constraints. The challenge is providing a different level of assurance for each task while maintaining efficient resource utilization. FlexPRET addresses this challenge by using separate hardware threads, each with allocated scheduling resources (Section 5.1).

Hardware peripherals support cycle-accurate I/O operations but perform a dedicated operation. Additionally, the peripheral must both exist and be available on the selected hardware platform. When using software for multiple independent I/O tasks, conflicts over execution priority in a single-threaded processor reduce timing accuracy and precision. With FlexPRET, hardware threads have no context switch overhead, and the precision of timing instructions is limited by how frequently the thread is scheduled (Section 5.2).

5.1 Mixed-Criticality Systems

An increasing trend in cyber-physical and real-time embedded systems is integrating tasks with different levels of criticality onto a single hardware platform—resulting in a *mixed-criticality system*. *Criticality* represents the required level of assurance against failure for a task or component. For application domains such as automotive and avionics, each criticality level has different certification requirements, which specify or restrict design and verification methodologies. The number and definition of criticality levels vary by standard, but the minimum is two levels, often defined as safety-critical and non-critical.

The trend is driven by the transition from federated to integrated architectures in both the avionics and automotive domains to reduce costs, even with increasing system complexity. Both improvements in application functionality and advances in hardware platforms increase system complexity. As in general-purpose systems, embedded system components are migrating from a single core processor to either a homogeneous or heterogeneous multicore system-on-a-chip (SoC) for higher performance and reduced power consumption. Tasks share a hardware platform by partitioning processor time, memory, networks, and other shared resources. There are advantages and disadvantages to a highly integrated hardware platform: there is more flexibility at both design and run time to reallocate spare resources, but system design and verification must account for the contention and interference caused by this resource sharing.

In avionics, the Integrated Modular Avionics (IMA) concept is to integrate multiple tasks with modular interfaces onto the same computing platform, reducing weight, power, and development costs [97]. With a well-defined common interface between software and hardware, spare resources are more usable, and functionality upgrades are possible without a complete system redesign. The ARINC 653 software standard [39] defines how a real-time operating system (RTOS) partitions space and time for applications with different criticality levels. To specify these levels, the DO-178C standard [13] defines five design assurance levels (DALs) with different definitions and requirements: (A) Catastrophic, (B) Hazardous, (C) Major, (D) Minor, and (E) No Safety Effect.

Specifications of criticality levels focus on the consequences of failures and requirements for assuring safe behavior. Meeting timing deadlines is just one possible requirement—tasks function unsafely in other ways. Nevertheless, we investigate the timing-related properties of FlexPRET for mixed-criticality systems. We assume high-criticality tasks are classified as hard real-time (must meet deadline), and low-criticality tasks are classified as soft real-time (reduced utility with deadline miss). This assumption does not hold for all applications, but a task of any criticality level could use a different real-time classification.

In this section, we describe a methodology for deploying mixed-criticality applications on FlexPRET (Section 5.1.1) and demonstrate this methodology with a simple example and a more complex case study (Section 5.1.2). Previous work in mixed-criticality has focused on software scheduling for single-threaded processors and relies on the assumption that each task has a different WCET bound for each criticality level [11] (Section 5.1.3). By supporting hardware-based isolation, FlexPRET provides higher confidence in timing behavior for certain tasks without any assumptions about differing WCET bounds.

5.1.1 Methodology

Partitioning a hardware platform in both space and time is an essential technique used in mixed-criticality systems. Privatizing memory segments and I/O devices are examples of space partitioning, and allocating segments of time for processor or shared resource usage are examples of time partitioning. Isolated partitions provide safety assurances but limit resource sharing for efficient utilization—a fundamental trade-off for mixed criticality sys-

tems. FlexPRET is designed to provide greater flexibility for this trade-off on an integrated platform.

An integrated architecture on a single-threaded processor requires software-based partitioning. As discussed in Chapter 1, partitioning can be either static and reservation-based or dynamic and priority-based. For safety guarantees, the RTOS must enforce the partitioning by monitoring execution times and protecting memory segments; if a non-critical task is responsible for not exceeding its execution-time budget and returning control to the scheduler, verification of that task must be at the same level of assurance as the most critical task.

Hardware-based partitioning is robust because it does not rely on software for isolation but has traditionally been provided by a federated architecture. Multicore processors support an integrated architecture with each core providing hardware-based isolation for a single task, although some resources such as the memory bus are shared between cores. Fine-grained multithreading supports a finer granularity of hardware-based isolation at the level of hardware threads. Therefore, FlexPRET supports more partitions isolated by hardware than a single-threaded core while still allowing direct sharing of spare resources.

Our proposed methodology for deploying mixed-criticality applications onto the FlexPRET processor uses the combination of hardware-based and software-based partitioning. We suggest the following:

1. *Assign only tasks of the same criticality level to each hardware thread, with fewer tasks per hardware thread at higher criticality levels.*
   The software testing and verification requirements vary by criticality level. With only one criticality level present on a virtual real-time machine (hardware thread), there are no conflicting requirements. An RTOS on each thread uses a traditional real-time scheduling algorithm, as the mixed-criticality aspect of the application is partitioned away. If the scheduling algorithm does not reallocate spare resources to another task on the same hardware thread, the hardware thread scheduler reallocates them to another hardware thread. This guideline of a single criticality level per thread can be relaxed if required to improve schedulability or efficiency, but the RTOS would use a mixed-criticality scheduling algorithm.

2. *For hardware threads with higher-criticality tasks, use hard real-time threads (HRTTs) and over-allocate scheduling resources.*
   With isolated scheduling and constant instruction latencies, HRTTs are predictable for static worst-case execution time (WCET) analysis, resulting in safe and tight WCET bounds. This high confidence in timing behavior is beneficial for guaranteeing safe functionality of higher-criticality tasks. The allocated scheduling frequency depends on the requirements of the other threads and has limited options (e.g. $f = 1/1, 1/2, 1/3, \dots$). HRTTs must be over-allocated scheduling resources, but SRTTs can use any spare cycles.

3. *For hardware threads with lower-criticality tasks, use soft real-time threads (SRTTs) and under-allocate scheduling resources.*
   With only a minimum scheduling frequency and bounded instruction latencies, static WCET analysis is likely to provide looser WCET bounds for a SRTT because there is no guarantee of spare scheduling resources. By considering the run-time behavior of other hardware threads, as is done in mixed-criticality schedulability analysis, tighter WCET estimates are obtainable. Higher overall efficiency is the rationale for under-allocating scheduling resources: if all tasks meet deadlines with allocated scheduling resources, spare cycles are not needed to meet deadlines and only improve response times.

4. *If multiple tasks are mapped to a single hardware thread, use the highest confidence scheduling algorithm that meets all deadlines; prefer non-preemptive over preemptive and static over dynamic for higher-criticality tasks.*
   Scheduling algorithm selection depends on task set properties, criticality level requirements, and implementation overhead. Ignoring overhead, the selected scheduling algorithm affects when spare cycles occur but not the number of them. The dependence of instruction latency on processor state is a primary motivator for disallowing preemption, which modifies processor state at unpredictable program points. This is not a problem in FlexPRET, but disallowing preemption still has other benefits, such as reduced load on SPM management with less task switching. Static scheduling algorithms are preferrable for higher-criticality tasks if software overhead and complexity is lower and time segments of cycle sharing are more predictable. For lower-criticality hardware threads, the scheduling algorithm should be more aggressive to efficiency use spare cycles.

These suggestions are not requirements for deploying mixed-criticality applications on FlexPRET but take advantage of the processor's unique properties. Due to the limited number of hardware threads and possible scheduling combinations, some hardware threads may execute mixed-criticality task sets. FlexPRET can always operate in single-threaded mode, but concurrent threads provide hardware-based isolation and improved overall processor throughput.

5.1.2 Evaluation

We use a simple mixed-criticality task set example to demonstrate the isolation and efficient resource utilization properties of FlexPRET. To demonstrate the task mapping and thread scheduling methodology, we simulate a more complex task set from the avionics domain that has more tasks than hardware threads.

Simple Mixed-Criticality Example

The simple mixed-criticality example consists of four periodic tasks, τ_A, τ_B, τ_C, and τ_D, simulated on a FlexPRET processor with 4 hardware threads, 32 KB I-SPM, 32 KB D-SPM, and a hardware multiplier, a reasonable configuration for a low-cost FPGA soft-core or ASIC. The task identifiers A, B, C, and D represent the criticality level of each task from highest to lowest, similar to design assurance levels (DALs) in the avionics domain. Tasks τ_A and τ_B have hard real-time requirements and are on separate HRTTs, and tasks τ_C and τ_D have soft real-time requirements and are on separate SRTTs.

Figure 5.1a shows each task's thread ID, initial thread mode, period (T_i), relative deadline (D_i), and worst-case thread cycles for each scheduling frequency f ($E_{i,1}, E_{i,1/2}, E_{i,1/3+}$). The thread cycle values are converted to worst-case execution time C_i using processor clock frequency f_p, where $C_i = E_{i,f}/(f_p \times f)$. For example at clock frequency 100 MHz, if τ_A executes every processor cycle ($f = 1$), it would take $4 * 10^5/(100 * 10^6\ Hz) = 4\ ms$ to complete, and if it instead executes every other processor cycle ($f = \frac{1}{2}$), it would take $3.65 * 10^5/(100 * 10^6\ Hz * \frac{1}{2}) = 7.3\ ms$ to complete.

In FlexPRET, the number of thread cycles to execute a path in the task depends on scheduling frequency, as lower scheduling frequencies hide instruction latencies, such as taken branches. As examined in Section 4.2, the variation depends on the percentages of different instruction types. To account for the throughput differences, each task iterates a program from the Mälardalen benchmark suite to reach the values of $E_{i,1}$ specified in Figure 5.1a, which are contrived to highlight properties of FlexPRET. The same number of iterations is performed at the other scheduling frequencies to measure values for $E_{i,1/2}$ and $E_{i,1/3+}$. 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). Each iteration takes the same (assumed) worst-case program path, and the `delay_until` mechanism performs periodic task release with spare cycle sharing once completed.

Figure 5.1b shows the selected scheduling configuration for this task set, but other configurations have possibly lower response times. Based on worst-case thread cycles and period requirements, task τ_A requires a scheduling frequency of at least 1/3 to meet its deadline, task τ_B does not require many cycles, task τ_C requires almost half of the processor cycles to meet its deadline, and task τ_D does not require many cycles. Therefore, slots are assigned such that τ_A and τ_B are over-allocated cycles with scheduling frequencies of 1/3 and 1/6; higher scheduling frequencies would reduce thread interleaving once these tasks complete.

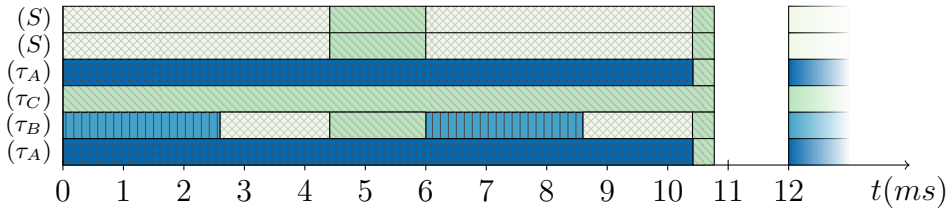
Tasks τ_C and τ_D are SRTTs and use the allocated cycles of tasks τ_A and τ_B once they complete. Even though these SRTTs have quite different execution cycles requirements, by default the hardware thread scheduler alternates between these two threads, completing task τ_D much earlier than task τ_C. As overall efficiency is higher when more threads are interleaved, one of the scheduling slots is assigned to task τ_C, which causes it to execute at a faster rate than task τ_D. Two slots are disabled and two automatically delegate to SRTTs to achieve the desired scheduling frequencies of the HRTTs.

Task	Thread ID	Thread Mode (Init)	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

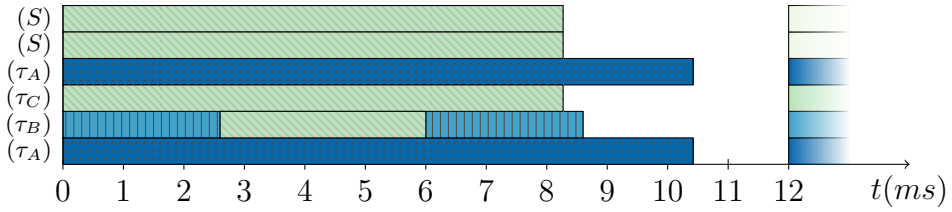
(a) The task set.

D	D	S	S	T_0 (τ_A)	T_2 (τ_C)	T_1 (τ_B)	T_0 (τ_A)

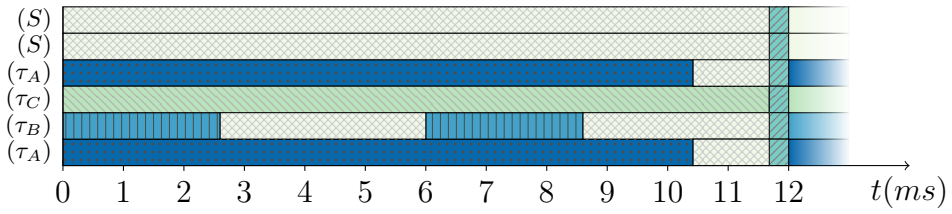
(b) The slots CSR.



(c) FlexPRET-4T in normal operation.



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



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

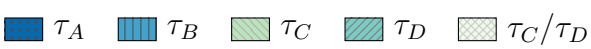


Figure 5.1: The task set and scheduler properties for a simple mixed-criticality example simulated on FlexPRET-4T under various conditions.

Figures 5.1c, 5.1d, and 5.1e show the simulated execution trace over a single hyperperiod, which is the least common multiple of the task periods, for different scenarios. The vertical axis is one iteration through the 6 active slots and shows which hardware thread uses each slot over time; the values in parenthesis along the axis indicate the allocated slot value. This format better visualizes the thread scheduling than cycle-by-cycle scheduling over time.

Figure 5.1c is an execution trace for normal operation where every task executes the worst-case program path. Tasks τ_A and τ_B complete before the respective deadlines, after which their allocated slots are used by SRTTs, which means tasks τ_C and τ_D. Whenever task τ_C completes, task τ_D uses all spare cycles; at time 5 ms task τ_D executes for 4 of every 6 cycles, and at time 10.5 ms, it executes every cycle. The timing behavior of tasks τ_A and τ_B is isolated and predictable, and task τ_C efficiently uses spare cycles to meet its deadline, even though it was under-allocated scheduling resources in the slots CSR.

This task set is not schedulable on Base-4T-RR with fixed round-robin scheduling because tasks τ_A and τ_C each require more than 1/4 of the processor cycles to meet the respective deadlines, which cannot occur with a fixed interleaving over 4 hardware threads. Also, even if all tasks could meet the respective deadlines with a scheduling frequency of $f = 1/4$, tasks τ_C and τ_D cannot use spare cycles. With active round-robin scheduling, all tasks meet deadlines but the timing behavior of each task is not isolated. Base-1T would need to use software scheduling to isolate tasks τ_A and τ_B, and it would require more execution cycles—all tasks would complete at time 11.8 ms instead of at time 10.8 ms in this example.

To demonstrate the isolation of HRTTs, Figure 5.1d shows an error case where τ_D completes immediately, and Figure 5.1e shows an error case where τ_D executes infinitely. For both cases, the timing behaviors of the most critical tasks τ_A and τ_B are identical. When task τ_D ends immediately, task τ_C executes more frequently and completes earlier than normal; when task τ_D never ends, τ_C executes less frequently and barely meets its deadline.

Mixed-Criticality Avionics Case Study

Deployment on FlexPRET of a mixed-criticality task set with more tasks than hardware threads requires a task mapping and thread scheduling methodology. We demonstrate our proposed methodology on an abstract task set from Vestal's influential paper on mixed-criticality scheduling [10]. The task set contains 21 tasks of varying criticality levels derived from a time-partitioned avionics system at Honeywell, where A is the highest assurance level and D is the lowest. To remove the need and effects of scratchpad memory management, the task set was simulated on a FlexPRET processor with 8 hardware threads, a hardware multiplier, and an unpractical 128 KB for the I-SPM and D-SPM.

As with the simple mixed-criticality example, Figure 5.2a shows each task's thread ID, initial thread mode, period (T_i), relative deadline (D_i), and worst-case thread cycles for each scheduling frequency f ($E_{i,1}, E_{i,1/2}, E_{i,1/3+}$). The overall processor utilization with all threads executing in single-threaded mode is 93%. Again, A and B level tasks iterate the *statemate* program, and C and D level tasks iterate either *jfdctint* or *insertsort*, to account

Task	Thread ID	Thread Mode (Init)	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.

T_5	T_3	T_4	T_2	T_5	T_1	T_4	T_0

(b) The slots CSR.

Figure 5.2: The task set and scheduler properties for a mixed-criticality avionics case study simulated on FlexPRET-8T.

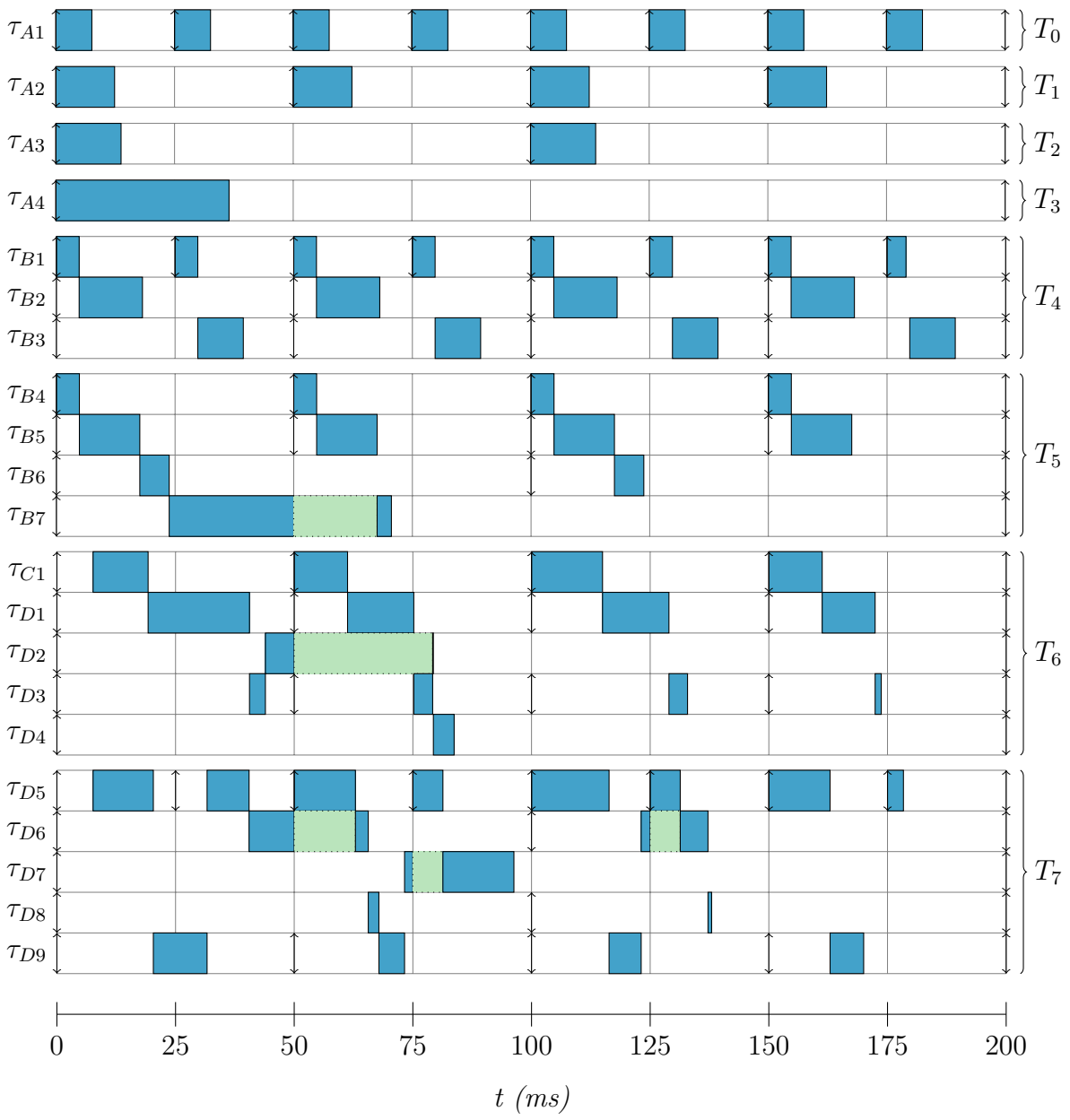


Figure 5.3: Execution trace for a mixed-criticality avionics case study simulated on FlexPRET-8T.

for thread latency dependence on scheduling. The `delay_until` mechanism shares cycles once a task completes until the next periodic release.

To follow (1), the most critical tasks $\tau_{A1} - \tau_{A3}$ each execute on separate hardware threads, isolated from all other tasks. There are not enough hardware threads for all B level tasks to use separate threads, so tasks $\tau_{B1} - \tau_{B2}$ share one thread, and tasks $\tau_{B4} - \tau_{B7}$ share another thread, isolated from tasks of other criticality levels. Even though tasks $\tau_{C1}, \tau_{D1} - \tau_{D9}$ could be mapped to one thread, overall efficiency is higher when more threads are interleaved, and the tasks are split between the remaining threads.

Figure 5.2b shows the thread scheduling configuration in the slots CSR. Each hardware thread is similar to a partition used by reservation-based scheduling in commercial RTOSs for mixed-criticality systems [40], but with isolation for HRTTs guaranteed by hardware. To follow (2), scheduling resources are over-allocated for the threads containing A and B level tasks, and these hardware threads are classified as HRTTs for predictability and isolation. Based on worst-case execution cycles, the scheduling frequency of 1/8 is sufficient for the four threads executing tasks $\tau_{A1} - \tau_{A3}$, and the scheduling frequency of 1/4 is sufficient for the two threads executing tasks $\tau_{B1} - \tau_{B7}$. To follow (3), the threads for the C and D level tasks are classified as SRTTs; there are no slots left to allocate, so these threads only execute with spare cycles.

Different scheduling algorithms are used for each hardware thread, following (4). Tasks $\tau_{A1} - \tau_{A3}$ do not need a scheduling algorithm, and tasks $\tau_{B1} - \tau_{B3}$ are able to use a non-preemptive static schedule to meet their respective deadlines. Tasks $\tau_{B4} - \tau_{B7}$ use preemptive rate-monotonic software scheduling to meet their respective deadlines because using a non-preemptive scheduler would require splitting a task for schedulability.

The two SRTTs use an earliest-deadline-first (EDF) scheduler because it is optimal with respect to feasibility for the task sets on each thread and straightforward to implement. The `interrupt_on_expire` mechanism releases tasks into the priority queue sorted by deadline, and the EDF scheduling algorithm executes whenever the queue is modified, either by task release or completion, and may preempt another task.

Figure 5.3 shows execution traces for a single hyperperiod of the task set ($t = 200\ ms$). Each hardware thread (label on right) has a subplot, with darker blue rectangles indicating the respective task (label on left) is executing and lighter green rectangles with a dotted line indicating its preempted. The up and down arrows indicate the deadline time of a task's job is equal to the release time of its next job (relative deadline is equal to the period). In this example trace, every job follows the worst-case program path.

For tasks $\tau_{A1} - \tau_{A4}$ and $\tau_{B1} - \tau_{B7}$, HRTTs are isolated so every job of each task takes the same amount of execution time to complete—demonstrating predictability and isolation for these critical tasks. For tasks τ_{C1} and $\tau_{D1} - \tau_{D9}$, however, the execution time of each task job varies. These SRTTs rely on spare cycles from the HRTTs, so execution time depends on the execution behavior of the HRTTs. When all the HRTTs are executing, the less critical tasks are not executing (e.g. at time 0 ms), and when fewer HRTTs are executing, the less critical tasks execute more frequently (e.g. at time 74 ms).

In this execution trace, all tasks meet their deadlines. With hardware-based isolation and simple scheduling algorithms, HRTTs support WCET analysis and high confidence in correct and repeatable timing behavior. Analysis of SRTTs requires accounting for HRTT execution behavior, but less critical tasks can rely more on testing. For a multithreaded processor without FlexPRET's thread scheduling technique, such as fixed or active round-robin scheduling, the tasks would need to be distributed among hardware threads in a balanced manner to meet deadlines, with less separation of criticality levels. For this task set, a single-threaded processor could use a mixed-criticality scheduling algorithm, but would have lower overall processor throughput and only software-based isolation enforced by preemption.

5.1.3 Related Work

Researchers have proposed approaches for designing and verifying mixed-criticality systems at both the software and hardware level. At the software level, real-time scheduling theory (Section 2.1) is extended by adding criticality levels, which affects the properties of existing scheduling algorithms. At the hardware level, either software and its analysis are adapted for mixed-criticality with COTS processors or architectures are modified.

Burns and Davis [11] performed a comprehensive survey of real-time software scheduling techniques for mixed-criticality systems. The main assumption used by this community is each task's WCET is dependent on criticality level, with a higher WCET for a higher criticality level. Extending task models with this property improves schedulability of mixed-criticality task sets; an unschedulable task set using the highest-criticality WCET values can be schedulable when evaluation occurs at each task's criticality level.

Reservation-based and priority-based scheduling, introduced in Chapter 1, are two approaches for scheduling mixed-criticality task sets on a single-threaded processor [38]. An example of reservation-based scheduling is an RTOS partitioning the processor according to the ARINC 653 standard for integrated modular avionics (IMA) [98]. Critical tasks are guaranteed resources and isolated using partitions, and some RTOSs reallocate spare segments of partitions to other tasks, such as WindRiver's VxWorks 653 [40].

Vestal [10] first proposed using priority-based preemptive scheduling for mixed-criticality task sets. Later work has addressed sporadic task sets [99], WCET overrun [100], and multicore processors [101]. An application deployed on FlexPRET would only use these scheduling algorithms if tasks of different criticality levels are on the same hardware thread.

These scheduling algorithms require run-time monitoring of execution times—an overrun of a low-criticality WCET bound cannot cause a high-criticality task to miss its deadline. Lemerre et al. [102] proposed some principles and mechanisms for a microkernel to correctly execute a mixed-criticality task set with temporal isolation for CPU time, spatial isolation for memory segments, deterministic communication, and fault recovery mechanisms.

Herman et al [103] considered RTOS overhead in scheduling mixed-criticality task sets on a multicore processor and evaluated an experimental implementation. Their framework supports 4 criticality levels, represented by A, B, C, and D from highest to lowest criticality, with execution priority on each core from highest to lowest criticality. The A and B level

tasks are each mapped to a specific core, and the jobs of the C and D level tasks can be scheduled to any core. The A level tasks are statically scheduled with the highest priority, the B and C level tasks execute according to earliest-deadline-first (EDF), and the D level tasks are best effort. FlexPRET uses a similar methodology but with hardware threads instead of cores, isolation for B level tasks, and no migration across threads for C and D level tasks.

Hardware approaches provide isolation or reduced interference between tasks of different criticality levels. The multicore processor used in the MERASA project supports mixed-criticality applications with one isolated HRTT per core, whereas FlexPRET supports multiple isolated HRTTs per core. FlexPRET and other predictable processors require isolation for mixed-criticality tasks that use shared resources external to the processor, such as a network-on-chip, memory bus, or DRAM controller. Goossens et al. [17] presented an approach for isolation that uses reservation-based temporal partitioning to create virtual resources for all shared physical resources.

5.2 Real-Time Unit for Precision-Timed I/O

Cyber-physical and real-time embedded systems interact with the physical world using sensors and actuators that perform *precision-timed I/O* operations because of timing constraints. A hardware platform performs precision-timed I/O with either dedicated hardware peripherals, a combination of hardware and software, or almost entirely with software. Unless the application is high-volume or high-cost, the hardware platform is not a completely customized design and instead selected from existing commercial-off-the-shelf (COTS) options to meet the requirements of the application. Therefore, flexibility to support different applications can justify using less efficient I/O mechanisms.

For a well-defined, constrained I/O operation, a dedicated hardware peripheral is usually the most efficient in area and power usage. Some low-latency or high-throughput I/O operations require custom circuitry, as the demands cannot be met with software on processors. A communication bus allows an application executing on a processor to access multiple hardware peripherals, transferring data or controlling operation.

For greater flexibility, software can perform an I/O operation with varying degrees of hardware support. The advantage of software is functionality can be modified offline or during run-time—such as either fixing bugs, reconfiguring protocols, or completely changing protocols. Requiring the least hardware support, *bit banging* uses software to directly control the values on I/O pins; the base FlexPRET implementation uses bit banging for I/O operations.

With additional hardware support, I/O operations require fewer instructions and less software interaction, reducing processor load. The XMOS processor [44] exemplifies this technique by providing ports that perform low-level commands on I/O pins. For example, ports can read or write I/O pins at a specific clock cycle, wait for specific input values

then timestamp, or perform serialization and deserialization. Examples of developed XMOS software peripherals are I2C, SPI, UART, PWM, CAN, AVB, and Ethernet MAC.

Using software to control multiple I/O operations is difficult on a single-threaded processor, particularly if it is designed for high performance using prediction mechanisms. If an I/O task does not have the highest priority, it could be blocked and not function correctly. If an I/O task does have the highest priority, it could frequently interrupt the execution of other tasks, incurring context switch overhead and complicating WCET analysis by constantly modifying processor state. On a single-threaded processor, only one I/O task has the highest priority at every point in time.

To avoid the overhead and complexity of using the main application processor for I/O operations, these operations can be offloaded to a smaller, specialized co-processor or *real-time unit*. This is not a new concept: the CDC 6600 uses fine-grained multithreaded peripheral processors [41] and Texas Instruments uses multiple programmable real-time units (PRUs) on several SoC platforms. These PRUs are single-threaded 32-bit processors with a predictable ISA, scratchpad memories, interrupt controller, and fast I/O pins. This section demonstrates the suitability of FlexPRET as a real-time unit, primarily for the following two techniques:

- *Software-controlled peripherals using bit banging* provides flexibility for uncommon I/O protocols, varying resource requirements, or already allocated hardware peripherals. With FlexPRET, multithreading supports multiple independent I/O operations without priority conflicts, and timing instructions allow programs to specify and operate with precise timing behavior. Interrupt-driven I/O on a hardware thread does not affect HRTTs and uses fewer processor cycles than frequently checking for new data.

- *Low-level I/O processing for high-level device interfaces* offloads computation from an application processor and reduces bus traffic. Also, a low-power RTU could monitor the environment and only wake an application processor when required. FlexPRET can perform low-level computation on I/O data from either hardware peripherals, software peripherals on the same or a different hardware thread, or a combination.

5.2.1 Methodology

Hardware peripherals are more suitable or required for some precision-timed I/O operations, but the flexibility of FlexPRET presents many options for implementing software-controlled peripherals. The methodology we use to implement the software peripherals in the section is:

1. *Assign independent I/O operations to different hardware threads when possible.*
   Unless constrained by the number of hardware threads, using a separate thread for each I/O operation decouples behavior and prevents conflicts with timing instructions.

2. *Allocate scheduling resources to meet timing requirements.*
   The accuracy and precision of I/O operations depend on hardware thread scheduling, as evaluated in Section 4.1. The timing constraints dictate the minimum scheduling resources allocated to the hardware thread, and an under-allocated hardware thread would require the SRTT classification to receive the spare cycles necessary to function correctly.

3. *Use timing instructions and interrupts for increased timing precision and overall processor efficiency.*
   The `interrupt_on_expire` mechanism allows program execution to continue without waiting or polling until a specific time, and the `delay_until` mechanism generates spare cycles for SRTTs instead of wasting cycles blocking. Some safety-critical applications require polling instead of interrupts for I/O mechanisms because interrupts affect the behavior of other tasks on a single-threaded processor. In FlexPRET, each HRTT is isolated by hardware from an interrupt occurring on a different thread, avoiding this problem.

5.2.2 Evaluation

Communication Protocols

A communication protocol specifies rules for sending data between devices over a physical medium. Serial protocols use a single wire to transmit data, and parallel protocols use multiple wires to transmit data. Parallel protocols have potentially higher throughput but require more pins and wire routing with interference between wires. In addition to defining data transmission, communication protocols define addressing, data loss handling, and other control functionality.

As an example of a software peripheral for a communication protocol on FlexPRET, we implemented a one-wire communication protocol that uses a pulse-width-modulated (PWM) signal with a fixed period. An example of a device that requires this type of protocol are Adafruit NeoPixel LED strips, where the communication data is the red-blue-green (RGB) pixel value for each LED in the strip.

In this protocol, an output pin is periodically driven high (every 1250 ns). To transmit a logical 1 bit, the pin is held high for a specified length of time (e.g. 800 ns), and to transmit a logical 0 bit, the pin is held high for a different specified length of time (e.g. 400 ns). Timing imprecision is allowed within a specified margin (e.g. 150 ns) because the receiving device only needs to differentiate between a logical 1 and 0. Figure 5.4 shows the transmission of one byte of data.

Listing 5.1 is a C function that transmits an array of data using this protocol. The `PERIOD`, `HIGH1`, and `HIGH0` macros depend on the timing specification of the protocol. Timing instructions control when the output pins change value, and the `get_time()`, `set_compare(time)`, and `delay_until()` inline functions execute the respective `GT`, `CMP`, and `DU` assembly instructions. The output-changing instruction directly follows the `delay_until` operation for

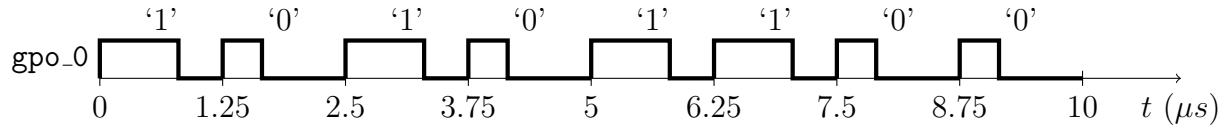


Figure 5.4: Transmission of byte value $0x35$ (00110101) from least significant to most significant bit using a PWM serial communication protocol.

Listing 5.1: C function for a variable duty cycle (PWM) protocol

```
1 void duty_comm(uint32_t* data, uint32_t length)
2 {
3     uint32_t i, j;                    // loop counters
4     uint32_t current;                 // current word of data
5     uint32_t time = get_time()+1000; // start time after init
6     for(i = 0; i < length; i++) {     // iterate through input data
7         current = data[i];            // read next word
8         for(j = 0; j < 32; j++) {     // iterate through each bit
9             time = time + PERIOD;
10            set_compare(time);
11            delay_until();            // wait until next period
12            gpo_0_high();             // output pin goes high
13            if(current & 1) {         // if bit == 1...
14                set_compare(time + HIGH1);
15                delay_until();        // stay high for .64*PERIOD
16                gpo_0_low();          // output bit goes low
17            } else {                  // if bit == 0...
18                set_compare(time + HIGH0);
19                delay_until();        // stay high for .32*PERIOD
20                gpo_0_low();          // output bit goes low
21            }
22            current = current >> 1;  // setup next bit
23        }
24    }
25 }
```

consistent timing; multiple program paths between these instructions could introduce variability into the timing behavior. The `get_time` operation only occurs once and the `time` variable stores later time values because using `get_time` for every `delay_until` operation would cause timing precision errors to accumulate.

Using the results of Section 4.1 for an I/O instruction following a `DU` instruction with a 100 MHz clock speed, the scheduling frequency needs to be faster than $f = 1/15$ for output within a 150 ns margin (a lower frequency is possible if the output operation is programmed to occur early). This value is based on the assumption that the `DU` instruction is reached before time expires, and whether this assumption holds depends on the program path.

The worst-case program path limits the minimum scheduling frequency for this program. The worst-case path occurs when the next word is fetched after a logical 1 bit, which requires exiting the j loop, incrementing the i loop, loading the next word from memory, entering the j loop, and setting a new comparison time. For example, if this path executes 12 instructions with a timing constraint of 500 ns (50 processor cycles) before the output pin goes high, the minimum scheduling frequency is $f = 1/4$. A timing analysis tool can determine worst-case paths between timing instructions to check timing requirements.

With a slight code modification, this program can implement a simple synchronous communication protocol that uses two wires; when one output pin transitions from high to low, the other output pin is holding the logical 1 or 0 bit to be transmitted. The serial peripheral interface (SPI) protocol uses a similar mechanism. Figure 5.5 shows the transmission of one byte of data. Listing 5.2 is a C function that implements this protocol, with the contents of the inner loop the only difference from the previous example.

These two protocols cannot execute independently on a single-threaded processor without hardware support. A software context switch takes longer than the time duration between output pin changes, so only one protocol can execute at a time. The single cycle context switch of fine-grained multithreading supports concurrent execution of these two protocols, and the flexible thread scheduling of FlexPRET allows SRTTs to use spare cycles during the `delay_until` operations and scheduling resource allocation to meet timing constraints. Furthermore, a hardware thread can offload the main application processor by computing data. An example for driving LED strips is the application processor sending a command that specifies a pattern with a FlexPRET RTU computing and sending pixel values.

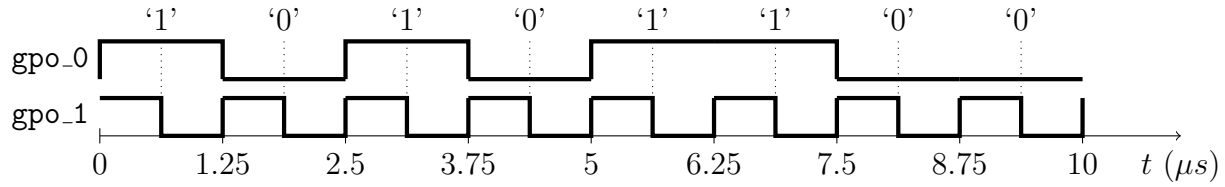


Figure 5.5: Transmission of byte value $0x35$ (00110101) from least significant to most significant bit using a synchronous communication protocol.

Listing 5.2: C function for a synchronous protocol

```
1 void sync_comm(uint32_t* data, uint32_t length)
2 {
3     uint32_t i, j;                    // loop counters
4     uint32_t current;                 // current word of data
5     uint32_t time = get_time()+1000; // start time after init
6     for(i = 0; i < length; i++) {     // iterate through input data
7         current = data[i];            // read next word
8         for(j = 0; j < 32; j++) {     // iterate through each bit
9             time = time + PERIOD/2;
10            set_compare(time);
11            delay_until();            // wait half period
12            gpo_1_high();             // posedge clk on pin
13            if(current & 1) {         // if bit == 1...
14                gpo_0_high();         // output bit goes high
15            } else {                  // if bit == 0...
16                gpo_0_low();          // output bit goes low
17            }
18            current = current >> 1;   // setup next bit
19            time = time + PERIOD/2;
20            set_compare(time);
21            delay_until();            // wait half period
22            gpo_1_low();              // negedge clk on pin
23        }
24    }
25 }
```

Chapter 6

Conclusions and Future Work

Cyber-physical and real-time embedded applications, particularly safety-critical applications, require high confidence in both software functionality and timing behavior. Integrated hardware platforms share resources to support the increasing functional complexity of applications, but interference and different levels of criticality complicate design and verification. The common approach of running a real-time operating system (RTOS) on processors optimized for average-case performance results in unpredictable behavior—mainly caused by interrupts and hardware prediction mechanisms—that is difficult to verify and certify. This dissertation focuses on processor architectures that provide high confidence in software functionality and timing behavior without sacrificing overall processor throughput.

Our processor architecture design, named FlexPRET, uses fine-grained multithreading to enable trade-offs between predictability, hardware-based isolation, and efficiency at the granularity of hardware threads. The hardware thread scheduler only executes hard real-time threads (HRTTs) at specific cycles and allows soft real-time threads (SRTTs) to use specific and spare cycles. The pipeline supports arbitrary thread interleaving, from one hardware thread to all hardware threads in the processor. Timing instructions integrate timer functionality into the processor for precise timing behavior, and these instructions also produce spare cycles when a thread is waiting.

We implemented a configurable version of FlexPRET in simulation and on FPGA, using the Chisel language to specify the design and generate both a cycle-accurate C++ simulator and Verilog code. We also implemented two baseline processors to evaluate the area and timing overhead of our techniques. FlexPRET has constant instruction latencies that depend on the scheduling frequency of the hardware thread, and overall throughput is higher when more hardware threads are interleaved in the pipeline.

Two application areas for this processor design are mixed-criticality systems and precision-timed I/O operations. By mapping tasks to threads based on criticality level, the mixed-criticality aspect is partitioned away. Critical tasks are then isolated using HRTTs for predictable WCET analysis, and less critical tasks use SRTTs and software scheduling algorithms to improve overall efficiency. Software can also perform multiple independent I/O

operations that typically require dedicated hardware by leveraging predictable instruction latencies, timing instructions, and fine-grained multithreading.

To demonstrate techniques on a processor with a low area footprint and modest throughput, the base FlexPRET design has a relatively low complexity: the pipeline is only five stages without caches, I/O peripherals, or a floating-point unit. More complex microarchitectures support applications with higher performance requirements and could be extended with similar hardware thread scheduling and timing instruction techniques. For additional pipeline components, such as a floating-point unit, the ease of integration depends on whether it supports an arbitrary interleaving of requests; if a pipelined component cannot follow the same schedule as instructions from hardware threads, it requires separate scheduling.

A processor is one component in a hardware platform for non-trivial applications. A non-volatile memory, such as Flash, stores code and data when the device is powered-off. A faster, high-capacity volatile memory, such as DRAM, stores code and data during run-time. To move data between these memories and the processor scratchpad memories, memory controllers are connected by a memory bus. If used as a co-processor or real-time unit, one or more FlexPRET processors also have a bus connection to other processors. For safe and tight WCET bounds, these hardware resources must also provide isolation and predictability, perhaps using similar scheduling techniques as the FlexPRET processor.

Research progress in scratchpad memory (SPM) management, WCET-aware compilers, and mixed-criticality support for shared resources would improve the usability and performance of predictable processors. Cache memories hide memory management from the programmer using hardware, but SPMs require explicit management by software. SPM management is necessary whenever code and data sizes are larger than SPM sizes, as FlexPRET has no direct connections to other memories. To reduce the burden on the programmer, a compiler can automatically insert code to manage SPM contents during run-time using direct memory access (DMA) commands. As memory access latency is one of the main limiters of processor performance, developing SPM management algorithms that achieve latency distributions that are competitive with caching is an important research challenge.

In general-purpose systems, the compiler optimizes code for average-case performance. In cyber-physical and real-time embedded systems, the performance criterion can be worst-case execution time instead of average-case execution time, as worst-case behavior limits the number of tasks that safety execute on a processor. An optimization that improves average-case performance can degrade worst-case performance. Therefore, compilers for real-time systems should support optimization of worst-case behavior. Rapid iterations motivate integration of WCET analysis into the compiler. FlexPRET is predictable for WCET analysis and would benefit from WCET-optimized programs.

The processor is not the only shared resource in mixed-criticality systems: system buses, memory controllers, and network-on-chips (NoCs) are often shared between tasks. As with the processor pipeline, the challenge is providing flexibility for the trade-off between isolation and efficiency—if the component is privatized in space and time for isolation, spare resources are left unused. Whereas the pipeline in FlexPRET is able to switch scheduling each cycle with no restrictions, some components are only schedulable at a coarser granular-

ity. For example, a DRAM command starts a multi-cycle operation with timing constraints on subsequent DRAM commands.

The architectural techniques used in FlexPRET also create new research directions. The flexibility of the hardware thread scheduler creates many possibilities for optimization. The *scheduling problem* for FlexPRET requires decisions at three levels: mapping of tasks to hardware threads, configuration of the slots CSR (which hardware thread has priority each cycle), and selection of the software scheduling algorithm on each hardware thread (if multiple tasks). For the example applications in Chapter 5, the scheduling decisions are manual and static. By capturing requirements in a model, however, algorithms can partly or fully automate the task mapping, hardware thread scheduling, and software scheduling decisions.

The FlexPRET scheduling problem has similarities to existing real-time scheduling problems. By considering each hardware thread as a virtual real-time machine, multiprocessor scheduling algorithms (Section 2.1.3) are applicable. The main difference is the execution frequency of each virtual real-time machine is controllable and constrained: HRTTs have a constant scheduling frequency, SRTTs have a minimum scheduling frequency, and scheduling resources are limited. This added dimension of control provides flexibility to improve schedulability, but also increases the complexity of the associated optimization problem.

For a static task mapping and hardware thread scheduling configuration, integer linear programming (ILP) or another technique could provide an optimal solution, similar to the partitioned approach in multiprocessor scheduling. If the run-time of the solver is unreasonable, heuristics can also provide solutions. Tasks can also migrate between hardware threads, similar to the global multiprocessor scheduling approach. Furthermore, the hardware thread scheduling configuration can change during run-time, supporting different modes of operation or highly dynamic operation. For example, a hardware thread's scheduling frequency could be temporarily increased to meet the deadline of a task.

A dynamic task mapping and hardware thread scheduling configuration for all tasks improves schedulability for soft-real time tasks, but reduces isolation and predictability for hard-real time tasks. Therefore, we recommend partitioning the system and using a combination of available techniques. For example, a mixed-criticality application would use static mapping and scheduling for critical tasks on HRTTs and dynamic mapping and scheduling for less critical tasks on SRTTs.

We envision FlexPRET as a foundational hardware component in a precision-timed infrastructure [75]. By supporting the specification, repeatability, and predictability of timing behavior, FlexPRET includes time in the abstraction level between software and hardware, allowing compilers and analysis tools to optimize and guarantee timing behavior. The specification of timing behavior by programming languages and models is then supported by the underlying abstraction layers—integrating timing correctness into the design flow for cyber-physical and real-time embedded systems.

Bibliography

[1] E. A. Lee, "Cyber physical systems: Design challenges," in *Proceedings of the 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, May 2008, pp. 363–369.

[2] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, 3rd ed. Springer, 2011.

[3] E. A. Lee, "Computing needs time," *Communications of the ACM*, vol. 52, no. 5, pp. 70–79, May 2009.

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

[5] *Bookout v. Toyota Motor Corp. CJ-2008-7969.*

[6] M. B. Jones. (1997). What really happened on Mars? [Online]. Available: `http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/` (visited on 06/01/2015).

[7] L. Sha, T. Abdelzaher, K.-E. Årzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok, "Real time scheduling theory: A historical perspective," *Real-Time Systems*, vol. 28, no. 2, pp. 101–155, Nov. 2004.

[8] 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 Transactions on Embedded Computing Systems*, vol. 7, no. 3, pp. 1–53, Apr. 2008.

[9] R. N. Charette. (Feb. 2009). This car runs on code, [Online]. Available: `http://spectrum.ieee.org/transportation/systems/this-car-runs-on-code` (visited on 05/26/2015).

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

[11] A. Burns and R. Davis, "Mixed criticality systems: A review," *Department of Computer Science, University of York, Tech. Rep*, 2015.

[12] *ISO 26262-1:2011 Road vehicles – Fuctional safety – Part 1: Vocabulary*, International Standardization Organization, 2011.

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

[14] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. Elsevier, 2011.

[15] P. Derler, E. A. Lee, and A. S. Vincentelli, "Modeling cyber-physical systems," *Proceedings of the IEEE*, vol. 100, no. 1, pp. 13–28, Jan. 2012.

[16] P. Axer, R. Ernst, H. Falk, A. Girault, D. Grund, N. Guan, B. Jonsson, P. Marwedel, J. Reineke, C. Rochange, M. Sebastian, R. von Hanxleden, R. Wilhelm, and W. Yi, "Building timing predictable embedded systems," *ACM Transactions on Embedded Computing Systems*, vol. 13, no. 4, 82:1–82:37, Feb. 2014.

[17] K. Goossens, A. Azevedo, K. Chandrasekar, M. D. Gomony, S. Goossens, M. Koedam, Y. Li, D. Mirzoyan, A. Molnos, A. B. Nejad, A. Nelson, and S. Sinha, "Virtual execution platforms for mixed-time-criticality systems: The CompSoC architecture and design flow," *ACM SIGBED Review*, vol. 10, no. 3, pp. 23–34, Oct. 2013.

[18] B. Wittenmark, J. Nilsson, and M. Törngren, "Timing problems in real-time control systems," in *Proceedings of the American Control Conference (ACC)*, vol. 3, Jun. 1995, pp. 2000–2004.

[19] M. P. Zimmer, J. K. Hedrick, and E. A. Lee, "Ramifications of software implementation and deployment: A case study on yaw moment controller design," in *Proceedings of the American Control Conference (ACC)*, Jul. 2015.

[20] R. Wilhelm and D. Grund, "Computation takes time, but how much?" *Communications of the ACM*, vol. 57, no. 2, pp. 94–103, Feb. 2014.

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

[22] J. Reineke, D. Grund, C. Berg, and R. Wilhelm, "Timing predictability of cache replacement policies," *Real-Time Systems*, vol. 37, no. 2, pp. 99–122, Aug. 2007.

[23] E. A. Lee and S. A. Seshia, "Quantitative analysis," in *Introduction to Embedded Systems, A Cyber-Physical Systems Approach*, Second Edition, http://LeeSeshia.org. 2015.

[24] 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*, vol. 03471, Sep. 2004.

[25] J. Kotker, D. Sadigh, and S. A. Seshia, "Timing analysis of interrupt-driven programs under context bounds," in *Proceedings of Formal Methods in Computer-Aided Design (FMCAD)*, Oct. 2011, pp. 81–90.

[26] J. Staschulat and R. Ernst, "Multiple process execution in cache related preemption delay analysis," in *Proceedings of the 4th ACM International Conference on Embedded Software (EMSOFT)*, Sep. 2004, pp. 278–286.

[27] S. Altmeyer, R. I. Davis, and C. Maiza, "Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems," *Real-Time Systems*, vol. 48, no. 5, pp. 499–526, Jun. 2012.

[28] I. Puaut, "WCET-centric software-controlled instruction caches for hard real-time systems," in *Proceedings of the 18th Euromicro Conference on Real-Time Systems (ECRTS)*, Jul. 2006, pp. 217–226.

[29] J. Whitham and N. C. Audsley, "Explicit reservation of local memory in a predictable, preemptive multitasking real-time system," in *Proceedings of the 18th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Apr. 2012, pp. 3–12.

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

[31] M. Schoeberl, "A Java processor architecture for embedded real-time systems," *Journal of Systems Architecture*, vol. 54, no. 1, pp. 265–286, Jan. 2008.

[32] J. Mische, I. Guliashvili, S. Uhrig, and T. Ungerer, "How to enhance a superscalar processor to provide hard real-time capable in-order SMT," in *Proceedings of the 23rd International Conference on Architecture of Computing Systems (ARCS)*, Feb. 2010, pp. 2–14.

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

[34] O. Avissar, R. Barua, and D. Stewart, "Heterogeneous memory management for embedded systems," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, Nov. 2001, pp. 34–43.

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

[36] Y. Kim, D. Broman, J. Cai, and A. Shrivastaval, "WCET-aware dynamic code management on scratchpads for software-managed multicores," in *Proceedings of the 20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Apr. 2014, pp. 179–188.

[37] M. Schoeberl, "Time-predictable cache organization," in *Proceedings of the 2009 Software Technologies for Future Dependable Distributed Systems (STFSSD)*, Mar. 2009, pp. 11–16.

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

[39] *653P1-3 Avionics Application Software Interface, Part 1, Required Services*, Aeronautical Radio, Inc. Std., 2010.

[40] Wind River. (2015). VxWorks 653 platform, [Online]. Available: `http://www.windriver.com/products/vxworks/certification-profiles/` (visited on 06/22/2015).

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

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

[43] I. Liu, "Precision timed machines," PhD thesis, EECS Department, University of California, Berkeley, May 2012.

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

[45] N. J. H. Ip and S. A. Edwards, "A processor extension for cycle-accurate real-time software," in *Proceedings of the IFIP International Conference on Embedded and Ubiquitous Computing (EUC)*, vol. LNCS 4096, Aug. 2006, pp. 449–458.

[46] D. Bui, E. A. Lee, I. Liu, H. D. Patel, and J. Reineke, "Temporal isolation on multiprocessing architectures," in *Proceedings of the 48th Design Automation Conference (DAC)*, Jun. 2011, pp. 274–279.

[47] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber, A. Jordan, E. Kasapaki, J. Knoop, Y. Li, D. Prokesch, W. Puffitsch, P. Puschner, A. Rocha, C. Silva, J. Sparsø, and A. Tocchi, "T-CREST: Time-predictable multi-core architecture for embedded systems," *Journal of Systems Architecture*, 2015.

[48] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Computing Surveys*, vol. 43, no. 4, p. 35, Oct. 2011.

[49] T. N. B. Anh and S.-L. Tan, "Real-time operating systems for small microcontrollers," *IEEE Micro*, vol. 29, no. 5, pp. 30–45, Sep. 2009.

[50] T. P. Baker and A. Shaw, "The cyclic executive model and Ada," *Real-Time Systems*, vol. 1, no. 1, pp. 7–25, Jun. 1989.

[51] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973.

[52] G. C. Buttazzo, "Rate monotonic vs. EDF: Judgment day," *Real-Time Systems*, vol. 29, no. 1, pp. 5–26, Jan. 2005.

[53] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, Sep. 1990.

[54] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM Journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, Mar. 1969.

[55] E. G. Coffman Jr, M. R. Garey, and D. S. Johnson, "An application of bin-packing to multiprocessor scheduling," *SIAM Journal on Computing*, vol. 7, no. 1, pp. 1–17, Feb. 1978.

[56] S. K. Dhall and C. L. Liu, "On a real-time scheduling problem," *Operations Research*, vol. 26, no. 1, pp. 127–140, Feb. 1978.

[57] B. Andersson and J. Jonsson, "Fixed-priority preemptive multiprocessor scheduling: To partition or not to partition," in *Proceedings of 7th International Conference on Real-Time Computing Systems and Applications (RTCSA)*, Dec. 2000, pp. 337–346.

[58] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 1977, pp. 238–252.

[59] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper, "Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution," in *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS)*, Dec. 2006, pp. 57–66.

[60] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm, "Reliable and precise WCET determination for a real-life processor," in *Proceedings of the 1st International Workshop on Embedded Systems (EMSOFT)*, Sep. 2001, pp. 469–485.

[61] Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," in *Proceedings of the 32nd Annual Design Automation Conference (DAC)*, Dec. 1995, pp. 456–461.

[62] T. Lundqvist and P. Stenstrom, "Timing anomalies in dynamically scheduled microprocessors," in *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS)*, Dec. 1999, pp. 12–21.

[63] Rapita Systems. (2015). RapiTime, [Online]. Available: `https://www.rapitasystems.com/products/rapitime`.

[64] S. A. Seshia and A. Rakhlin, "Game-theoretic timing analysis," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov. 2008, pp. 575–582.

[65] S. A. Seshia and J. Kotker, "Gametime: A toolkit for timing analysis of software," in *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Mar. 2011, pp. 388–392.

[66] S. A. Seshia and A. Rakhlin, "Quantitative analysis of systems using game-theoretic learning," *ACM Transactions on Embedded Computing Systems*, vol. 11, no. S2, 55:1–55:27, Aug. 2012.

[67] J. D. Ichbiah, B. Krieg-Brueckner, B. A. Wichmann, J. G. P. Barnes, O. Roubine, and J.-C. Heliard, "Rationale for the design of the Ada programming language," *SIGPLAN Notices*, vol. 14, no. 6, pp. 1–261, Jun. 1979.

[68] E. Kligerman and A. D. Stoyenko, "Real-Time Euclid: A language for reliable real-time systems," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 9, pp. 941–949, Sep. 1986.

[69] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, Jan. 2003.

[70] National Instruments. (2015). LabVIEW, [Online]. Available: `http://www.ni.com/labview/`.

[71] Y. Zhao, J. Liu, and E. A. Lee, "A programming model for time-synchronized distributed real-time systems," in *Proceedings of 13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS)*, Apr. 2007, pp. 259–268.

[72] *IEEE standard for a precision clock synchronization protocol for networked measurement and control systems".*

[73] S. Matic, I. Akkaya, M. Zimmer, J. C. Eidson, and E. A. Lee, "PTIDES model on a distributed testbed emulating smart grid real-time applications," in *Proceedings of the IEEE Conference on Innovative Smart Grid Technologies (ISGT-EUROPE)*, Dec. 2011, pp. 1–8.

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

[75] 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 *Proceedings of the Electronic System Level Synthesis Conference (ESLsyn)*, May 2013, pp. 1–6.

[76] L. Thiele and R. Wilhelm, "Design for timing predictability," *Real-Time Systems*, vol. 28, no. 2, pp. 157–177, Nov. 2004.

[77] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand, "Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 7, pp. 966–978, Jul. 2009.

[78] C. Cullmann, C. Ferdinand, G. Gebhard, D. Grund, C. Maiza (Burguière), J. Reineke, B. Triquet, and R. Wilhelm, "Predictability considerations in the design of multi-core embedded systems," *Proceedings of Embedded Real Time Software and Systems (ERTS)*, pp. 36–42, May 2010.

[79] C. Rochange and P. Sainrata, "A time-predictable execution mode for superscalar pipelines with instruction prescheduling," in *Proceedings of the 2nd Conference on Computing Frontiers (CF)*, May 2005, pp. 307–314.

[80] J. Whitham and N. Audsley, "Time-predictable out-of-order execution for hard real-time systems," *IEEE Transactions on Computers*, vol. 59, no. 9, pp. 1210–1223, Sep. 2010.

[81] F. Bodin and I. Puaut, "A WCET-oriented static branch prediction scheme for real time systems," in *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS)*, Jul. 2005, pp. 33–40.

[82] J. Yan and W. Zhang, "A time-predictable VLIW processor and its compiler support," *Real-Time Systems*, vol. 38, no. 1, pp. 67–84, Oct. 2007.

[83] M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, C. W. Probst, S. Karlsson, and T. Thorn, "Towards a time-predictable dual-issue microprocessor: The Patmos approach," in *Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES)*, vol. 18, Mar. 2011, pp. 11–21.

[84] A. El-Haj-Mahmoud, A. S. AL-Zawawi, A. Anantaraman, and E. Rotenberg, "Virtual multiprocessor: An analyzable, high-performance architecture for real-time computing," in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, Sep. 2005, pp. 213–224.

[85] 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, Nov. 1999.

[86] I. Puaut and C. Pais, "Scratchpad memories vs locked caches in hard real-time systems: A quantitative comparison," in *Proceedings of the 2007 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Apr. 2007, pp. 1–6.

[87] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee, "PRET DRAM controller: Bank privatization for predictability and temporal isolation," in *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Oct. 2011, pp. 99–108.

[88] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar, "Bounding memory interference delay in COTS-based multi-core systems," in *Proceedings of the 20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Apr. 2014, pp. 145–154.

[89] M. Paolieri, E. Quiñones, and F. J. Cazorla, "Timing effects of DDR memory systems in hard real-time multicore architectures: Issues and solutions," *ACM Transactions on Embedded Computing Systems*, vol. 12, no. 1, 64:1–64:26, Mar. 2013.

[90] H. Kim, D. Broman, E. A. Lee, M. Zimmer, A. Shrivastava, and J. Oh, "A predictable and command-level priority-based DRAM controller for mixed-criticality systems," *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 317–326, Apr. 2015.

[91] M. Zimmer, D. Broman, C. Shaver, and E. A. Lee, "FlexPRET: A processor platform for mixed-criticality systems," in *Proceedings of the 20th IEEE Real-Time and Embedded Technology and Application Symposium (RTAS)*, Apr. 2014, pp. 101–110.

[92] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The RISC-V instruction set manual, volume I: User-level ISA, version 2.0," EECS Department, University of California, Berkeley, UCB/EECS-2014-54, May 2014.

[93] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The RISC-V compressed instruction set manual, version 1.7," EECS Department, University of California, Berkeley, UCB/EECS-2015-157, May 2015.

[94] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanović, "The RISC-V instruction set manual volume II: Privileged architecture version 1.7," EECS Department, University of California, Berkeley, UCB/EECS-2015-49, May 2015.

[95] 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 *Proceedings of the 49th Annual Design Automation Conference (DAC)*, Jun. 2012, pp. 1216–1225.

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

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

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

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

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

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

[102] M. Lemerre, E. Ohayon, D. Chabrol, M. Jan, and M.-B. Jacques, "Method and tools for mixed-criticality real-time applications within PharOS," in *Proceedings of the 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW)*, Mar. 2011, pp. 41–48.

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