A PRET Architecture Supporting Concurrent Programs with Composable Timing Properties

Isaac Liu, Jan Reineke, and Edward A. Lee
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA 94708-1770
Email: {liuisaac, reineke, eal}@eecs.berkeley.edu

***Abstract*—In order to improve design time and efficiency of systems, large scale system design is often split into the design of separate functions, which are later integrated together. For real time safety critical applications, the ability to separately verify timing properties of functions is important. If the integration of functions on a particular platform destroys the timing properties of individual functions, then it is not possible to verify timing properties separately. Modern computer architectures introduce timing interference between functions due to unrestricted access of shared hardware resources, such as pipelines and caches. Thus, it is difficult, if not impossible, to integrate two functions on a modern computer architecture while preserving their separate timing properties. This paper describes a realization of PRET, a class of computer architectures designed for timing predictability. Our realization employs a thread-interleaved pipeline with scratchpad memories, and has a predictable DRAM controller. It decouples execution of multiple hardware contexts on a shared hardware platform, which allows for a straight forward integration of different functions onto a shared platform.**

I. Introduction

Embedded systems interact with the physical environment. They react based on sensing of the physical world and often have to satisfy timing constraints imposed by their interaction with the environment. Hard real-time systems are an important class of embedded systems that exhibit safety-critical timing properties: a violation of timing constraints in the system could lead to catastrophic events. Examples of such systems include aircraft flight control systems and automotive engine control units.

This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #0720882 (CSR-EHS: PRET) and #0931843 (ActionWebs), the U. S. Army Research Office (ARO #W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312), the Air Force Research Lab (AFRL), the Multiscale Systems Center (MuSyC), one of six research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program, and the following companies: Bosch, National Instruments, Thales, and Toyota.

Sangiovanni-Vincentelli et al. [1] outline several challenges as the designs continue to scale up for these systems. Among those challenges, this paper deals with *timing composability* and *timing predictability* at the computer architecture level. *Timing composability* is the ability to integrate components while preserving their temporal properties. *Timing predictability* is the ability to predict timing properties of the system.

A. Timing Composability

Large-scale system design relies on the ability to design and verify components separately, and to later integrate them. If component properties may be destroyed during integration, then the components can no longer be designed and verified separately. In order to preserve component properties during integration, a *federated architecture* is often used. In a federated architecture, every major function is implemented separately on a dedicated hardware unit, often called *electronic control unit* (ECU). As these ECUs are only loosely coupled through an interconnect, interference is limited, preserving certain properties which are independently verified. However, since each ECU is only executing a single function, they are idle for most of the time. In order to reduce resource consumption, there is a shift towards *integrated architectures*, where multiple functions are integrated on a single, shared hardware platform. The challenges to switch from a federated to an integrated architecture are considerable, as noted in [2], [3]. In particular for hard real-time systems, it is crucial to guarantee that the timing properties are preserved during system integration. Modern architectures that allow unrestricted sharing of resources cause unpredictable interference between the components. This hinders our ability to compose functions together on a shared resource while maintaining timing properties. Caches are a well-known example of an unpredictable shared resource that can cause dramatic effects on execution time.

B. Timing Predictability

Several researchers have outlined the importance, requirements and difficulties of designing timing-predictable systems [4], [5], [6]. Modern computing abstraction layers have abstracted away time. Common programming languages such as C or Java do not associate any timing semantics with a program; a correct execution of a C or Java program has nothing to do with how long it takes to run. Thus, when designing systems where timing needs to be guaranteed, an additional analysis is needed to determine the worst-case execution time (WCET) in order to verify that the timing constraints can be met.

However, it is difficult to analyze the execution time of a program. Wilhelm et al. [7] describe abundant research and effort that has been put into determining the WCET of a program. The precision and usefulness of the analysis heavily depends on the predictability of the underlying architecture [8], [9]. Conventional architectures have introduced caches and prediction and speculation units that improve average-case execution time (ACET). Such features do not, in general, improve WCET, yet their complex dynamic behavior makes it is extremely difficult, if not practically impossible, to obtain precise bounds on the execution time on modern architectures.

C. Contribution

The key challenges we help overcome are the difficulty of designing timing-predictable systems and the difficulty of integrating functions when designing complex large-scale systems. Edwards and Lee [10] proposed a paradigm shift in the design of computer architectures, focusing on timing predictability instead of average-case performance. In this paper, we review the goals of that work, and outline progress in an ongoing project at Berkeley toward the goals. We describe a realization of PRET that is designed for timing predictability, to enable simple architectural timing analysis for each context. This realization provides interference-free concurrent execution of multiple contexts to allow for simple and efficient integration of multiple independent functions.

D. Related Work

Several researchers have proposed modifications to modern computer architectures that improve timing predictability with some average-case performance penalty [11], [12], [13], [14], [15], [16], [17], [18]. Yan and Zhang [11] propose modifications to a VLIW architecture to make instructions execute in constant time. They extend the VLIW compiler to support full if conversion, hyperblock scheduling, and intra-block nop insertion to enable efficient WCET analysis. Rochange and Sainrat [14] propose modifications to a dynamic superscalar pipeline by stalling instructions between basic blocks. They make the timing of basic blocks independent of one another, by stalling instruction fetching of a basic block until all instructions in the previous basic block have been completed. Whitham and Audsley [12] describe similar work with a superscalar architecture, combining it with trace scheduling to direct the processor branch predictions towards the worst-case path. Uhrig and Maier [13] and Barre et al. [16] both modify a conventional simultaneous multi-threaded architecture. They make the timing of one thread independent of the behavior of other threads by assigning it the highest priority. That one thread thus has priority when any resource contention occurs, simplifying its timing analysis. However, there are no guarantees for the other hardware threads in the processor. El-Haj-Mahmoud and Al-Zawawi [15] propose a processor architecture that can be partitioned into a set of virtual processors. The timing of these virtual processors is independent of each other providing composable timing to tasks running on the different virtual processors. The partitioning of the architecture is flexible. The architecture can be partitioned into a few higher-performance processors or many simple low-performance processors or a combination of the two extremes. Schoeberl [17] introduces JOP, the Java Optimized Processor which is a timing predictable Java processor. It uses a two-level stack architecture along with fixed-length microcode into which the Java byte code is translated. Every microcode instruction has a fixed execution time independent of its context. Hansson et al. [18] proposed CoMPSoC, which is a template for composable multi-processor systems on chip. They also introduce the CoMPSoc design methodology in which real-time requirements are described per application on a level that is understood by the developer. The particular configuration of the system, including the network on chip, is then derived automatically from these requirements. Wilhelm et al. [9] give recommendations for future architectures in time-critical embedded systems. Based on the principle *to reduce the interference on shared resource*, they recommend to use caches with LRU replacement, separate instruction and data caches, and so-called *compositional* architectures, such as the ARM7.

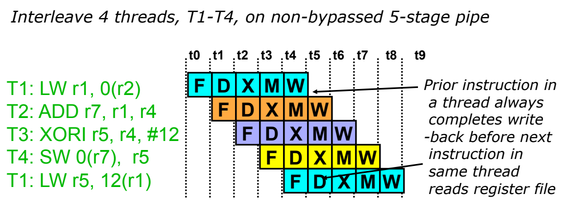
Akesson et al. [19] and later Paolieri et al. [20] proposed DRAM controllers with predictable and composable timing behavior. They decouple execution time of memory accesses by devising DRAM access patterns which can be executed independently of previous memory accesses. In addition they make use of predictable arbitration mechanisms such as TDMA or latency-rate servers to share the DRAM in systems with multiple clients.

The work presented in this paper looks to redesign the architecture with timing predictability as the main focus, only introducing performance improvements when predictability is not sacrificed.

II. Precision Timed Machine

Computer architects have gone to great lengths to improve the average-case performance of their architecture. Features such as caches and branch predictors are common schemes which greatly improve performance on the average. These schemes predict the dynamic behavior of a program to avoid stalling the processor. However, such schemes can have a detrimental effect on the worst-case execution time: in case of a misprediction the execution time is often greater than if no prediction was made in the first place. Determining within a worst-case execution time analysis whether or not a misprediction occurs, however, is often extremely difficult. The prediction depends on the internal state of the prediction unit, which is determined by its execution history. In case of a cache, the execution history is the sequence of memory accesses that have been performed since the cache was turned on. The execution history may thus contains memory accesses of previously executed tasks, and, in case of a shared cache on a multi-core system, of memory accesses by concurrently running tasks. Similarly, the execution history of a branch prediction unit is the sequence of branches that have or have not been taken. Even if the execution history is determined by the task under analysis only, static analysis of branch prediction and caches is hard, as the number of possible execution histories explodes with increasingly complex program flow. If the execution history depends on concurrently executing tasks, as in multi-core systems, or on other tasks, as in premptive multi-tasking, analysis becomes even more challenging.

A common side effect of having multiple prediction units that introduce variable execution latency is a phenomenon known as timing anomalies [21], [22]. Timing anomalies are situations where a local worst-case behavior does not result in the global worst-case execution time. For example, a cache miss should intuitively result in a longer execution time than a cache hit. This is not always the case. For instance, if the cache miss allows the processor to resolve a conditional branch that it would otherwise mispredict. Branch misprediction would not only entail a miss penalty in itself, it can also result in further memory accesses negatively influencing the state of the cache. This phenomenon makes analyzing worst-case execution time extremely difficult if not practically infeasible for some architectures, as analyses have to consider all possibilities, i.e. cache hit and cache miss, in order to be correct.



Interleave 4 threads, T1-T4, on non-bypassed 5-stage pipe
t0 t1 t2 t3 t4 t5 t6 t7 t8 t9
T1: LW r1, 0(r2)
T2: ADD r7, r1, r4
T3: XORI r5, r4, #12
T4: SW 0(r7), r5
T1: LW r5, 12(r1)
F D X M W
Prior instruction in a thread always completes write-back before next instruction in same thread reads register file

Fig. 1. An example of interleaved threading removing data dependencies; reproduced from a presentation by Krste Asanović.

A. Thread-Interleaved Pipelines

Pipelines improve the throughput of instructions executed on a processor by overlapping execution of successive instructions. However, pipeline hazards caused by instruction dependencies force the pipeline to stall. Common architectures add hardware units in attempt to predict branches or prefetch data to avoid later stalls. As mentioned before, these units may increase worst-case execution time and make timing analysis extremely difficult. Our realization of PRET employs a thread-interleaved pipeline to retain the benefit of pipelining, i.e. high instruction throughput, without suffering from pipeline stalls and complex prediction units.

Multi-threaded architectures contain multiple hardware contexts. They increase throughput over regular pipelined architectures by quickly switching context whenever a thread is stalled. However, dynamic thread-scheduling policies is challenging for execution time analysis. A thread-interleaved pipeline [23], [24] is a special multi-threaded architecture that fetches instructions from hardware threads in a round-robin order. Thread-interleaved processors preserve the benefits of a multi-threaded architecture – increased throughput, but use a predictable fine-grained thread-scheduling policy – round robin. Providing enough hardware threads then removes data dependencies between the instructions executing in the pipeline. For example, if there is the same number of hardware threads as there are pipeline stages, each stage of the pipeline will be occupied by a different hardware thread; there will be no dependencies between pipeline stages, and the execution time of each hardware thread is independent of all others. In our implementation of thread-interleaving, if one thread is stalled waiting for a memory access, the other threads can continue to execute normally without being affected. Figure 1 shows an example of how thread-interleaved pipelines removes data dependencies.

B. Scratchpad Memories

Besides the pipeline, the memory hierarchy of an architecture is a major source of timing unpredictability. Caches are a small but fast memory that is often used to bridge the latency gap between processors and main memory. They buffer frequently used instructions and data, exploiting spatial and temporal locality. Which instructions and data to cache is decided by the cache's replacement policy. Since caches are abstracted away from the programmer by the architecture, the programmer is not aware of whether the data is fetched from main memory or the cache. Fetching from the cache or the memory however has huge implications on the performance of the memory operation; often, the difference is hundreds of processor cycles. Thus, a huge part of analyzing execution time depends on whether or not a memory access can be accurately classified as a cache access or memory access. However, the destination of the memory access is highly dependent on the replacement policy and current state of the cache [25], which makes this prediction difficult.

Scratchpad memories [26] were initially introduced to save power in embedded systems. Scratchpads, like caches, are small but fast memory, except that scratchpads strip away the hardware logic for the replacement policy. Instead of a hardware-controlled replacement policy, the scratchpad replacements are managed by software. This can be done either explicitly by the programmer, or automatically with compiler-inserted instructions. Because the scratchpad allocation is done in software, the contents of the scratchpad are known statically, and precise and efficient timing analysis is easy.

As noted before, if the cache is shared by several threads in a multi-threaded architecture, then each thread's memory access influences the execution history and therefore the state of the cache. Even with scratchpads unrestricted sharing would be problematic. We partition the scratchpad memories among all hardware threads in PRET. In doing so, each thread's scratchpad allocation is independent, allowing us to do timing analysis of each thread independently.

C. Dynamic Random Access Memory

Modern DRAM (Dynamic Random Access Memory) architectures are highly parallel, containing multiple banks that can service memory requests concurrently. However, if subsequent memory operations from the processor need to access the same bank, then a bank conflict occurs and the second access must wait until the first access finishes. Modern memory controllers attempt to optimize throughput by queueing up memory requests and reordering them to reduce bank conflicts. This creates timing variability because the latency for each access to DRAM depends on the memory accesses surrounding it.

The PRET core leverages its predictable scheduling policy of multiple hardware threads and the parallel structure of the DRAM to achieve a predictable DRAM access [27]. Each hardware thread is assigned private banks of the DRAM. Since the hardware threads execute in a round-robin fashion, bank conflicts cannot occur because threads access different memory banks for their own request. No consecutive memory request will go to the same bank. Shared memory can be achieved by allocating separate DRAM banks that are accessed in a time-triggered fashion by the threads. Alternatively, the scratchpad can also be used to provide a small shared memory with shorter access latency.

Another source of timing variability in DRAM stems from its need to be refreshed periodically. DRAM cells leak charge, which causes them to lose their data over time. Thus, they need periodic refreshes to retain their data. Refreshes are commonly initiated by the DRAM controller, which uses a built-in timer to ensure refreshes occur when needed. However, if a memory access occurs during a refresh, then the memory access needs to be queued until the refresh is finished. Thus, to be conservative, timing analysis would need to take into account a possible refresh during any memory operation. In our approach, we propose using a distributed RAS-only refresh [28] to each bank separately, instead of refreshing all banks at once. Memory refresh operations are now equivalent to row accesses of a bank, which allows the memory controller to refresh each thread's banks separately. This can be done independently for each thread when there are no memory operations from that thread. It is also possible to bring the abstraction level of refreshes up to the software and use a static analyzer to insert refresh instructions in the program as long as it can guarantee that the refresh requirements are met. This allows a more accurate execution time analysis of memory accesses, because we ensure that there are no conflicts between DRAM accesses and refreshes.

D. Discussion

The PRET architecture decouples the execution of hardware threads in its thread-interleaved pipeline. This allows us to do separate timing analysis for all threads, and guarantee that the analysis will hold when we compose tasks together using different hardware threads.

For an integrated architecture, this is crucial because it allows us to composably integrate independent components while preserving the temporal properties. This allows for a simple and efficient integration of large-scale systems.

Within each thread context, instructions are independent of their execution context within the program, and the latency of memory operations no longer depends on any previous memory operations as it would if caches or modern memory controllers were employed. This enables a simple and accurate architectural timing analysis of a program.

III. Conclusion

As systems continue to scale in size and complexity, the ability to design and verify functions separately is crucial. It allows the complexity of the whole system to be broken down into simpler functions. This modularization can lead to shortened design time and safer designs. However, if the hardware platform destroys timing properties of functions during system integration, then the timing properties of functions can no longer be designed and verified separately. In this paper we describe an implementation of PRET that allows concurrent programs to be composed while preserving their temporal properties. This implementation utilizes a thread-interleaved pipeline, scratchpad memories and a composable and predictable DRAM controller.

References

[1] A. Sangiovanni-Vincentelli and M. Di Natale, "Embedded system design for automotive applications," *Computer*, vol. 40, no. 10, pp. 42–51, 2007.

[2] R. Obermaisser, C. El Salloum, B. Huber, and H. Kopetz, "From a federated to an integrated automotive architecture," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 28, no. 7, pp. 956–965, 2009.

[3] C. Watkins and R. Walter, "Transitioning from federated avionics architectures to integrated modular avionics," in *DASC '07*, October 2007.

[4] T. A. Henzinger, "Two challenges in embedded systems design: Predictability and robustness," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 366, issue 1881, pp. 3727–3736, 2008.

[5] E. A. Lee, "Absolutely positively on time: What would it take?" *Computer*, vol. 38, pp. 85–87, 2005.

[6] L. Thiele and R. Wilhelm, "Design for timing predictability," *Real-Time Systems*, vol. 28, no. 2-3, pp. 157–177, 2004.

[7] R. Wilhelm et al., "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 1–53, 2008.

[8] 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, 2003.

[9] 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," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 28, no. 7, pp. 966–978, 2009.

[10] S. A. Edwards and E. A. Lee, "The case for the precision timed (PRET) machine," in *DAC '07: Proceedings of the 44th Annual Design Automation Conference*, 2007, pp. 264–265.

[11] J. Yan and W. Zhang, "A time-predictable VLIW processor and its compiler support," *Real-Time Systems*, vol. 38, no. 1, pp. 67–84, 2008.

[12] J. Whitham and N. Audsley, "Predictable out-of-order execution using virtual traces," in *RTSS '08*, 2008, pp. 445–455.

[13] S. Uhrig, S. Maier, and T. Ungerer, "Toward a processor core for real-time capable autonomic systems," in *Proceedings of the Fifth IEEE International Symposium on Signal Processing and Information Technology*, 2005.

[14] C. Rochange and P. Sainrat, "A time-predictable execution mode for superscalar pipelines with instruction prescheduling," in *CF '05*. New York, NY, USA: ACM, 2005, pp. 307–314.

[15] 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 *CASES '05*. New York, NY, USA: ACM, 2005, pp. 213–224.

[16] J. Barre, C. Rochange, and P. Sainrat, "A predictable simultaneous multithreading scheme for hard real-time," in *ARCS '08*, 2008, pp. 161–172.

[17] M. Schoeberl, "A time predictable Java processor," in *DATE '06: Proceedings of the Design, Automation and Test in Europe Conference*, 2006, pp. 800–805.

[18] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken, "Compsoc: A template for composable and predictable multi-processor system on chips," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 14, no. 1, pp. 1–24, 2009.

[19] B. Akesson, K. Goossens, and M. Ringhofer, "Predator: a predictable SDRAM memory controller," in *CODES+ISSS '07*, 2007, pp. 251–256.

[20] M. Paolieri, E. Quinones, F. Cazorla, and M. Valero, "An analyzable memory controller for hard real-time CMPs," *Embedded Systems Letters, IEEE*, vol. 1, no. 4, pp. 86 –90, dec. 2009.

[21] T. Lundqvist and P. Stenström, "Timing anomalies in dynamically scheduled microprocessors," in *RTSS '99*, 1999, p. 12.

[22] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker, "A definition and classification of timing anomalies," in *WCET '06*, July 2006.

[23] B. J. Smith, *Architecture and applications of the HEP multiprocessor computer system*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, pp. 342–349.

[24] E. Lee and D. Messerschmitt, "Pipeline interleaved programmable DSP's: Architecture," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 35, no. 9, pp. 1320–1333, 1987.

[25] 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, November 2007.

[26] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: design alternative for cache on-chip memory in embedded systems," in *CODES '02*, 2002, pp. 73–78.

[27] S. A. Edwards, S. Kim, E. A. Lee, I. Liu, H. D. Patel, and M. Schoeberl, "A disruptive computer design idea: Architectures with repeatable timing," in *ICCD '09*, 2009.

[28] Micron Technology, Inc., "Various methods of DRAM refresh – rev. 2/99," 1994, http://download.micron.com/pdf/technotes/DT30.pdf.