Temporal Isolation on Multiprocessing Architectures *

Dai Bui
University of California, Berkeley
Berkeley, CA
daib@eecs.berkeley.edu

Edward A. Lee
University of California, Berkeley
Berkeley, CA
eal@eecs.berkeley.edu

Isaac Liu
University of California, Berkeley
Berkeley, CA
liuisaac@eecs.berkeley.edu

Hiren D. Patel
University of Waterloo
Waterloo, Ontario, Canada
hdpatel@uwaterloo.ca

Jan Reineke
University of California, Berkeley
Berkeley, CA
reineke@eecs.berkeley.edu

ABSTRACT

Multiprocessing architectures provide hardware for executing multiple tasks simultaneously via techniques such as simultaneous multithreading and symmetric multiprocessing. The problem addressed by this paper is that even when tasks that are executing concurrently do not communicate, they may interfere by affecting each other's timing. For cyber-physical system applications, such interference can nullify many of the advantages offered by parallel hardware. In this paper, we argue for temporal semantics in layers of abstraction in computing. This will enable us to achieve temporal isolation on multiprocessing architectures. We discuss techniques at the microarchitecture level, in the memory hierarchy, in on-chip communication, and in the instruction-set architecture that can provide temporal semantics and control over timing.

Categories and Subject Descriptors

C.1.3 [**Computer Systems Organization**]: Processor Architectures—*Other Architecture Styles*; C.3 [**Special-Purpose and Application-Based Systems**]: *real-time and embedded systems*; J.7 [**Computers in Other Systems**][computer-based systems]

General Terms

Design

Keywords

Precision-timed architectures, microarchitecture, pipelines, memory hierarchy, network on chip, instruction set architecture

*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), #0931843 (ActionWebs), and #1035672 (CSR-CPS Ptides)), 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.

DAC '11 San Diego, California USA

1. INTRODUCTION

A cyber-physical system (CPS) uses embedded computers to monitor and control physical processes [8]. These systems are typically in feedback loops such that the physical processes affect the behavior of the computation, and vice versa. Note two important aspects of physical processes: they are inherently concurrent, and time in physical processes progresses at its own pace; time is uncontrollable. Therefore, to control physical processes, embedded computers need to provide support for concurrent behaviors and timely reactions.

Modern embedded computing architectures are becoming increasingly parallel by using multiprocessing techniques such as multicores and simultaneous multithreading (SMT). These parallel architectures deliver higher performance and also address the need for concurrency. However, ensuring that real-time requirements are met is a challenge in such architectures. A major problem is that multiprocessing techniques can introduce temporal interference even between logically independent behaviors.

For example, in a multicore machine with shared caches, the processes running on one core can affect the timing of processes on another core even when there is no communication between these processes. Similarly, SMT [4] shares a wide-issue superscalar pipeline across multiple hardware threads with instructions dispatched using scoreboarding mechanisms. This allows threads to occupy more pipeline resources when other threads are idle or stalled for long latency operations. This results in temporal interference between the threads because of the contention for pipeline resources.[1] On the other hand, symmetric multiprocessing (SMP) techniques use multiple processing units connected with an on-chip communication interconnect such as

[1] Several research groups have introduced ways to mitigate the temporal interference in SMTs by allocating a real-time thread with the highest priority [2], time-sharing the real-time thread [12], and partitioning the instruction cache [11].

a bus or network-on-chip. SMP exploits thread-level parallelism by designating a thread to a different processing unit. While this removes temporal interference caused by sharing pipeline resources between multiple threads, temporal interference is reintroduced at the on-chip communication interconnect. For instance, sharing the same off-chip memory between processing units requires arbitration and coherence, which results in temporal interference. In fact, any sharing of resources such as busses, memories, switches, buffers, and I/O devices can result in temporal interference.

Programming multiprocessing architectures for CPS applications is another challenge. Model-based design advocates software synthesis from higher-level models, where the models themselves can have temporal semantics. For example, MathWorks Simulink® models can specify periodic behaviors, and tools such as Real-Time Workshop® and dSPACE TargetLink® can synthesize multithreaded C code for execution on embedded processors. But ensuring that the real-time behavior of the deployed application matches the original model is becoming increasingly difficult with multiprocessing architectures.

We contend that software synthesis to multiprocessing architectures for time-critical applications could be made much easier if we could ensure temporal isolation across well-defined boundaries. Temporal isolation enables multiple processes to execute concurrently without having an impact on another process' temporal behavior.

To provide temporal isolation, it will never be enough to just generate C code and hope for the best. There has to be underlying support. This will require changes at lower levels of abstraction, which brings us to the purpose of this paper: to examine what changes need to be made at the lower levels to support temporal isolation for effective software synthesis for multiprocessing architectures.

It is certainly possible to achieve temporal isolation on just about any hardware by overprovisioning. This means providing sufficient resources so that there is no contention for them. This does not preclude sharing of resources, but it does constrain how the sharing is done. Time-division multiple access (TDMA) schemes, for example, share resources by dividing their access into fixed time slots. This is commonly done in distributed real-time systems, using TDMA busses such as ARINC, TTP, and FlexRay. The constraints imposed by TDMA, however, make software less reactive and often result in considerable overdesign, where for example sensors are polled at high rates to avoid having to design a reactive response.

Minimizing temporal interference requires more control over shared resources [9]. This includes anything that performs switching and routing in a network-on-chip (NoC), as well as shared buses, shared caches and memories, cache coherence protocols, and task scheduling policies that work across processor resources.

In this paper, we begin with low levels of abstraction and work our way up. We begin with a brief discussion of the relevant layers of abstraction. Then we discuss what we believe must change at the microarchitecture level, the memory architecture, the NoC, and the instruction-set architecture (ISA), in that order. We also believe that profound changes are required in higher-level programming models, but that discussion is out of scope for this paper. We conclude the paper with some simple illustrations of how to take advantage of temporal isolation.

2. ABSTRACTION LAYERS NEED TIME

Although computers are concretely implemented using circuits, programs and applications are designed using higher-level abstractions. These abstractions include microarchitectures, instruction-set architectures (ISAs), programming languages, operating systems, and concurrent models of computation.

At the lowest levels of abstraction, circuits and microarchitectures, timing is central to correctness. If in a microarchitecture, for example, the output of an ALU is latched at the wrong time, the ISA will not be correctly implemented. At higher levels, however, timing is hidden. An ISA typically has no temporal semantics at all. The time it takes to realize an instruction in the ISA is irrelevant to correctness.

Achieving temporal isolation at the higher levels alone, therefore, will be challenging. Time is not part of the semantics, making it impossible to control. As a result, designers are forced to reach down into the lower levels of abstraction. For example, execution-time analysis of programs today requires detailed modeling of microarchitectures.

In this paper, we argue that we should revisit and rethink each of these layers, judiciously introducing some form of temporal semantics. This will make it possible to achieve temporal isolation without always reaching down to the lowest levels of abstraction.

3. MICROARCHITECTURE

Multithreaded microarchitectures enable true concurrency of tasks for embedded applications on a single core. This is attractive because embedded applications are inherently concurrent. However, when general purpose multithreaded microarchitectures implements ISAs, temporal interference is introduced when the hardware dynamically manages the underlying architectural resources. For example, simultaneous multithreaded architectures try to fill a multi-issue wide pipeline with its multiple threads dynamically. This allows more efficient use of the resources since idle threads will not consume any resources and the busier threads can in turn use more resources and finish quicker. But as a result, a thread's resource consumption directly depends on the activity of other threads sharing the core.

Several techniques have been proposed to remove temporal interference for multithreaded architectures. In particular, Mische et al. [12] used a simultaneous mulithreaded architecture and assigned a top priority to a thread dedicated as the real-time thread. The architecture was modified slightly to allow interruption of instruction execution and rollback. The modification allows the real-time thread to access any hardware resource it needs by interrupting execution of other lower priority threads. This guarantees temporal isolation for the real-time thread, but not for any other thread running on the core. If multiple real-time tasks are needed, then a time sharing of the real-time thread is required. A more static approach was proposed by El-Haj-Mahmoud et al. [5] called the virtual multiprocessor. The virtual multiprocessor uses static scheduling on a multithreaded superscalar processor to remove temporal interference. The processor is partitioned into different time slices and superscalar ways, which are used by a scheduler to construct the thread execution schedule offline.

The precision timed machine (PRET) project reintroduces computer architecture with time as its first class citizen.

Our realization of PRET [10] employs a thread-interleaved 5 stage pipeline. The thread-interleaved pipeline contains four hardware threads that run in the pipeline. Instead of dynamically scheduling the execution of the threads, a predictable round-robin thread schedule is used to remove temporal interference. The round-robin thread schedule fetches a different thread every cycle, removing data hazard stalls stemmed from the pipeline resources. Unlike the virtual multiprocessor, the tasks on each thread need not be determined a priori, as hardware threads cannot affect each other's schedule. Unlike Mische et al. [12]'s approach, all the hardware threads in our architecture can be used for real time purposes.

Multi-core architectures also allow true concurrent execution of tasks on multiple cores. Although tasks execute on separate cores, timing interference still exists due to contention from the memory architecture or communication fabric. Thus, the challenge of using multi-core architectures to remove timing interference from the arbitration of the shared components. The MERASA [17] project propose an interference-aware bus arbiter to bound the interference for real-time tasks. The bus arbiter splits the arbitration from within a core and in between cores. Real-time bus requests are given priority, and when multiple real-time requests are present, a round-robin policy is used for arbitration. The MERASA project also propose an analyzable memory controller which bounds the interference of memory accesses using a similar arbitration scheme as their bus arbiter. Pitter et al. [13] designed a Java chip multiprocessor with one shared memory, and evaluated different arbitration schemes to access the shared memory. They concluded that a TDMA arbitration is the key component enabling WCET analysis of memory accesses. In the following sections we discuss more about the memory hierarchy and network-on-chip.

4. MEMORY HIERARCHY

4.1 Caches and Scratchpad Memories

There is a large gap between the latency of current processors and that of large memories. Thus, a hierarchy of memories is necessary to provide both low latencies and large capacities. In conventional architectures, caches are part of this hierarchy. In caches, a replacement policy, implemented in hardware, decides which parts of the slow background memory to keep in the small fast memory. Replacement policies are hardwired into the hardware and independent of the applications that are running on the architecture.

Without further adaptation, caches do not provide temporal isolation: the same application, processing the same inputs, may exhibit wildly varying cache performance depending on the state of the cache when the application's execution begins [18]. The cache's state is in turn determined by the memory accesses of other applications running earlier. Thus, the temporal behavior of one application depends on the memory accesses performed by other applications. In multi-core systems, caches are shared among several cores. In such systems, temporal interference between different applications is more immediate: memory accesses of applications running on different cores are interleaved; an access of one application may cause eviction of a memory block of another application, and thus incur additional cache misses later.

A solution to avoid at least some of the interferences in caches described above are partitioned caches. Different applications and/or different cores are assigned partitions of the cache, and can therefore not interfere with each other. In such a scheme, an application's temporal behavior still depends on the state of the cache at the end of the previous execution of the same application. Some caches allow to lock their contents, to eliminate such interferences.

Scratchpad memories (SPMs) are an alternative to caches in the memory hierarchy. The same memory technology that is employed to implement caches is also used in SPMs, static random access memory (SRAM), which provides constant low-latency access times. In contrast to caches, however, an SPM's contents are under software control: the SPM is part of the addressable memory space, and software can copy instructions and data back and forth between the SPM and lower levels of the memory hierarchy. Accesses to the SPM will be serviced with low latency, predictably and repeatably. However, similar to the use of the register file, it is the compiler's responsibility to make correct and efficient use of the SPM. This is challenging, in particular when the SPM is to be shared among several applications, but it also presents the opportunity of high efficiency, as the SPM management can be tailored to the specific application, in contrast to the hardwired cache replacement logic.

4.2 Dynamic Random Access Memory

At the next lower level of the memory hierarchy, many systems employ Dynamic Random Access Memory (DRAM). DRAM provides much greater capacities than SRAM, at the expense of higher and more variable access latencies.

Conventional DRAM controllers do not provide temporal isolation. As with caches, access latencies depend on the history of previous accesses to the device.

DRAM devices consist of a set of independent banks, each of which stores data in DRAM cells in a two-dimensional matrix organized in rows and columns. In order to access a memory block in a particular bank, a memory controller needs to first transfer the row containing the memory block into the row buffer, by issuing a row access command. Then, it can access contiguous parts of this row by column access commands. Subsequent accesses to the same row need not transfer the row into the row buffer again. The latency of a memory access thus depends on whether the accessed memory block is contained in the same row as the immediately preceding access to the same bank.

Modern DRAM controllers reorder accesses to minimize row accesses and thus access latencies. As the data bus and the command bus, which connect the processor with the DRAM device, are shared between all of the banks of the DRAM device, controllers also have to resolve competition of for these resource by different competing memory accesses.

We are currently developing a DRAM controller, which provides temporal isolation among different clients. This controller allocates private banks to different clients. In addition, it employs a time-triggered access scheme to the shared address and data bus. This completely decouples the access latencies of clients from one another.

Previous related work by Akesson et al. [1] achieves temporal isolation differently: the backend of the controller, which issues commands to the DRAM device provides bounds on access latencies. Interference between different clients is bounded by arbitration schemes such as CCSP or Round-Robin. Given the guarantees provided by the backend and

the arbitration mechanism, the approach of Akesson et al. is to delay response to every memory access to the worst case.

5. ON-CHIP COMMUNICATION

General-purpose on-chip networks are built to maximize throughput. They dynamically route packets through the network. Packets from different sources compete for the limited capacity of the network. The latency of a packet depends on the current network activity and the routing decisions taken by the network. Thus, without further adaptation, such networks do not provide temporal isolation between different applications. In the following, we discuss two approaches that can achieve predictable performance or temporal isolation for on-chip communication.

The Æthereal network-on-chip [6], from NXP, uses a time-division multiple-access (TDMA) scheme for packets that are subject to guaranteed service constraints. Guaranteed-service packets are sent in fixed-length slots. Contention is avoided in two ways: by *globally* scheduling the packet injection from network nodes at network interfaces, and by using contention-free routing that solves contention by *delaying* packets at source nodes. Global scheduling limits the scalability of this approach. In Æthereal, TDMA slot allocation is not done at runtime, presumably to prevent temporal interference, but this prevents the application software from exploiting path diversity to avoid congestion.

Rosen et al. [14] propose optimizing a TDMA bus schedule to minimize program worst-case execution time (WCET) for chip multiprocessors (CMPs). The paper also shows that more complicated TDMA bus schedules could result in better performance but would make the bus scheduler implementation complicated and costly. Our ISA extensions, described below, can be used to implement more complicated bus access patterns in software instead of at the pure hardware level.

An alternative to TDMA are priority-based mechanisms. Such mechanisms do, however, introduce temporal interference between different applications. Shi et al. [15] show how to analyze such a priority-based mechanism to determine latency bounds.

6. INSTRUCTION SET ARCHITECTURES

Common instruction set architectures (ISAs) do not have a temporal semantics, i.e., the time it takes to execute a sequence of instructions is not constrained by the ISA at all. Different microarchitectural realizations of the same ISA may exhibit completely different timing behavior. In fact, the execution time of a program may even vary from one execution to the next on the same microarchitectural realization.

On the other hand, higher-level models, expressed using e.g. MathWorks Simulink®, do have temporal semantics. To synthesize binaries from such higher-level models, tools such as Real-Time Workshop® can generate C code, which is then compiled into a binary of an instruction set architecture. As neither the C code nor the binary exhibit temporal semantics, designers are forced to reach down into the microarchitectural level to verify that the temporal constraints of the higher-level model are met. For example, execution-time analysis of programs today requires detailed modeling of microarchitectures.

We propose to endow ISAs with temporal semantics and in particular with instructions that enable control over timing. This, in turn enables control over timing at the C level, which can be used to implement higher-level models with temporal semantics.

6.1 Four Desirable Capabilities of an ISA

We have identified the following four capabilities to control timing that would be desirable at the ISA level:

C1 Execute a block of code taking at least a specified time [7].

C2 Execute a block of code. Conditionally branch if the execution of the block exceeded a specified amount of time (the deadline).

C3 Execute a block of code. If, *during* the execution of the block, a specified amount of time is exceeded, branch immediately to an exception handler.

C4 Execute a block of code in *at most* a specified amount of time (the deadline).

Capabilities C1, C2, and C3 can be added to a given ISA relatively easily. They can be used to ensure that a block takes at least a certain amount of time, and to act upon deadline misses, either immediately, in case of C3, or after finishing the execution of a block of code (C2).

Capability C4, on the other hand, is more challenging, but also more intriguing, as it requires to bound the execution times of blocks of code.

6.2 ISA Extensions

In order to implement these four capabilities, we propose the following ISA extensions. Here we assume that time is measured in *thread cycles*, where one thread cycle corresponds to the minimum amount of time between instruction issues for a thread. Most instructions execute in one thread cycle, but some may take multiple cycles to execute. The *PRET principle* requires that it be possible to determine from the assembly code how many cycles elapse during the execution of a sequence of instructions. Below, "current time" is the thread cycle count when the instruction starts.

- **set_time %r, *offset***: load current time + *offset* into register %r.
- **delay_until %r**: ensure that the following instruction executes at time [%r][2], if possible, and otherwise at the earliest possible time.
- **branch_expired %r, *target***: branch to *target* if current time is greater than [%r].
- **expection_on_expire %r, *id***: arm processor to throw exception *id* when current time equals [%r] + 1.
- **deactivate_exception *id***: disarm the processor for exception *id*.
- **mtfd %r**: ensure that current time is less than or equal to [%r] whenever this instruction is executed.

Capability C1 above, for example, can be implemented using **set_time** and **delay_until**. Capability C3 requires **set_time**, **exception_on_expire**, and **deactivate_exception**.

The most interesting of the above instructions is **mtfd**, the name of which stands for "meet the final deadline." It implements capability C4. How can such an instruction be realized? If the code between a **set_time** instruction and an **mtfd** instruction is defined in a Turing-complete programming language (which includes all widely used programming languages), then the amount of time it takes to execute that

[2]Here, [%r] denotes the value of register %r.

code is undecidable. The **mtfd** instruction requires that some static analysis technique be able to assure that the execution time is bounded. This can be implemented, for example, by having a compiler include execution-time analysis of the body of code along all paths between **set_time** and **mtfd**, and generate from this analysis a specification of hardware requirements that ensure that the deadline is met. For example, it could generate a bound on the number of cycles required by the particular ISA instructions that the compiler uses along the paths. Although no such compiler will be able to generate such a specification for all programs and all architectures (this would solve an undecidable problem), it is certainly possible to generate one for some programs and restricted classes of architectures. The generated specification could be included in the object file produced by the compiler, and a linker/loader could check the specification against the capabilities of the hardware being targeted. It could then reject the program if the target hardware does not have adequate capability.

Use of **mtfd** should be limited to situations where timing control is strictly required (for example, in safety-critical systems), because the above strategy will always result in a tool chain that will reject some valid programs.

7. APPLICATIONS

Many potential benefits stem from an ability to achieve precise timing while executing concurrently. In this section, we give a few simple examples that take advantage of such precise timing.

7.1 Mutual Exclusion

To illustrate how the ISA extensions above can be used for synchronization, we present a simple producer/consumer example. This example has been adapted from an earlier version in [10].

In our example, the producer and the consumer communicate through a shared buffer. A general approach to managing a shared buffer across separate threads is to have mutually-exclusive critical sections that only a single thread can access at a time.

We synchronize the producer and the consumer using the ISA extensions described above, avoiding the common solution to mutual exclusion: locks. In our approach, the time that a thread waits to access the resource does not depend on the behavior of the other threads accessing the resource. The threads are temporally isolated.

Figure 1 gives pseudo-code for the producer and the consumer accessing the shared variable `buf`. The producer iterates and writes an integer value to the shared data. The consumer reads this value from this shared data and stores it in an array. For simplicity, our consumer does not perform any other operations on the consumed data. It just stores the data in the array.

The ISA extensions in Figure 1 are interleaved with the C-like code and marked in color and italics. We use staggered deadlines at the beginning of each thread to offset the threads and force the producer to start before the consumer (our architecture enables simultaneous starting of threads). Inside each thread, deadlines force each loop to run in lockstep, each thread progressing at the same rate. The relative timing of the threads is constant, so that the consumer reads data only after the producer has written it.

We assume here that the shared buffer is located in the scratchpad memory. This guarantees that accesses to the buffer will take a single thread cycle and will be atomic, so we only need to ensure that the n^{th} write occurs before the n^{th} read. For some programs (not this one), we may also need to ensure that the n^{th} read occurs before the $(n+1)^{th}$ write. We leave this as an exercise.

The offsets to achieve this are given by deadlines at the beginning of the program. The deadline of 31 thread cycles on line 2 of the consumer is computed from the assembly language instructions and is the smallest deadline that the consumer thread can meet. The offset of the producer loop is 30 thread cycles. This ensures that the consumer reads after the producer writes, independently of which hardware threads the producer and consumer are running in.

Once inside the loop, deadlines force each thread to run at the same rate, maintaining the memory access schedule. We choose the smallest possible period that both the producer and consumer can make. Analysis of the assembly language instructions reveals that a loop iteration of the consumer takes 9 thread cycles, whereas an iteration of the producer takes less time.

7.2 Message-dependent Deadlock Avoidance

Careful control of timing can also help make a multiprocessing system free of deadlocks. Consider a program that transmits packets repeatedly, where the time between packet transmissions is data dependent. If messages are sent faster than the recipient can handle, buffers along the path may fill. This can result in message-dependent deadlock (MDD) [16]. The problem can be avoided by enforcing a minimum amount of time between message sends, something easily implemented with *delay_until* [3], and enforcing a maximum amount of time between recipient reads, something we can do with *mtfd*. This relies on controlled latency in the network (see Section 5).

While it is clear that such controlled timing can, in principle, prevent MDD, in practice, doing this well may be challenging. Consider for example a message source that, using *delay_until*, is constrained to transmit message with a time gap of at least τ seconds. Suppose that on the path to the recipient there is a buffer that can hold N messages. Then the recipient must be constrained, using *mtfd*, to receive at least $M-N+1$ messages in every time window of length $M\tau$, for any natural number M. There are simple schemes that guarantee this property [3]. For example, we can use *mtfd* to ensure that the recipient reads a message at least every τ seconds. But this is over-constraining. Looser constraints on the receiving problem are undoubtedly possible.

8. CONCLUSIONS

Temporal isolation can be achieved today at coarse granularity, but precise control has gotten increasingly more difficult due to clever techniques at the lower layers, the microarchitecture, the memory system, and the NoC. The fact that today's ISAs have no temporal semantics makes these clever techniques perfectly valid, but for applications that require precise or controllable timing, the clever techniques enormously complicate the system design task. A system designer does not know the temporal behavior until the system is completely built and deployed on particular hardware, and even then, the temporal behavior emerges in unpredictable ways from minute details of the implementation.

This paper argues that by introducing temporal seman-

```
1  void producer() {
2    set_time R1, 30
3    volatile int *buf = (int*)(0x3F800200);
4    unsigned int i = 0;
5    while (true) {
6      delay_until R1
7      mtfd R1
8      *buf = i;
9      addi R1, R1, 9
10     i++;
11   }
12 }
```

Listing 1: Producer

```
1  int consumer() {
2    set_time R1, 31
3    volatile int *buf = (int*)(0x3F800200);
4    unsigned int i = 0, arr[8];
5    for (i = 0; i < 8; i++)
6      arr[i] = 0;
7    while (true) {
8      delay_until R1
9      mtfd R1
10     int tmp = *buf;
11     arr[i mod 8] = tmp;
12     i++;
13     addi R1, R1, 9
14   }
15 }
```

Listing 2: Consumer

Figure 1: Producer/Consumer Example: Using ISA extensions for synchronization.

tics at the lower layers, we can ensure that the behavior of deployed parallel and concurrent systems matches their specifications and their behavior during testing. More importantly, since temporal behavior can be isolated, time-sensitive services become composable even when they share resources.

9. REFERENCES

[1] B. Akesson et al. Composability and predictability for independent application development, verification, and execution. In M. Hübner and J. Becker, editors, *Multiprocessor System-on-Chip — Hardware Design and Tool Integration*. Springer, 2010.

[2] J. Barre, C. Rochange, and P. Sainrat. A predictable simultaneous multithreading scheme for hard real-time. In *Proc. of ARCS*, volume LNCS 4934, pages 161–172. Springer, 2008. doi:10.1007/978-3-540-78153-0_13.

[3] D. Bui, H. Patel, and E. Lee. Deploying hard real-time control software on chip-multiprocessors. In *Proc. of RTCSA*, pages 283–292, 2010. doi:10.1109/RTCSA.2010.43.

[4] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17:12–19, 1997.

[5] 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 *Proc. of CASES*, pages 213–224, New York, NY, USA, 2005. ACM. doi:10.1145/1086297.1086326.

[6] K. Goossens et al. Guaranteeing the quality of services in networks on chip. In *Networks on chip*, pages 61–82, Hingham, MA, USA, 2003. Kluwer.

[7] N. J. H. Ip and S. A. Edwards. A processor extension for cycle-accurate real-time software. In *Proc. of the IFIP International Conference on Embedded and Ubiquitous Computing (EUC)*, pages 449–458, Seoul, Korea, 2006. doi:10.1007/11802167_46.

[8] E. A. Lee. Cyber physical systems: Design challenges. In *Proceedings of ISORC*, pages 363 – 369, Orlando, Florida, 2008. IEEE. doi:10.1109/ISORC.2008.25.

[9] E. A. Lee. Computing needs time. *CACM*, 52(5):70–79, 2009. doi:10.1145/1506409.1506426.

[10] B. Lickly et al. Predictable programming on a precision timed architecture. In *Proc. of CASES*, pages 137–146, USA, 2008. ACM. doi:10.1145/1450095.1450117.

[11] S. Metzlaff, S. Uhrig, J. Mische, and T. Ungerer. Predictable dynamic instruction scratchpad for simultaneous multithreaded processors. In *Proc. of MEDEA*, pages 38–45, New York, NY, USA, 2008. ACM. doi:10.1145/1509084.1509090.

[12] J. Mische, S. Uhrig, F. Kluge, and T. Ungerer. Exploiting spare resources of in-order SMT processors executing hard real-time threads. In *ICCD*, pages 371–376, 2008. doi:10.1109/ICCD.2008.4751887.

[13] C. Pitter and M. Schoeberl. A real-time java chip-multiprocessor. *ACM TECS*, 10(1):9:1–34, 2010. doi:10.1145/1814539.1814548.

[14] J. Rosen, A. Andrei, P. Eles, and Z. Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *Proc. of RTSS*, pages 49–60. IEEE, 2007. doi:10.1109/RTSS.2007.13.

[15] Z. Shi and A. Burns. Real-time communication analysis for on-chip networks with wormhole switching. In *Proc. of NOCS*, pages 161–170. IEEE, 2008. doi:10.1109/NOCS.2008.4492735.

[16] Y. H. Song and T. M. Pinkston. On message-dependent deadlocks in multiprocessor/multicomputer systems. In *HiPC*, pages 345–354, London, UK, 2000. Springer-Verlag.

[17] T. Ungerer et al. Merasa: Multicore execution of hard real-time applications supporting analyzability. *IEEE Micro*, 30(5):66 –75, Sept.-Oct. 2010. doi:10.1109/MM.2010.78.

[18] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE TCAD*, 28(7), 2009. doi:10.1109/TCAD.2009.2013287.