RTSS, Paris, France, December, 2017, DOI: 10.1109/RTSS.2017.00041

Abstract PRET Machines

Invited TCRTS award paper

Edward A. Lee (award recipient)
UC Berkeley
Berkeley, CA 94720
Email: eal@eecs.berkeley.edu

Jan Reineke
Saarland University
Saarland Informatics Campus
Saarbrücken, Germany
Email: reineke@cs.uni-saarland.de

Michael Zimmer
Swarm64 AS
Berlin, Germany
Email: michael@swarm64.com

***Abstract*—Prior work has shown that it is possible to design microarchitectures called PRET machines that deliver precise and repeatable timing of software execution without sacrificing performance. That prior work provides specific designs for PRET microarchitectures and compares them against conventional designs. This paper defines a class of microarchitectures called abstract PRET machines (APMs) that capture the essential temporal properties of PRET machines. We show that APMs deliver deterministic timing with no loss of performance for a family of real-time problems consisting of sporadic event streams with deadlines equal to periods. On the other hand, we observe a tradeoff between deterministic timing and the ability to meet deadlines for sporadic event streams with constrained deadlines.**

I. Introduction

Cyberphysical systems (CPSs) involve software that interacts with the physical world, often with timing-sensitive safety-critical physical sensing and actuation. The timing of the execution of the software in such systems matters quite a bit because it matters in the physical world. Real-time software is not just an information technology because delivering the correct *information* is not sufficient. *When* the information is delivered is also important.

Way back in 1988, Stankovic cataloged a few misinterpretations of the term "real time" and laid out a research agenda that is dishearteningly valid today [1]. For example, he points out that real-time computing is not just fast computing. In fact, many real-time systems execute on decidedly slow computers, such as low-end microcontrollers, and timing precision, predictability, and repeatability are more important than speed.

Exact timing in physical processes is difficult to define, much less achieve. However, real-time systems may require that events occur in a specified order given by some deterministic model. And they may require that events occur sufficiently close to a specified event in some time measurement device. What is "sufficiently close" will depend on the application. A control system engineer, for example, may use a Newtonian model of time and, under this model, may be able to prove that stability is maintained if events occur within some specified latency after some stimulus.

No engineered system is perfect. No matter what specifications we use for what a "correct behavior" of the system is, there will always be the possibility that the realized system will deviate from that behavior in the field. The goal of engineering, therefore, needs to be to clearly define what is a correct behavior, to design a system that realizes that behavior with high probability, to provide detectors for violations, and to provide safe fallback behaviors when violations occur.

A straightforward way to define correct behavior is to specify what properties the output of a system must exhibit for each possible input. A **repeatable** property is one that is exhibited by every correct behavior for a given input. This notion is central to the idea of testing, where a system is checked to see whether its reaction to specified test inputs yields the required properties. For real-time systems, the timing of an output is an essential property of that output. Edwards et al. argue that "repeatable timing is more important and more achievable than predictable timing" [2].

At a minimum, to achieve these goals for CPS, real-time software methodologies rely on being able to bound the execution time of sections of code. We may also need tighter control over timing in order to ensure that the *order* of events conforms to a specified correct behavior. A simple bound on execution time may not be sufficient because finishing an execution early may change the order in which events occur.

But even the minimal requirement, bounding execution time, is difficult both in theory and in practice. In theory, we know from Rice's theorem that all nontrivial, semantic properties of programs are undecidable [3]. Whereas timing is not a semantic property of programs in any modern programming language, it *depends on* nontrivial semantic properties of programs. For example, it depends on halting, in that any program (or program segment) that fails to halt has no bound on its execution time unless execution is infinitely fast. Therefore, in theory, bounded time is undecidable.

But "undecidable" simply means that no algorithm can determine whether the property holds for *all* programs. In practice, engineers routinely rely on execution-time analysis that "solves" the halting problem, bounding the length of the paths that an execution takes through a program.

This work was supported in part by the iCyPhy Research Center (Industrial Cyber-Physical Systems, supported by Denso, Ford, IHI, National Instruments, Siemens, and Toyota), and by the National Science Foundation, NSF award #1446619 (Mathematical Theory of CPS).

There are a number of reasons that timing properties are difficult to make repeatable. At the microarchitecture level, instruction set architectures (ISAs) define the correct behavior of a microprocessor implementation in a way that makes timing irrelevant to correctness. Timing is merely a performance metric, not a correctness criterion. In contrast, arithmetic and logic *are* correctness criteria. A microprocessor that fails to take a branch when the condition evaluates to true is simply an incorrect implementation. But a microprocessor that takes a long time to take the branch is just slow. Computer architects have long exploited this property, that timing is irrelevant to correctness. They have developed clever ways to deal with deep pipelines, such as speculative execution and instruction reordering, and with memory heterogeneity, such as multi-level caches. These techniques, however, introduce highly variable and unpredictable timing. The goal is to speed up a typical execution, not to make timing properties repeatable.

The design of modern programming languages reflects the microarchitectural choice, so timing is again irrelevant to correctness. Hence, programmers have to step outside the programming abstraction to control timing, for example by writing to memory-mapped registers to set up a timer interrupt, or more indirectly, by making operating system calls to trigger context switches. The result is timing granularity that is much more coarse than what is achievable in the hardware. More important, since interrupts occur unpredictably relative to whatever is currently executing, these techniques inevitably make timing behavior nonrepeatable.

Despite these challenges, engineers have managed to make reliable real-time systems. How? Techniques include:

1. overprovisioning,
2. using old technology,
3. execution time analysis, and
4. real-time operating systems (RTOSs).

Overprovisioning is common because Moore's law has given us impressively fast processors. If the execution of software is essentially instantaneous with respect to the physical processes with which it is interacting, then the timing of the software becomes irrelevant. However, overprovisioning is becoming increasingly difficult as the complexity of CPS applications increases and as Moore's law slows down. Moreover, many CPS applications are extremely cost sensitive or energy constrained, making overprovisioning a poor choice.

Using old technology is also common. Safety-critical avionics software, for example, rarely uses modern programming languages, operating systems, or even interrupts. Software is written at a very low level, I/O is done through polling rather than interrupts, and multitasking is avoided. Programmable logic controllers (PLCs), widely use in industrial automation, are often programmed using ladder logic, a notation that dates back to the days when the logic of digital controllers was entirely controlled with mechanical relays. Many embedded systems designers avoid multicore chips because of problems they introduce with timing [4], a strategy that is becoming increasingly difficult as single-core chips become more rare. And programmers often disable or lock caches, thereby getting little advantage from the memory hierarchy.

The third approach, execution-time analysis, puts bounds on the time it can take for sections of code to execute [5]. This is fundamentally a hard problem because of Rice's theorem. However, even when the execution paths through the code can be analyzed, often with the help of manual annotations such as bounds on loops, the microarchitectural features mentioned above can make analysis extremely difficult. The analysis tools need a detailed model of the particular implementation of the processor that will run the code, including every minute (and often undocumented) detail. As a result, a program that has been validated using execution-time analysis is only validated for the particular piece of silicon that has been modeled. Manufacturers of safety-critical embedded systems, therefore, are forced to stockpile the hardware that they expect to need for the entire production run of a product. This runs counter to most basic principles in modern supply chain management for manufacturing, and it makes it impossible to take advantage of technology improvements for cost reduction, improved safety, or reduced energy consumption.

Moreover, execution-time analysis tools often need to make unrealistic assumptions, such as that interrupts are disabled, in order to get reasonable bounds. Interrupts can disrupt the state of the machine, for example by altering the cache or changing the state of the branch predictor. If interrupts are enabled, then analysis tools need to make pessimistic assumptions about the state of the machine, resulting in loose bounds on execution time [6]. If bounds are loose, then overprovisioning is unavoidable.

In practice, designers either avoid interrupts altogether (as commonly done in avionics) or attempt to keep program segments short so that the time during which interrupts are disabled is small. Both strategies are increasingly difficult as we demand more functionality from these programs. As execution time increases, either the polling frequency decreases or the variability of the timing of other tasks that get locked out by disabled interrupts increases.

The fourth technique, RTOSs, provides real-time scheduling policies in a multitasking operating system. At the core, RTOSs use timer interrupts and priorities associated with tasks. There is a long history of strategies that can be proven optimal under (often unrealistic) assumptions, such as no context-switch overhead [7]. In simple scenarios, these strategies can yield repeatable behaviors, but in more complex scenarios, they can even become chaotic [8], which makes behavior impossible to predict. Moreover, because of the reliance on interrupts, RTOSs violate the typical assumptions made for execution-time analysis. For such an approach to be sound, great care needs to be taken to account for all overhead induced by context switches including cache-related preemption delays [9]. A consequence is that when RTOSs deliver predictable timing, the precision of the resulting timing is several orders of magnitude coarser than what is in principle achievable with the underlying digital hardware.

Taken together, these techniques do make it *possible* to

design safety-critical real-time embedded software, but their weaknesses suggest that it may be time to step back and reexamine the problem of real-time computing with fresh eyes. After all, microprocessors are realized in a technology, synchronous digital logic, that is capable of realizing sub-nanosecond timing precision with astonish reliability and repeatability. It is the layers of abstraction overlaid on this technology, ISAs, programming languages, RTOSs, and networks, that discard timing.

In this paper, we focus on timing precision, predictability, and repeatability. We build on prior work that shows that it is possible to design microarchitectures called PRET machines that deliver precise and repeatable timing of software execution without sacrificing performance. That prior work has demonstrated this by providing specific designs for PRET microarchitectures and comparing them against conventional designs. This paper defines a class of microarchitectures called abstract PRET machines (**APM**s) that capture the essential temporal properties of PRET machines. We prove that APMs deliver deterministic timing with no loss of performance by considering a family of real-time problems consisting of sporadic event streams and time-critical reactions to those events. Moreover, we show that for some workloads, APMs can achieve *better* performance than a conventional machine by eliminating pipeline bubbles and requiring less pessimistic bounds on execution time.

II. Background Work

In 2007, Edwards and Lee introduced the term **PRET machines** for precision timed machines and argued for their use in real-time applications. At the time, they admitted that "the revolution may take decades" [10]. Indeed, the revolution is off to a slow start, though there are some signs of interest in industry.

A. Processors

XMOS (http://www.xmos.com/), founded in 2005, has produced since 2008 multicore processors with deterministic timing aimed at high-performance audio applications. XMOS coined the term "software defined silicon" to capture the fact that the software in these processors can interact with hardware at a level of precision comparable to hardware.

ARM Holdings (http://www.arm.com/), a leading producer of embedded processors now owned by SoftBank, announced in 2016 an ARM Cortex-R52 processors that claims "hard determinism" and appears to have some PRET-like features [11]. Specifically, the architecture is optimized to avoid variable time, nondeterministic operations and is also optimized quick interrupt entry and context switching.

De Dinechin et al. [12] describe the Kalray MPPA®-256, a many-core processor specifically designed to be suitable for time-critical computing. It consists of 16 clusters, which each contain 16 in-order VLIW cores. Each of the clusters also contains 16 SRAM banks, which can be flexibly allocated to the 16 cores of the respective cluster to eliminate interference between processes on different cores.

There are also commercial specialized coprocessors with some PRET-like features. One example is Qualcomm's Hexagon, used as a digital signal processor (DSP) in the Qualcomm Snapdragon SoC. Another example is NXP Semiconductors' second-generation Enhanced Time Processor Units (eTPU2), used for timing control in microcontrollers. These units support 32 channels that can each process an input signal and generate an output signal, with a priority-based hardware scheduler to support concurrency.

PRET machines have also been a subject of study in research labs. Our Berkeley team has developed three generations of PRET architectures, the third of which forms the foundation for this paper. Schoeberl has developed a Java processor and a family of techniques for managing memory hierarchies in a deterministic way [13], [14]. Hahn et al. [15] show to build a pipelined processor without timing anomalies [16], [17]. On the software side, Andalam et al. have proposed a language called PRET-C which they evaluated on a dedicated target architecture called ARPRET, a modified version of the Xilinx MicroBlaze with some PRET-like features [18].

B. Execution-Time Analysis

Execution-time analysis is a challenging problem [5]. In addition to solving the undecidable problems of program flow analysis, accurate determination of worst-case execution time (**WCET**) requires detailed modeling of the processor implementation, including all details of the memory system, pipeline, and instruction set realization. A state-of-the-art industrial tool, AbsInt's aiT (https://www.absint.com/ait/), was used on the Airbus A380 for analyzing safety-critical software. Open-source research tools such as platin [19] and GameTime [20] illustrate creative innovations that are possible. GameTime, for example, combines flow analysis with empirical measurements on a particular processor implementation.

Assessing the effectiveness of these tools is challenging. Wägemann et al. describe a tool called GenE that synthesizes benchmarks whose flow facts are known [21]. These benchmarks can be used to test WCET tools because the actual worst-case flow path is known by construction. They use the tool to evaluate aiT and platin, showing that for most benchmarks, aiT comes impressively close to the actual WCET, whereas platin does not do as well. However, their synthesized benchmarks also discovered programs for which aiT's WCET estimate was *less* than the actual execution time, revealing a bug in the tool's modeling of the ARM Cortex-M4 platform (which was later fixed, according to AbsInt). Such bugs seem inevitable because, as expressed by Wägemann et al. "the most accurate execution-time model is the processor itself" (which arguably is not a model), and the processor itself is rarely fully documented. PRET machines help to correct this problem by including certain timing properties in the very definition of the ISA. They become part of what it means to correctly realize the processor architecture.

Another key property of PRET machines is that the execution time of each instruction is independent of execution history. Conventional processors do not have this property, which

reduces repeatability and increases the complexity of timing analysis. For example, whether a memory instruction misses in a cache and takes long to execute depends on the state of the cache, which depends on the execution history. Capitalizing on this property, Reineke and Doerfert [22] introduce *architecture-parametric* WCET analysis, which bounds a program's WCET in terms of architectural parameters, such as the scratchpad or the DRAM latency, which may vary from one generation of a PRET architecture to the next.

C. Berkeley PRET Machines

We base this paper on the latest of three generations of PRET machines developed at Berkeley. The first generation, which was never realized in silicon, uses a SPARC-based processor model to demonstrate the ideas of using hardware multithreading, scratchpad memories, and scheduled access to main memory in PRET machines [23]. The second generation, PTARM, which was realized as an FPGA implementation, was an extension of an ARM instruction set [24]. This project demonstrated that with sufficient concurrency in the application, achieving repeatable timing behavior does not come at a performance cost. Reineke et al. also used this architecture to show that variable latencies in DRAM memories could be managed by bank privatization [25].

The third generation, FlexPRET, is an open-source RISC-V architecture with a variable number of hardware threads [26], [27]. FlexPRET can operate as a conventional RISC-V processor with only one thread, a mode in which its performance is comparable to a conventional in-order processor. But an application can define hard-real-time threads that, when active, occupy a fixed, deterministic schedule of execution with repeatable timing. FlexPRET thereby supports a mixture of timing-critical and best-effort software tasks, opportunistically devoting to the best-effort tasks all cycles that are unused by the timing-critical tasks. Kim et al. describe a memory controller that supports such mixed criticality systems, preserving timing determinacy for timing-critical tasks without sacrificing performance for the best-effort tasks [28].

All Berkeley PRET machines use fine-grained multithreading (interleaved multithreading, barrel processor), a technique dating back to the CDC 6600 and also found currently in XMOS processors. In fine-grained multithreading, instructions from different hardware threads are interleaved in the processor pipeline. When a pipeline only executes instructions from a single hardware thread, resolving dependencies between instructions in different pipeline stages requires either speculation or not performing useful work for a cycle or more. Each wasted cycle is called a pipeline bubble. Increased spacing between dependent instructions, introduced by interleaving, reduces or eliminates both speculation and pipeline bubbles. The costs of such interleaving are concurrently storing state for all hardware threads, each of which requires its own register set; reduced throughput of any particular hardware thread; and fine-grained sharing between threads of cache or scratchpad memory. For this price, the timing of program execution becomes much more deterministic, and strong assurances can be provided of meeting hard-real-time deadlines.

The SPARC-based PRET machine and PTARM use fixed round-robin scheduling between the hardware threads, even if a hardware thread is idle waiting for an event, to provide predictable and repeatable timing. In other words, each hardware thread executes at a fixed, constant rate, for example, once every four cycles. The XMOS XS1 processor supports round-robin scheduling between all active hardware threads, which allows a hardware thread to wait for an event without using any pipeline cycles. The drawback for timing precision is that the rate at which a hardware thread executes then depends on the number of active threads at each clock cycle. FlexPRET offers the best of both by supporting an arbitrary interleaving of fine-grained threads with a flexible scheduler.

In FlexPRET, hardware threads are classified as either hard-real-time threads (**HRT**) or soft-real-time threads (**SRT**). The scheduler issues instructions from HRTs at a fixed, constant rate, so the timing of these threads is not affected by anything else going on in the machine. Only in a cycle that is not reserved for an HRT will the scheduler issue an instruction from an SRT. SRTs can also reserve cycles to maintain a minimum rate, but otherwise the scheduler uses a round-robin selection between all active SRTs. The number of HRTs and SRTs is variable but bounded by the number of register sets provided by the hardware. Of course, both HRTs and SRTs can perform conventional multitasking, so the total number of threads in the application is not limited by the hardware. A FlexPRET configured with a single SRT is, effectively, a conventional RISC-V processor.

The FlexPRET thread scheduler was designed to provide flexibility with low complexity, but other implementations could also provide properties useful for PRET machines. In this paper, we ignore the potential complexity of the scheduler, defining abstract PRET machines to be flexible enough to achieve any specified duty cycle for HRTs.

All Berkeley PRET machines also add temporal properties to some instructions in the ISA to make timing behavior a property of a program instead of a side effect of the implementation. The challenge is to introduce timing control without preventing performance optimizations by compilers and processor architects. One example is a deadline instruction that waits until a counter, decremented every cycle, reaches zero and continues if the deadline has already expired [29]. The units of the counter can be either clock cycles or time. A more interesting and stricter version of this instruction, called MTFD [30], guarantees that the deadline has not expired when the instruction executes. Timing analysis is then needed for a program to run correctly on a particular processor.

III. Abstract PRET Machines

In this paper, we introduce **abstract PRET machines** (**APM**s) based on the FlexPRET architecture described in the previous section. An APM issues a single instruction each cycle, and its pipeline supports all possible interleavings of

instructions from hardware threads. By using an idealized case, we evaluate an entire family of concrete PRET machines.

Let T denote the set of all hardware threads. Then $N = |T|$ is the number of hardware threads, or equivalently, the number of sets of machine registers. At any given time t, $H(t) \subseteq T$ denotes the set of active hardware threads that are designated hard-real-time threads. The dispatch unit dispatches instructions from these threads into the execution unit according to a fixed periodic schedule. Any cycles that are left unused by this schedule are used by any active soft-real-time threads in $T \setminus H(t)$.

Each HRT $h \in H(t)$ has a periodic schedule for dispatching its instructions into the execution pipeline. We assume that this periodic schedule is sparse enough that there will never be pipeline bubbles that prevent the dispatch of an instruction from an HRT into the pipeline or that when bubbles do occur, they occur deterministically, independent of the data. Under this assumption, the execution of an HRT exhibits deterministic timing at the precision of a clock cycle. This assumption is easy to achieve in practice. The FlexPRET microarchitecture presented in [26], [27] never experiences pipeline bubbles when an instruction is issued every fourth clock cycle. Furthermore, if instructions are issued more frequently than this, a pipeline bubble in FlexPRET reduces overall throughput but not determinism because FlexPRET forgoes optimizations like branch prediction and speculative execution.

Let $p_h(t)$ denote the **duty cycle** or fraction of dispatch cycles used by thread h at time t. E.g., if $p_h(t) = 0.25$, then the schedule for h may dispatch an instruction every fourth clock cycle. Clearly, $0 \leq p_h(t) \leq 1$ for all t. No HRT may use more than 100% of the available cycles in the machine.

The fraction of cycles used by all HRTs at time t is

$$u_H(t) = \sum_{h \in H(t)} p_h(t). \tag{1}$$

We call this the **HRT utilization**. We similarly require that $u_H(t) \leq 1$ for all t. The ensemble of HRTs cannot use more than 100% of the available cycles. But in addition to this trivial constraint, there is a more complicated constraint that HRTs must make mutually exclusive use of hardware resources. This is easily accomplished by a brute-force method of constraining the values of $p_h(t)$ to a few possibilities (e.g., 1/2, 1/4, 1/8, 1/16) and then performing a round-robin policy. But optimizing the schedule with more allowable values of $p_h(t)$ is likely to be challenging. For our discussion of APMs, we ignore this nontrivial scheduling problem, but any concrete implementation will require some mechanism for constructing the schedules for HRTs (see section VII below).

If $u_H(t) = 1$ for any t, then at that time, no cycles are available for SRTs. Many applications will strive to avoid this situation or at least ensure that this situation does not persist for very long because it could make applications non-responsive to soft-real-time tasks such as user interaction.

The worst-case execution time (WCET) of software running on an APM depends on the duty cycle of the HRT it is running in. We capture this dependency by a function $C(p)$, mapping the HRT's duty cycle p, with $0 < p \leq 1$, to the software's WCET.

A duty cycle of 1 corresponds to conventional pipelined execution. When comparing PRET machines with conventional processors, we assume an execution time of $C(1)$ on the conventional processor, and we use C as an abbreviation for $C(1)$ in such cases.

Smaller duty cycles generally lead to greater execution times, and so $C(p) \geq C(1)$ for $0 < p < 1$, but smaller duty cycles also reduce and eventually eliminate pipeline bubbles, as consecutive instructions of a HRT are spread further apart, and so

$$C(p) \cdot p \leq C, \tag{2}$$

which captures that thread interleaving may increase throughput compared with conventional pipelined execution of a single thread. The above inequality can be slightly generalized as follows:

$$\forall p, q : p \leq q \Rightarrow C(p) \cdot p \leq C(q) \cdot q, \tag{3}$$

which simplifies to (2) for $q = 1$.

IV. Interrupts

Interrupts are the standard way that all modern microprocessors get data in and send data out to the outside world. To be sure, interrupts create many subtle software problems. As far back as 1972, Edsger Dijkstra lamented, "[I]n one or two respects modern machinery is basically more difficult to handle than the old machinery. Firstly, we have got the interrupts, occurring at unpredictable and irreproducible moments; compared with the old sequential machine that pretended to be a fully deterministic automaton, this has been a dramatic change, and many a systems programmer's grey hair bears witness to the fact that we should not talk lightly about the logical problems created by that feature" [31]. Despite this lament, to this day, interrupts remain the primary method for I/O and are central to every modern operating system design including real-time OSs.

But interrupts make the execution time of any particular chunk of software unpredictable. In addition to the time that it takes to execute the interrupt service routine (**ISR**), the execution of the ISR will disrupt state in the machine that affects the execution time of the program that is interrupted. For example, the ISR can disrupt the cache and the state of the branch predictor. For these reasons, all modern execution-time analysis tools assume uninterrupted execution. Overhead due to interrupts has to be accounted for during response-time analysis. While there is a lot of work on bounding the effect of interrupts on the cache state [32], [33], [9], we are not aware of an all-encompassing analysis accounting for all overhead induced by interrupts.

Many designers of safety-critical real-time software avoid interrupts altogether. Some, such as aircraft manufacturers, are even prohibited from using interrupts, thereby excluding

from their software toolkits almost all of the last 40 years of advances in operating systems.

We consider problems where the environment creates events to which the software must respond. Some such problems, including many classical feedback control problems, tolerate timing variability as long as the average latency remains bounded. These classical feedback control problems are essentially continuous systems in that timing perturbations have bounded effects. Such problems are best handled using the SRTs of an APM because these will minimize the average response time, just as in a conventional architecture. If an application includes *only* such continuous problems, then APMs do not provide much benefit; their performance will be comparable to that of a conventional architecture and their HRTs will go unused.

For APMs, the more interesting problems are ones that benefit from deterministic response times. These include systems with discontinuous behaviors, such as discrete-event systems, where for example reversing the order of two events can have drastic consequences. For such situations, the HRTs of an APM are extremely useful. Such situations, with careful design, can still be handled by conventional techniques if the response time can be *bounded*. But if we can reduce the *variability* of the response times, then we can improve testability and confidence. We show here that we *can* reduce the variability and that there is no cost in performance to doing so.

For a polling style of I/O, both the variability and the latency will depend on the polling interval. This creates a pressure for small polling intervals, which leads to overprovisioning. An interrupt style of I/O, on the other hand, results in variabilities and latencies that are hard to control in conventional machines, but easy to control in PRET machines. We therefore focus on the interrupt style.

In order to make any guarantees at all about the response of a software system to interrupt requests, we have to impose some constraints on the environment. No microprocessor can respond to an unbounded number of interrupts requests in bounded time. Even PRET machines cannot perform such magic.

We model interrupt requests as sporadic streams of events, which are events that arrive at random with a minimum inter-arrival time T (for historical reasons this is often also referred to as the *period*). No two interrupt requests from the environment belonging to the same sporadic stream may occur within less than T time units.

A. Interrupt Handling on a Conventional Processor

If the execution time $C > T$, then no real-time guarantee is possible because the sporadic model allows requests to arrive every T time units in the worst case. A more interesting scenario has multiple sporadic streams of interrupt requests. Consider n streams with periods $T_1, \ldots, T_n$. Suppose that the time it takes to execute the responses to these is $C_1, \ldots, C_n$. Then, under work-conserving scheduling, a necessary and sufficient condition for bounded response times is that

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1. \tag{4}$$

In effect, violating this requirement would mean requiring more than 100% utilization of the processor.

However, we also care about the variability of response times. The response time R_i is the time between when an interrupt request in stream i is asserted and the time the interrupt has been handled. This includes the execution time of the ISR, so $R_i \geq C_i$, where C_i depends on the program flow in the ISR. We assume that this execution time is either constant (which it can be if the ISR is simple), or that we can determine the WCET $\bar{C}_i$ of the ISR. If we use a PRET machine, we can make the execution time of the ISR constant using a deadline instruction, thereby reducing the variability in response times to the variability in $R_i - \bar{C}_i$.

Before we see how a PRET machine reduces the variability of response times, let us study the variability that a conventional machine experiences with the scenario of two sporadic interrupt request streams. Assuming non-preemptive execution of the ISRs, the response time R_1 for requests from stream 1 may include C_2, the execution time for handling requests from stream 2. With sporadic inputs, it is always possible to get almost simultaneous interrupt requests from the two streams, but one must be handled first, thereby delaying the other by the execution time of the first. This means that the *variability* in the response time R_i for each stream is at least the execution time for handling *the other* stream.

If interrupts are enabled during the handling of a request, then the variability in handling requests is more complicated to determine. Consider the handling of an interrupt from stream 1, and assume it takes time C_1 (without interruption) to handle the request. Then with interruption, the actual time could be C_1 (if there is no occurrence of stream 2 during the handling), $C_1 + C_2$ (if there is one occurrence of stream 2 during the handling), $C_1 + 2C_2$ (if there are two occurrences), etc. The worst case will be $C_1 + mC_2$, where

$$m = \left\lceil \frac{C_1}{T_2 - C_2} \right\rceil.$$

In the worst case, in stream 2, interrupt requests are occurring as fast as possible, once every T_2 time units, and when the request occurs, it is immediately handled, preempting the handling of a request from stream 1. In that case, in each interval of length T_2, the time available to execute the ISR for stream 1 is $T_2 - C_2$ (neglecting context-switch overhead). Hence, the number of times that the ISR for stream 1 may be interrupted is m as shown.

In both cases, whether interrupts are disabled or not, the variability in the time it takes to handle any one interrupt request includes at least the time for one execution of a request from the other stream, possibly more than one execution. Even if the execution time of each ISR is constant, the response time will be highly variable and dependent on other handlers. This makes software design non-modular, since a change in the

handling of one stream affects the behavior of the handling of the other stream. And the variability will typically be a large multiple of a clock cycle, typically hundreds or thousands.

Given two sporadic streams, we could reduce the variability of one of those streams by disabling interrupts while it is handled but not disabling interrupts while the other is handled. Here, we can reduce the variability in handling time for that one stream to a small constant multiple of a clock period. But this is also non-modular because it requires that there be no more than one critical interrupt in an entire system design and no other code blocks that disable interrupts. Every other interrupt will suffer high variability. Moreover, every other interrupt will have to be handled with nested interrupts enabled, which means that execution-time analysis must be more conservative. This reduces processor utilization because the system must be designed for worst-case disruptions of timing, even if those disruptions are unlikely. Equivalently, this solution requires more overprovisioning (faster processors, more energy) to provide guarantees.

B. PRET Machine Interrupt Handling

Abstract PRET machines eliminate this variability. We can have a multiplicity of sporadic interrupt streams each of which is handled with variability that is a small multiple of the clock period and is independent of the execution times of the other handlers. More interestingly, this can be done with no loss in performance or utilization, so we can operate right up to bound (4), and sometimes beyond that bound, as we will show below. Our APM assumes two possible interrupt mechanisms, described in each of the next two subsections.

1) Interrupt Handlers in Hard-Real-Time Threads: The first mechanism is the simplest to analyze but has more limitations. In this first mechanism, interrupts become enabled when an HRT stalls to wait for an interrupt request. While it is stalled, its scheduled cycles can be used by SRTs. For the scenario of multiple sporadic interrupt streams, the simplest strategy is to assign one HRT thread to each sporadic stream. One limitation of this approach, therefore, is that we cannot handle more sporadic interrupt streams than there are hardware threads.

In this case, the execution time $C(p)$ of the interrupt handler is the time between when the HRT resumes (an interrupt request has unblocked it) and the time when it stalls again to wait for another interrupt. $C(p)$ depends on the duty cycle p of the HRT that the interrupt handler is mapped to. Just as with a conventional processor, we will require that $C(p) \leq T$, otherwise no real-time guarantee is achieved. But unlike a conventional processor, there is almost no variability in the time it takes to handle an interrupt. If the execution time is constant, then the variability is no larger than the maximum number of clock cycles between instruction issues of the HRT's schedule. This is typically just a few clock cycles, so very small indeed.

The variability in the time it takes to handle an interrupt has no dependence on the time it takes to handle any other interrupt in the system. However, the *magnitude* of the time it takes to handle the interrupt does have such a dependence. This is because PRET machines cannot perform magic. They cannot provide guarantees that require greater than 100% processor utilization. Hence, the instruction-issue schedule for an HRT must allow enough cycles for any other interrupt handlers that might be simultaneously active to also be scheduled. We emphasize, however, that these cycles are not wasted. They will be used by an SRT if not needed by an HRT, but even more fundamentally, they represent resources that even a conventional processor has to provide to achieve comparable timing bounds. PRET machines do not sacrifice performance compared to conventional machines, as we prove below.

Let us examine this dependence more closely. We require that an HRT handling an interrupt stream with period T be assigned a schedule with a duty cycle p, such that $C(p)$ is no greater than T. This schedule needs to leave enough idle cycles that the same constraint can be satisfied for every other HRT handling a sporadic interrupt stream. Otherwise, the processor could find itself in a situation where greater than 100% utilization is required to meet the timing constraints. It is therefore useful to determine the smallest duty cycle p_i sufficient to process interrupt stream i within its period T_i:

$$p_i = \min\{p \mid C_i(p) \leq T_i\} \tag{5}$$

In a real PRET machine, limitations of the instruction dispatch hardware will constrain what values of p are possible. But with sufficient hardware, we can come as close as we like to the APM ideal, so we go ahead and make this assumption.

To schedule n sporadic streams with periods $T_1, \ldots, T_n$, execution times $C_1, \ldots, C_n$, and duty cycles $p_1, \ldots, p_n$ determined by Equation (5), we require that

$$\sum_{i=1}^{n} p_i \leq 1. \tag{6}$$

How does this constraint relate to the bound in Equation (4)? Our claim is that it is actually weaker than that bound. In other words, the satisfaction of (4) implies the satisfaction of (6) but not necessarily vice versa. Consequently, if a conventional processor can deliver bounded response times, i.e. (4) is satisfied, then so can an APM, i.e. (6) is satisfied. Hence, when we require bounded response times, APMs give up no performance and almost entirely eliminate variability. The price for this is an increase in average response times and a modest increase in hardware cost.

To prove this statement, assume (4) holds and let p'_i be C_i/T_i. Due to (2), we obtain $C_i(p'_i) \cdot p'_i = C_i(p'_i) \cdot C_i/T_i \leq C_i$, which implies $C_i(p'_i) \leq T_i$. By the definition of p_i in (5) we have that $p_i \leq p'_i$. Because (4) holds, $\sum_{i=1}^{n} \frac{C_i}{T_i} = \sum_{i=1}^{n} p'_i \leq 1$, which directly implies that $\sum_{i=1}^{n} p_i \leq 1$ and thus (6) also holds.

Even more interestingly, this style of interrupt handling has the potential to *improve* performance compared to conventional interrupt handling. In a conventional scheme, one ISR runs at a time. This means that in each clock cycle, the instruction issued belongs to the same instruction stream of the instruction issued in the previous cycle. Consequently, this instruction stream will suffer a performance loss due to pipeline bubbles. For example,

a conditional branch instruction will have to stall until the branch condition is evaluated, or worse (for variability), the branch would be speculatively taken or not taken depending on the state of a branch predictor. Whenever thread interleaving reduces the number of pipelines bubbles, the necessary duty cycle p_i to process an sporadic interrupt stream in time will be strictly smaller than the ratio $p'_i = \frac{C_i}{T_i}$. In such cases, (6) is strictly weaker than (4). We also note that (4) is based on the assumption of zero context switch time, which is unrealistic on conventional processors, but a key asset of PRET machines.

Moreover, because of the potential for pipeline bubbles, WCET analysis for conventional interrupt handling is likely to be pessimistic, further reducing performance because fewer streams are safely executable on a processor. This situation gets even worse if interrupts are enabled during the handling of requests, which may improve responsiveness but can also cause timing analysis to provide looser bounds [6]. For safety-critical systems, having tighter bounds on WCET must be viewed as a performance improvement.

An APM need not suffer this performance loss nor the variability due to pipeline bubbles. If the HRT schedule is sufficiently sparse, then no pipeline stall will ever be needed. Notice that the HRT schedule *is required to be sparse* if there are multiple sporadic interrupt streams, otherwise we exceed full utilization. Hence, an APM performs even better for complex applications than for simple ones. It can come closer to 100% utilization of the pipeline with essentially no variability. Liu et al. demonstrated this improved performance on a multicore implementation of the PTARM (second generation Berkeley PRET machine) by realizing a real-time computational fluid dynamics simulation [34]. The problem is embarrassingly parallel, so they were able to keep 55 cores each with four hardware threads, hence a total of 220 threads, continuously operating with no pipeline bubbles. Moreover, because of the precision timing of the PRET machine, they were able to eliminate synchronization overhead for interprocessor communication, so nearly every cycle was performing floating-point arithmetic. Each machine had a fixed round-robin schedule of four hardware threads.

2) Interrupt Handlers in Soft-Real-Time Threads: A second interrupt handling mechanism in our APM has more flexibility at the expense of more difficult analysis. In this second mechanism, interrupts are always handled by an SRT, which implements conventional interrupts. This means, of course, that an APM can always implement whatever I/O policy you would implement on a conventional machine, so trivially an APM performs no worse than a conventional machine. But APMs offer more interesting possibilities that a conventional machine does not have. One observation is that it becomes unnecessary for a program to ever disable interrupts in order to bound response times because bounded response times will be delivered by an HRT. In this case, the only mechanism by which interrupts become disabled is to dedicate 100% of the cycles to HRTs.

Assuming programs do not disable interrupts, then when an interrupt request is raised, it is handled at the very next available SRT dispatch cycle, preempting any SRT that might have otherwise used that cycle. Consequently, the latency to begin handling an interrupt is bounded by the maximum time between idle dispatch cycles not occupied by active HRTs. The exact value of this bound will depend on the schedule for HRTs, but we can easily approximate it if we assume that the unused cycles are evenly spaced. In this case, the bound will be $u_H(t)/(1 - u_H(t))$, where $u_H(t)$ is the HRT utilization given in (1).

Because the interrupt is handled in an SRT, if we need bounded response time, the ISR should delegate that handling to an HRT. The ISR may, for example, activate a blocked HRT. If all it ever does is delegate each interrupt to an HRT that is assumed to be stalled waiting for such delegation, then this mechanism is equivalent to the first mechanism, but with a small additional overhead of performing the delegation. This overhead is bounded as long as $u_H(t) < 1$, though calculating this bound could be complicated by the requirement to consider any pipeline bubbles that occur during the execution of the ISR in an SRT prior to the ISR setting the bit that enables the HRT. If the ISR is kept short, however, and quickly enables the HRT, then the variability introduced will be limited to a few clock cycles, many orders of magnitude smaller than the variability introduced by conventional interrupt handling.

This second mechanism is much more flexible than the first because the ISR can use priorities, criticality, and other application-specific requirements, together with the state of the machine (which and whether HRTs are stalled, for example) to make decisions about how to handle the interrupt. Moreover, this second mechanism can easily realize a scenario where there are more sporadic interrupt streams than there are hardware threads, although doing so will increase the variability of at least some of the interrupt handlers.

C. Deadlines

One issue is that we may want interrupt handling for a sporadic stream with period T to have a deadline D. That is, we require the response time R to be no larger than some value D. A simple case is $D = T$, where we require that an interrupt be handled before the earliest time the next request from the same stream may arrive. If $D = T$, the constraints (4) and (6) are necessary and sufficient to meet the deadline. But we may require $D < T$, which asserts that even rare events should get quick responses. In such situations, (4) and (6) are generally not sufficient to guarantee that all deadlines are met.

In 1974, Dertouzos [35] showed that Earliest Deadline First (EDF) scheduling is optimal among preemptive scheduling algorithms for conventional single-core processors. If a set of interrupt requests can feasibly be scheduled, then an algorithm that always schedules one of the requests with the nearest deadline will generate a feasible schedule. Like most optimality results, this result depends on the assumption that context switches have zero cost, which is unrealistic on conventional processors. Nevertheless, we will conduct the comparison between conventional and PRET interrupt handling using this assumption.

In 1990, Baruah et al. [36] gave necessary and sufficient conditions for sporadic task sets to be feasible. As EDF is optimal, these conditions also answer whether or not all deadlines will be met under EDF scheduling. Baruah et al.'s analysis is based on the concept of *demand bound functions*. The demand bound function $dbf_i(t)$ is the largest cumulative execution requirement of all interrupt requests that can be generated by sporadic stream i, which have both their arrival times and their deadlines within a contiguous interval of length t. For the sporadic model we have adopted, $dbf_i(t)$ can be determined as follows:

$$dbf_i(t) = \max(0, \left\lfloor \frac{t + T_i - D_i}{T_i} \right\rfloor \cdot C_i) \tag{7}$$

A set of sporadic streams can feasibly be scheduled if the cumulative execution requirement of all streams does not exceed the capacity of the processor for any interval length:

$$\forall t > 0 : \sum_{i=0}^{n} dbf_i(t) \leq t \tag{8}$$

Under which conditions can PRET machine interrupt handling guarantee that all deadlines will be met? We focus on the case where each interrupt is assigned a dedicated HRT thread, as in Section IV-B1, which ensures minimal variability in response times. Then, the duty cycle p_i^D of the HRT assigned to interrupt stream i needs to be sufficiently high to guarantee that $C_i(p_i^D)$ is less than the deadline D_i. Thus the smallest possible duty cycle is given by:

$$p_i^D = \min\{p \mid C_i(p) \leq D_i\} \tag{9}$$

To schedule n sporadic streams with deadlines $D_1, \ldots, D_n$, execution times $C_1, \ldots, C_n$, and duty cycles $p_1^D, \ldots, p_n^D$ determined by Equation (9), we require that

$$\sum_{i=1}^{n} p_i^D \leq 1. \tag{10}$$

How does this constraint relate to the bound in Equation (8)? Is there a price to pay for deterministic response times? It turns out that the two constraints are incomparable. That is, (10) does not imply (8) nor vice versa.

To see this, note that $D = T$ is a special case for which we have shown that if a conventional processor can meet the deadlines, then an APM can also do so with deterministic response times. But it turns out that there are circumstances in which a conventional processor can meet deadlines, but no machine can meet the deadlines with deterministic response times. In such circumstances, to meet the deadlines, we are forced to use SRT interrupts in an APM and forgo at least some deterministic response times.

Consider an example consisting of two interrupt streams where first interrupt stream is characterized by $C_1(p) = 9/p$, $D_1 = 9$, and $T_1 = 1000$, and the second by $C_2(p) = 1/p$, $D_2 = 10$, and $T_2 = 10$. To meet the deadline of the first interrupt stream, a duty cycle of $p_1 = 1$ is required. If this stream is assigned an HRT, then no other HRT can be allocated without violating (4). Hence, the second stream will have to be serviced by an SRT and will suffer nondeterministic response times. Deadlines will be met, but not both with deterministic response times.

The two interrupt streams can also be feasibly scheduled using conventional preemptive scheduling, which can be seen by evaluating (10) or by using fixed-priority scheduling with the first stream assigned a higher priority than the second. The response time of the first stream will always be 9. Any single interrupt request of stream 2 may conflict with at most one interrupt request of stream 1 due to the first stream's high minimum inter-arrival time. Thus, the second stream's response time will vary considerably, between 1 and 10, but it will always meet its deadline. In this case, the behavior of the APM and the conventional machine match, and no benefit is derived from the APM.

The above example demonstrates that there is a tradeoff between achieving deterministic response times and meeting deadlines when deadlines are shorter than periods. In the extreme case shown in the example, an interrupt request may temporarily require *all* cycles of a processor. However, due to its high period, this does not preclude other interrupt streams from meeting their deadlines, but at the expense of variable response times. This tradeoff probably deserves more study. In more complex scenarios with mixed criticality and more sporadic streams to handle, hybrid scheduling approaches that combine the benefits of a reduced number of pipeline bubbles afforded by PRET machines with the flexibility of conventional preemptive scheduling may prove beneficial. Is there a scheduling algorithm that minimizes the variability of response times while being optimal w.r.t. meeting deadlines? We also note that PRET machine scheduling is similar to P-fair scheduling [37], which should be investigated further.

The above comparison considers preemptive scheduling under the assumption of zero context-switch costs, which is unrealistic on modern processors featuring stateful resources such as caches and branch predictors. A more refined analysis would either consider a model of context-switch overhead or study non-preemptive scheduling.

V. The (Surprising) Benefits of Concurrency

Oddly, PRET machines seem to benefit from concurrency. The more concurrent operations to be accommodated, the more likely we can eliminate pipeline bubbles. PRET machines also make programs more modular because it is easier to isolate concurrent behaviors from one another. With conventional interrupts, for example, deterministic timing is achievable for exactly one highest-priority responder. All others will be nondeterministic. Moreover, the design is less modular because the response times of all but one responder depend on the executions time of other responders.

Since APMs have SRTs, we trivially lose nothing with APM compared to a conventional design. An APM can always be configured to have exactly one active thread, an SRT, in which case, it *is* a conventional design. The only cost is a modest amount of unused hardware. But as the complexity

of applications increases, APMs open the possibility of much greater determinism and modularity. Moreover, APMs can mix conventional designs with deterministic hard-real-time responders. An SRT is capable of anything a conventional design can do, but not vice versa.

VI. A Commitment to Determinism

All of engineering is built on models. For the purposes of this paper, we will define a "model" of a system to be any description of the system that is not Kant's thing-in-itself (*das Ding an sich*). Every model rests on a modeling paradigm. A programming language, for example, is just such a modeling paradigm. What constitutes a well-formed program is well defined, as is the meaning of the execution of such a program. The program is a model of what a machine does when it executes the program. Synchronous digital circuits constitute another such modeling paradigm. They model what an electronic circuit does. Models abstract away details, and layers of models may be built one on top of another.

Properties of the modeling paradigm are fundamental when an engineer builds confidence in a design. A synchronous digital circuit, as a model, realizes a deterministic function of its input. A single-threaded program is also a deterministic function of its inputs. The determinism of these modeling paradigms is assumed without question. Without such determinism, we would not have billion-transistor chips and million-line programs handling our banking.

The timing exhibited by a program is not specified in the model (the program). Whether an execution of the program is correct does not depend on the timing, so this model permits implementations with arbitrary timing. Nevertheless, we assert that the model (the single-threaded program) is deterministic because the model does not include timing in its notion of the behavior of the program. Hence, within the modeling paradigm of a program, deterministic timing is not achievable.

A model can only predict aspects of behavior that lie within its modeling paradigm. Our essential claim in this paper is that we should make a commitment to using models that include aspects of behavior that we care about. If we care about timing, we should use models that *do* include timing in their notion of behavior. Today, with real-time systems, we do not do that.

Instead, today, timing properties emerge from a physical implementation. When we map a particular program onto a particular microprocessor, a real physical chip embedded in a real board, with real memory chips and peripherals sharing the bus, only then do we get timing properties. Timing is a property of the thing-in-itself not of the model. It emerges from the implementation.

What about adaptability, resilience, and fault tolerance? Any cyber-physical system will face the reality of unexpected behaviors and failures of components. Using deterministic models does not prevent us from making fault-tolerant and adaptive systems. On the contrary, it *enables* it. A deterministic model defines unambiguously what a *correct* behavior is. This enables detection of *incorrect* behaviors, an essential prerequisite to fault-tolerant adaptive systems.

PRET machines offer a deterministic temporal model and are capable of interrupt-driven I/O that does not disrupt the timing of timing-critical tasks. We believe that PRET machines will eventually become widely available because their benefits to safety-critical systems are enormous and their performance is competitive with conventional architectures. They deliver *repeatable* behavior, where the behavior in the field is assured of matching their behavior on the test bench with extremely high precision and high probability (at the same level of confidence as we currently get from synchronous digital logic circuits). In our expectation, it is just a matter of time before the world accepts the paradigm shift that they entail.

VII. Open Issues

PRET machines make a commitment to deterministic timing. The APM model shows that they deliver drastic reductions in timing variability with no loss of performance. The FlexPRET implementation shows that they can be realized at modest hardware cost. What problems remain?

First, software support is needed for constructing PRET applications and sharing resources across applications. One immediate issue is that applications that require multiple HRTs would benefit enormously from compiler and operating system support to optimize the construction of their static instruction-issue schedules. Currently, we construct these schedules by hand, a viable approach only when the number of HRTs is small and when their timing constraints are not dynamically varying. A particularly interesting challenge would be to synthesize schedules on the fly to satisfy real-time constraints specified in terms of the MTFD instruction.

Second, operating systems will have to be developed that exercise admission control to prevent dynamically instantiated HRTs from disrupting the timing behaviors of other HRTs. Every PRET machine (and indeed, every machine) will have performance limitations, but for PRET machines, these limitations are clear and precise, so stark distinctions can be made between allowed and not-allowed behaviors. Applications can be guaranteed temporal isolation from one another, an essential feature for safety-critical applications [30].

Third, PRET machines offer the opportunity for substantial reductions of energy consumption. Programs specify their required temporal behavior and yield to precise timing analysis. As a consequence, once a temporal behavior has been determined to be acceptable for a given application, the clock frequency and voltage can be reduced to the point where the specified timing behavior is just barely met. This contrasts with today's situation, where imprecise timing analysis forces overprovisioning, where the substantial headroom that is required translates directly into increased energy consumption.

Fourth, there are myriad opportunities for software methodologies that take advantage of temporal semantics. For example, programming models such as PTIDES [38] make the timing requirements of concurrent programs explicit and could be used to build safety-critical PRET applications. These models need better developed language and operating system support.

VIII. Conclusion

Abstract PRET machines represent a family of microprocessor architectures with a deterministic model of temporal behavior. They combine hard-real-time threads that have well-defined timing properties with conventional soft-real-time threads, sharing resources in a balance determined by the application. Cycles that are unused by HRT threads fall over to SRT threads, so the cycles are not wasted.

Hardware support for multithreading leads to the interesting property that applications become more deterministic as concurrency increases, dramatically the opposite of what we experience with conventional processors. With enough concurrency, pipeline bubbles and memory latencies become irrelevant, the timing of execution of threads becomes regular and predictable, the variability in the response to interrupts drops by orders of magnitude, and processor utilization can approach 100%. Since prior work has shown that the hardware cost of realizing PRET machines is modest, the only argument against PRET machines is that we do not yet know how to write software that fully takes advantage of the newfound determinism. This needs to be the next research agenda.

References

[1] J. A. Stankovic, "Misconceptions about real-time computing: a serious problem for next-generation systems," *Computer*, 21(10) pp. 10–19, 1988.

[2] S. A. Edwards, S. Kim, E. A. Lee, I. Liu, H. Patel, and M. Schoeberl, "A disruptive computer design idea: Architectures with repeatable timing," in *Int. Conf. on Computer Design (ICCD)*. IEEE, 2009, pp. 54–59.

[3] H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Transactions of the American Mathematical Society*, 74(2) pp. 358–366, 1953.

[4] J. Nowotsch and M. Paulitsch, "Leveraging multi-core computing architectures in avionics," in *2012 Ninth European Dependable Computing Conference, Sibiu, Romania, May 8-11, 2012*, 2012, pp. 132–143.

[5] 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. Stenstr, "The worst-case execution-time problem - overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, pp. 1–53, 2008.

[6] J. Schneider, "Cache and pipeline sensitive fixed priority scheduling for preemptive real-time systems," in *Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS 2000), Orlando, Florida, USA, 27-30 November 2000*, 2000, pp. 195–204.

[7] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, 2nd ed. Springer, 2005.

[8] L. Thiele and P. Kumar, "Can real-time systems be chaotic?" in *EMSOFT*. ACM, 2015, pp. 21–30.

[9] 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*, 2012.

[10] S. A. Edwards and E. A. Lee, "The case for the precision timed (PRET) machine," in *Design Automation Conf. (DAC)*, 2007.

[11] R. Smith, "ARM announces the Cortex-R52 CPU: Deterministic & safe, for ADAS & more," 2016, AnandTech Blog.

[12] B. D. de Dinechin *et al.*, "Time-critical computing on a single-chip massively parallel processor," in *DATE*, 2014, pp. 1–6.

[13] M. Schoeberl, "A time predictable Java processor," in *DATE*, 2006, pp. 800–805.

[14] M. Schoeberl and P. Puschner, "Is chip-multiprocessing the end of real-time scheduling?" in *9th Int. Workshop on Worst-Case Execution Time (WCET) Analysis*. OCG, 2009.

[15] S. Hahn, J. Reineke, and R. Wilhelm, "Toward compact abstractions for processor pipelines," in *Correct System Design*, R. Meyer, A. Platzer, and H. Wehrheim, Eds., 2015, pp. 205–220. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-23506-6_14

[16] T. Lundqvist and P. Stenström, "Timing anomalies in dynamically scheduled microprocessors," in *RTSS*, 1999, pp. 12–21.

[17] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker, "A definition and classification of timing anomalies," in *WCET*, 2006.

[18] S. Andalam, P. S. Roop, and A. Girault, "Predictable multithreading of embedded applications using PRET-C," in *Formal Methods and Models for Codesign (MEMOCODE)*. IEEE/ACM, 2010, pp. 159–168.

[19] P. Puschner, D. Prokesch, B. Huber, J. Knoop, S. Hepp, and G. Gebhard, "The T-CREST approach of compiler and WCET-analysis integration," in *Int. Symp. on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, 2013.

[20] S. A. Seshia and J. Kotker, "GameTime: A toolkit for timing analysis of software," in *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS),*, 2011.

[21] P. Wägemann, T. Distler, C. Eichler, and W. Schröder-Preikschat, "Benchmark generation for timing analysis," in *Real-Time Embedded Technology and Applications Symp. (RTAS)*. IEEE, 2017.

[22] J. Reineke and J. Doerfert, "Architecture-parametric timing analysis," in *20th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2014, Berlin, Germany, April 15-17, 2014*, April 2014, pp. 189–200. [Online]. Available: http://embedded.cs.uni-saarland.de/publications/ArchitectureParametricTimingAnalysis-RTAS2014.pdf

[23] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee, "Predictable programming on a precision timed architecture," *Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2008.

[24] I. Liu, J. Reineke, D. Broman, M. Zimmer, and E. A. Lee, "A PRET microarchitecture implementation with repeatable timing and competitive performance," in *Int. Conf. on Computer Design (ICCD)* 2012, pp. 87–93.

[25] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee, "PRET DRAM controller: Bank privatization for predictability and temporal isolation," in *Int. Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2011, pp. 99–108.

[26] M. Zimmer, D. Broman, C. Shaver, and E. A. Lee, "FlexPRET: A processor platform for mixed-criticality systems," in *Real-Time and Embedded Technology and Application Symp. (RTAS)*, 2014.

[27] M. Zimmer, "Predictable processors for mixed-criticality systems and precision-timed I/O," EECS, UC Berkeley, Report UCB/EECS-2015-181, August 10 2015, PhD Thesis. [Online]. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-181.html

[28] 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," in *Real-Time and Embedded Technology and Application Symp. (RTAS)*, 2015.

[29] N. J. H. Ip and S. A. Edwards, "A processor extension for cycle-accurate real-time software," in *Int. Conf. on Embedded and Ubiquitous Computing (EUC)*, vol. LNCS 4096. Springer, 2006, pp. 449–458.

[30] D. Bui, E. A. Lee, I. Liu, H. D. Patel, and J. Reineke, "Temporal isolation on multiprocessing architectures," *Design Automation Conf. (DAC)* 2011.

[31] E. W. Dijkstra, "The humble programmer," *Communications of the ACM*, vol. 15, no. 2, pp. 859–866, 1972.

[32] J. V. Busquets-Mataix *et al.*, "Adding instruction cache effect to schedulability analysis of preemptive real-time systems," in *RTAS*, June 1996, pp. 204–212.

[33] C.-G. Lee *et al.*, "Analysis of cache-related preemption delay in fixed-priority preemptive scheduling," *IEEE Transactions on Computers*, vol. 47, no. 6, pp. 700–713, 1998.

[34] I. Liu, E. A. Lee, M. Viele, G. G. Wang, and H. Andrade, "A heterogeneous architecture for evaluating real-time one-dimensional computational fluid dynamics on FPGAs," in *Int. Symp. on Field-Programmable Custom Computing Machines (FCCM)*, 2012.

[35] M. L. Dertouzos, "Control robotics :the procedural control of physical processes," *Proceedings IFIP Congress*, 1974. [Online]. Available: http://ci.nii.ac.jp/naid/80013086160/en/

[36] S. K. Baruah, A. K. Mok, and L. E. Rosier, "Preemptively scheduling hard-real-time sporadic tasks on one processor," in *Proceedings 11th Real-Time Systems Symposium*, Dec 1990, pp. 182–190.

[37] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, no. 6, pp. 600–625, Jun 1996. [Online]. Available: https://doi.org/10.1007/BF01940883

[38] J. Eidson, E. A. Lee, S. Matic, S. A. Seshia, and J. Zou, "Distributed real-time software for cyber-physical systems," *Proceedings of the IEEE (special issue on CPS)*, vol. 100, no. 1, pp. 45–59, 2012.