Fun with a Deadline Instruction

Martin Schoeberl
Institute of Computer Engineering
Vienna University of Technology, Austria
mschoebe@mail.tuwien.ac.at

Hiren D. Patel
University of Waterloo
Waterloo, Ontario, N2L 3G1, Canada.
hdpatel@uwaterloo.ca

Edward A. Lee
University of California, Berkeley
Berkeley, CA 94720, USA
eal@eecs.berkeley.edu

***Abstract*—In this paper we present example applications using a deadline instruction. The deadline instruction brings cycle accurate timing information into the application code. We have implemented the mechanism in a time-predictable Java chip-multiprocessor. As a proof of the accuracy that can be gained, a digital to analog conversion of audio signals is implemented completely in software. Furthermore, we show how the deadline instruction can be used to verify bytecode execution times on chip-multiprocessors and how to synchronize tasks to a time-division based memory arbiter.**

I. Introduction

Lee argues that building reliable real-time embedded systems require software to yield predictable and repeatable timing behaviors [11]. By having predictable timing behaviors, designers can analyze systems (e.g., with static timing analysis) and predict behaviors under various stimuli and operating conditions. Repeatable timing behaviors promote testing of the system such that multiple executions of the system under specific operating conditions yield the same timing behaviors. This is, of course, in addition to the requirement of correct functionality. Designers can then assess the reliability of the system through analysis and testing methods.

Since current abstraction layers in real-time embedded computing are a culmination of years of borrowing of concepts and tools from general purpose computing, they neglect the importance of predictable and repeatable timing. Instead, they focus on solely improving average-case performance. As a result, the property of timeliness has been abstracted away at various abstraction layers of computing allowing for computer architectures with speculative execution, deep pipelines, caches and complex memory hierarchies, and programming languages, multithreading and operating systems. While most of these provide techniques to optimize average-case performance, they make it extremely difficult to build reliable real-time embedded systems. This is because the resulting computer architectures exhibit unpredictable and nonrepeatable timing behaviors. Consequently, we need a paradigm shift for future real-time embedded systems where timeliness is made a first-class citizen of real-time embedded computing. Processor architectures shall be optimized for the worst-case execution time (WCET) instead of the average-case performance [19], and they shall exhibit predictable and repeatable timing behaviors.

In this paper, we explore the Java Optimized Processor (JOP) [18] and its supporting tools for building reliable real-time embedded systems. By design, the JOP is a time-predictable computer architecture for real-time embedded systems. It also has a chip-multiprocessor (CMP) version. Our focus here is on making time a first-class citizen, and on bringing the property of repeatable timing to JOP. We accomplish this by extending the hardware architecture and the programming interface to support timing instructions [4], [12]. In addition, we also allow for handling timing violations through timing exceptions [13] by using standard Java exceptions or borrowing the semantics of deadline miss handlers from the real-time specification for Java (RTSJ) [3].

The paper is organized as follows: In the following section related work is described and Section III gives short overview of JOP. The deadline instruction, deadline miss handling, and the implementation in JOP is described in Section IV. Several use cases of the deadline mechanism are given in Section V and the paper is concluded in Section VI.

II. Related Work

Ip and Edwards [9] first proposed the deadline instruction for their H8 PRET machine. The proposed processor was a single-cycle, non-pipelined design with a simple memory hierarchy that connected the processor with a fast SRAM. They showed that the deadline instruction combined with the simple architecture design made it simple to predict execution times, and to control the timing behaviors in software for repeatability.

A later incarnation of the PRET processor was proposed by Lickly et al. [12] that borrowed the deadline instruction and incorporated it into a multi-threaded processor based on the SPARC instruction-set architecture. This architecture uses a thread-interleaved pipeline, replicates the register file, and designates individual software-managed scratchpad memories for each of the hardware threads. One instruction from one hardware thread occupied at most one stage in the pipeline. This has the advantage that there are no data dependencies between the hardware threads. For stalls, the processor uses a replay mechanism where an instruction waiting for either a main memory access or a multi-cycle operation to complete is pushed through the pipeline as a nop, and reinserted into the pipeline without incrementing the program counter. This version of the SPARC-PRET processor showed that it is possible to maintain predictable and repeatable timing behaviors with high precision without foregoing performance.

The Virtual Simple Architecture (VISA) of Mueller at al. [1] enabled hard real-time tasks operation on unpredictable processors. The processor can be operated in two modes: in super-scalar mode and as a simple, analyzable in-order pipeline. The real-time tasks are split into subtasks and each subtask gets a deadline assigned. The execution of real-time tasks starts in the advanced processing mode and if a deadline is missed, the processor is switched to the simple mode. The individual deadlines have to be set so that there is enough slack time after missing a subtask's deadline so that this subtask and all following subtasks can be executed in the slow mode without missing the final task deadline. The advantage of this architecture is, that in most cases the advanced mode will finish in time and leave processing time for non-real-time tasks. In contrast, the deadline instruction provides detailed timing control in software, and not just task completion times. Unlike VISA, PRET allows programmers to explicitly control the timing in software.

Andalam et al. [2] adopt the PRET philosophy [5] and propose a synchronous language extension for C called PRET-C. This extension supports synchronous concurrency, timing constructs to control logical time, and preemption. While this is a step toward the right direction for language design, the deadline instruction works on the processor clock cycle or physical time, which in the PRET-C language is not possible as yet.

The XMOS [14] processor architecture and tools support many of the PRET principles. Their XCore chip implements a thread-interleaved pipeline architecture with direct access to a fast SRAM. They also provide round-robin thread scheduling as in [12], but allow for another mode to switch out threads waiting on I/O or off-chip memory accesses. Their XCore can be tiled with other XCore processors through an interconnect. They also developed a language called XC, which allows for explicit control over timing.

III. A Time-predictable Java Processor

We evaluate the concept of a deadline instruction in the context of a time-predictable Java processor, called JOP [18]. JOP is an implementation of the Java virtual machine (JVM) in hardware and the primary implementation technology is in a field-programmable gate array (FPGA). JOP is open-source and the design is available from http://www.jopdesign.com/.

The primary design constraint of JOP is time-predictable execution of real-time Java programs. JOP targets the future safety-critical specification for Java [7]. Execution time of bytecodes, the instructions of the JVM, can be predicted with cycle accuracy. Therefore, the low-level part of worst-case execution time (WCET) analysis is greatly simplified. The distribution of the JOP project includes a WCET analysis tool [22], [8].

JOP is also used as a basis for research on time-predictable chip-multiprocessing (CMP) [15]. A time-division multiple access (TDMA) based arbiter for the main memory isolates the timing behavior of the different cores. With the known, static schedule the memory access time can be predicted statically [16]. Therefore, WCET analysis for tasks running on individual cores in a JOP CMP system is still possible.

Listing 1. Deadline exception generated in Java

```
public void deadWait(int time) throws DeadlineMissed {

    sys.deadLine = time;
    if (sys.missedDeadline) {
        throw new DeadlineMissed();
    }
}
```

IV. The Deadline Instruction

A deadline instruction brings time semantics into the application program. In the simplest case, the deadline instruction delays execution until a specified time in the future. Compared to simple timers, this instruction allows expressing timing constraints in the application code with clock cycle accuracy.

For code with variable execution time the deadline instruction can be used to enforce a maximum execution time of a code block. Therefore, when the worst-case execution time (WCET) of these blocks is known, the deadline can be set to those values, or higher – assuming single threaded execution. However, when the WCET estimate is wrong (or unknown) deadlines can be missed. The possible options on deadline misses are described in the next section.

A. Deadline Violation

What happens on a deadline violation? Four options exist: (1) ignore the missed deadline, (2) set an overrun status bit, (3) throw an exception, or (4) invoke a deadline miss handler. For hard real-time systems, which undergo a rigid certification process, one can argue that deadline misses are avoided by design. This is the approach taken by the upcoming specification for safety-critical Java [7]. Polling a status register to query if the deadline was missed is similar to the return value of waitForNextPeriod() in the RTSJ [3]. It is up to the application programmer to decide if the miss is handled by the application code.

Generation of a Java exception allows handling of deadline misses in the application. We can generate checked or unchecked exceptions in the hardware. Unchecked exceptions are for events that indicate an abnormal program flow (e.g., division by zero). They can be caught, but the application can also decide to not handle that exception. In that case the thread is terminated. Checked exceptions force the programmer to handle the exception. Those exceptions can only be declared at method bodies. Therefore, we have to wrap the deadline instruction into a method. Generation of a checked exception can be implemented in Java with the status polling approach, as shown in Listing 1.

The main drawback of handling the deadline miss in the application code is that on a deadline miss the processor is forced to even execute more code, probably worsen the

situation. Another option is to use explicit deadline miss handlers. Deadline miss handlers are available in the RTSJ when the optional feature of deadline monitoring is implemented. The miss handler can be a task with its own scheduling parameters. The priority can be higher than the application task that generated the deadline violation or it can be lower to defer the action taken on a miss. If the miss handler is implemented as a first level interrupt handler, the overhead of a scheduler decision can be avoided and the miss handler runs at top priority. An advantage of the miss handler is that the error handling is not interleaved with application code. Furthermore, a single handler can be used for all deadline instructions in the application. Perhaps there is a degraded application mode available and the miss handler triggers this mode on any deadline miss, or a simple reboot of the system is the appropriate action to take.

The above discussed suggestions, except RTSJ miss handlers, have a main drawback: the deadline miss is detected at the end of execution of the code block, but not at the actual miss time. The deadbranch instruction, as presented in [13], provides a mechanism to setup a timer before the execution of the code that will raise an exception when the timer expires. The end of the code block under the deadline control has to reset the timer with a deadload instruction. A combination of deadline monitoring and maximum execution time enforcement results in following code pattern:

```
time = sys.cntInt+1000;
try {
    sys.setTimeout = time;
    // do the work
    sys.deadLine = time;
} catch (DeadlineMiss dm) {
    // handle deadline miss
}
```

The first statement queries the current time and adds the maximum allowed execution time (which equals the deadline). Setting of the timeout value starts the timer that will throw a hardware generated exception (DeadlineMiss) when not reset. At the end of the code block under timer control, the access to the deadLine register will stop the timeout counter and stall until the deadline is reached.

B. Implementation

We have implemented a semantic equivalent to the deadline instruction as proposed in [12]. Instead of changing the instruction set of JOP, we have implemented an I/O device for the cycle accurate delay. The time value for the delay until is written to that I/O device and the device delays the acknowledge signal until the deadline. Therefore, the write instruction is delayed. This simple device is independent of the processor and can be used in any architecture where an I/O request needs an acknowledgement.

The deadline device is implemented within the system I/O device that contains a cycle counter, a 1 MHz counter, the timer interrupt logic, and the watchdog. The deadline device uses the cycle counter. Therefore, the timing granularity is a single clock cycle.

I/O devices on JOP are mapped to so called *hardware objects* [21]. A hardware object represents an I/O device as a plain Java object. Field read and write access are actual I/O register read and write access. The following code shows how to obtain a hardware object for the system device, read out the current processor clock tick, and perform the deadline operation.

```
SysDevice sys = IOFactory.getFactory().getSysDevice();

int time = sys.cntInt;
time += 1000;
sys.deadLine = time;
```

The first instruction requests a reference to the system device hardware object. This object (sys) is accessed to read out the current value of the clock cycle counter. The deadline is set to 1000 cycles after the current time and the assignment sys.deadline = time writes the deadline time stamp into the I/O device and blocks until that time.

Deadline miss handlers can be implemented on JOP with the programmable timer interrupt. The interrupt handler can be implemented in Java as a Runnable() [20].

C. Deadline Mechanism in Software

The deadline mechanism can be implemented on standard processors completely in software as long as a timer tick or clock cycle counter is available. A simple busy wait till the deadline expires gives the best resolution. The following code, equivalent to the former code fragment, shows this busy loop in the context of JOP:

```
SysDevice sys = IOFactory.getFactory().getSysDevice();

int time = sys.cntInt;
time += 1000;
while (time - sys.cntInt >= 0) {
    ;
}
```

The subtraction of the current counter value from the timeout value and the comparison against 0 allows using an overflowing counter. For a counter width of n bits, the maximum relative timeout is $2^{n-1}-1$ cycles.

With a busy waiting loop, polling the clock cycle counter, the jitter of the deadline will be in the order of a few cycles. Some of the example use cases of the deadline mechanism, as described in the next section, will not tolerate this jitter.

Implementation of deadline monitoring on standard processors requires a timer that can raise an interrupt on expiration. Depending on the operating system, the actual execution latency of the miss handler can be in the order of several thousand cycles [20].

V. Usage Examples

A cycle accurate delay – the deadline instruction – can be used for many different applications: e.g., generation of timed I/O operations, WCET measurements of code fragments running on a TDMA based CMP system, using time for communication synchronization, and synchronizing tasks to

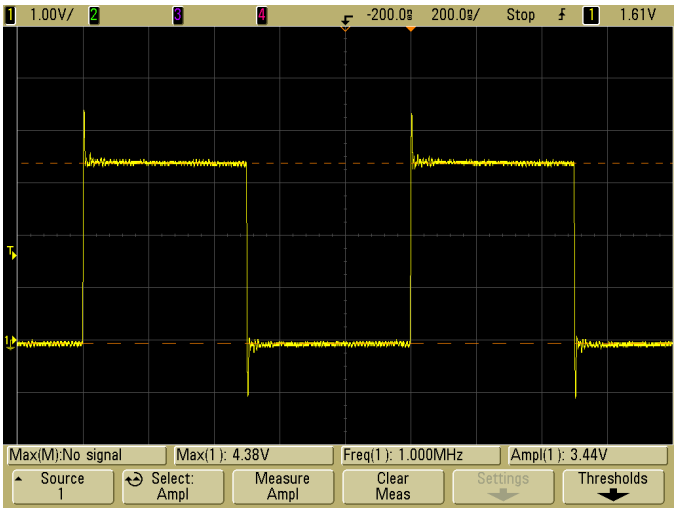

Fig. 1. A 1 MHz signal generated in software with the help of a deadline instruction

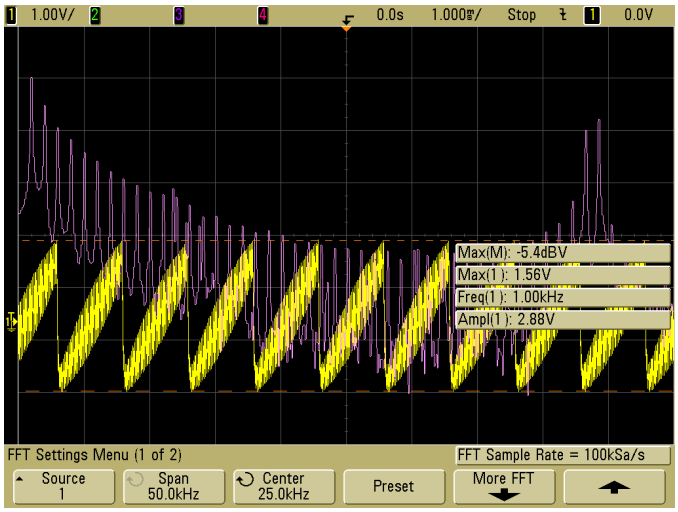

Fig. 2. A 1 kHz Sawtooth signal and the analyzed spectrum up to 50 kHz

a TDMA based arbiter for single path programming. We describe a few application examples in this section.

A. Evaluation

As a first test of the achievable timing resolution we generate a 1 MHz signal and measure the resulting frequency with an Agilent Technologies DSO6034A oscilloscope. Figure 1 shows the result. From the measurement we see that the generated frequency is exact (within the measurement resolution) and there is no jitter.

B. Audio Payback

For the evaluation of the deadline instruction we have implemented a playback of audio signals. To challenge the implementation the audio samples are not sent to an off-chip DAC, but the DAC is completely implemented in software. With the help of the deadline instruction the application generates a cycle accurate pulse-width modulation (PWM) of the the audio signal. The signal is output on an I/O pin of the processor and a simple, passive 2nd order low-pass filter converts the PWM signal to the audio signal. Without any further amplification we use a standard head-set to evaluate the quality of the signal (listen to the music).

The code fragment in Listing 2 shows the DAC loop. The loop contains two deadline instructions: the first one generates the sample period and the timeout value is increased in each iteration by the period in clock cycles; the second deadline instruction depends on the sample value and shifts the 1 to 0 transition within the period.

As a first audio experiment we generate a 1 kHz sawtooth signal. Figure 2 shows the signal in yellow and the spectrum in magenta. The signal is sampled at 100 kHz by the oscilloscope. The spectrum is displayed logarithmic with 10 dB per octave, and 5 kHz per unit. A sawtooth signal contains even and odd harmonics with an amplitude of A/n for the harmonic n, which can be seen in the spectrum. Furthermore, we see

Listing 2. A software DAC generating a PWM signal with the deadline instruction

```
final static int PERIOD = CLK_FREQ/44100;

for (;;) {
    time += PERIOD;                    // converter period
    off  = time + getSample();
    sys.deadLine = time;
    pwm.port = 1;                      // high output
    sys.deadLine = off;
    pwm.port = 0;                      // low output
}
```

two peaks around 44 kHz that represent the sample signal of 44.1 kHz. In the time domain we can see the sample signal superimposed on the sawtooth signal. It is out of the audio band and therefore not heard. However, due to non-linearity of the speakers, difference signals in the audio band can be produced. To avoid this interfering signal a higher order low-pass is needed or the sample frequency has to be increased with over-sampling. The resulting lower resolution (due to over-sampling) of the individual samples can be compensated with error propagation to the following samples, similar to the Floyd-Steinberg algorithm for image dithering [6].

Implementing over-sampling for audio signals in software is probably beyond the processing power of JOP. However, for low-frequency DACs, which are used in control applications, over-sampling and the use of the deadline instruction is a valuable option.

C. Instruction Measurement

Pitter has implemented a CMP version of JOP with a TDMA based memory arbiter [15]. A static memory access schedule removes any time dependencies between tasks executed on different processor cores. With the known, static TDMA schedule the WCET of individual bytecodes that access memory can be analyzed. Pitter has extended the WCET tool of JOP [22] to

Listing 3. Bytecode execution time measurement on a TDMA based system

```
int a[] = new int[1];

// A 0.1s interval in multiple of the TDMA round plus 1
int shift = CLOCK_FREQ/10/(TDMA_LENGTH)*TDMA_LENGTH+1;

// get measurement overhead
int time = Native.rd(Const.IO_CNT);
time = Native.rd(Const.IO_CNT)-time;
int off = time;

int start = sys.cntInt + shift;
for (int i=0; i<TDMA_LENGTH; ++i) {
    sys.deadLine = start;
    // measurement start
    time = Native.rd(Const.IO_CNT);
    a[0] = 1;
    // measurement stop
    time = Native.rd(Const.IO_CNT)-time;
    System.out.println(time-off);
    start += shift;
}
```

include the TDMA schedule. Examples of bytecode execution times, dependent on the number of processor cores and the time slot length, are given in the appendix of his PhD thesis.

Static WCET analysis is the preferred methodology for real-time systems. However, experimental evaluation of the static analysis tool is needed to build confidence that the static analysis is correct. In the case of execution time of individual bytecodes, the experimental validation is not trivial. The actual execution time depends on the phasing between the start of the instruction and the TDMA arbitration schedule. A guaranteed execution and measurement of all possible phases of the bytecode under investigation just with a test program is practically impossible. Tests with random delays can give some confidence to the static analysis.

With a deadline instruction the generation of test cases of all possible phases is trivial. The code segment under investigation is started after a deadline instruction. The start time of a measurement is shifted, with respect to the TDMA schedule, one cycle each iteration – for a TDMA round of n cycles, the start time is a multiple of $n+1$ cycles. After n measurements all possible phase relationships have been evaluated and the WCET of this code fragment is measured.

Listing 3 shows the measurement of the execution time of bytecode iastore at all possible TDMA phases. For a three core CMP with a TDMA slot length of 6 cycles the TDMA schedule (n) is 18 cycles. The measured execution times of the code fragment are between 27 and 44 clock cycles. The measured code consists of following bytecodes:

```
iconst_0
iconst_1
iastore
```

The first two bytecodes execute in a single cycle – they don't access main memory. Therefore, the array store instruction takes between 25 and 42 clock cycles. The static WCET analysis of iastore for the measured CMP configuration results in 41 clock cycles. This is one cycle lower than the measured WCET. Therefore, we have to investigate the timing model of the JOP CMP configuration to find this error.

D. Single-Path Programs on a CMP System

Single-path programming is a methodology to eliminate all data dependent control decisions [17]. If-conversion with predicated instructions and loops with constant bounds are the key elements. Execution of single-path programs on not too complex processors results in constant execution time. Therefore, static WCET analysis can be substituted by measurements. In [10] it has been shown that even processors with instruction caches deliver constant, and therefore repeatable, execution timing.

JOP, in a single processor setting, is time-predictable enough for the single-path programming paradigm. However, in a CMP configuration the execution time depends on the phasing of the task relative to the TDMA memory access schedule. The deadline instruction can be used to synchronize the task execution start with the TDMA schedule. Each start of a task has to be delayed till a start time that is a multiple of the TDMA schedule. For example, if three processors are used with a TDMA slot of 4 cycles, tasks are only allowed to start at multiples of 12 cycles. For single-path programming it does not matter at what phasing, relative to the TDMA schedule, the task starts as long as it starts every time at the same phasing. In that case the execution time can simply be measured. An evaluation of this approach is described in [23].

VI. Conclusion

An application visible deadline instruction brings timing semantics into the application program. We have implemented a simple I/O device for a cycle accurate delay of the processor pipeline in the context of the Java processor JOP. The I/O device delays a write access to the timing port till the expiration of the target time. It can be used in any processor that stalls on an I/O access operation till the I/O acknowledgement.

The deadline mechanism can be used for various tasks. For example, it allows generation of cycle accurate I/O operations as we have shown with a software DAC to generate audio signals. We have also evaluated the deadline instruction in the context of a CMP system with a TDMA based memory arbiter. With this experiment we found an error in the execution time model for the JOP CMP system, which we have to further investigate. Furthermore, application tasks can be synchronized to the TDMA schedule with a cycle accurate delay.

References

[1] Aravindh Anantaraman, Kiran Seth, Kaustubh Patil, E. Rotenberg, and F. Mueller. Virtual simple architecture (visa): exceeding the complexity limit in safe real-time systems. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, volume 31, 2 of *Computer Architecture News*, pages 350–361, New York, June 9–11 2003. ACM Press.

[2] S. Andalam, P. Roop, A. Girault, and C. Traulsen. PRET-C: A new language for programming precision timed architectures. Technical Report 6922, 2009.

[3] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[4] Stephen A. Edwards and Edward A. Lee. The case for the precision timed (PRET) machine. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 264–265, New York, NY, USA, 2007. ACM.

[5] Stephen A. Edwards and Edward A. Lee. The case for the precision timed (pret) machine. In *Proceedings of the 44th annual conference on Design automation*, pages 264 – 265. SESSION: Wild and crazy ideas (WACI), June 2007.

[6] Robert W. Floyd and Louis Steinberg. An adaptive algorithm for spatial greyscale. *Proceedings of the Society for Information Display*, 17(2):75–77, 1976.

[7] Thomas Henties, James J. Hunt, Doug Locke, Kelvin Nilsen, Martin Schoeberl, and Jan Vitek. Java for safety-critical applications. In *2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert 2009)*, York, United Kingdom, Mar. 2009.

[8] Benedikt Huber. Worst-case execution time analysis for real-time Java. Master's thesis, Vienna University of Technology, Austria, 2009.

[9] Nicholas Jun Hao Ip and Stephen A. Edwards. A processor extension for cycle-accurate real-time software. In *Proceedings of the IFIP International Conference on Embedded and Ubiquitous Computing (EUC)*, volume 4096, pages 449–458, Seoul, Korea, August 2006.

[10] Raimund Kirner and Peter Puschner. Time-predictable task preemption for real-time systems with direct-mapped instruction cache. In *Proc. 10th IEEE International Symposium on Object-oriented Real-time distributed Computing*, Santorini Island, Greece, May 2007.

[11] Edward A. Lee. Computing needs time. *Commun. ACM*, 52(5):70–79, 2009.

[12] Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D. Patel, Stephen A. Edwards, and Edward A. Lee. Predictable programming on a precision timed architecture. In Erik R. Altman, editor, *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2008)*, pages 137–146, Atlanta, GA, USA, October 2008. ACM.

[13] Isaac Liu, Ben Lickly, Hiren D. Patel, and Edward A. Lee. Poster abstract: Timing instructions - ISA extensions for timing guarantees. IEEE Real-Time and Embedded Technology and Applications Symposium, April 2009.

[14] David May. XMOS XS1 Architecture, 2008.

[15] Christof Pitter. *Time-Predictable Java Chip-Multiprocessor*. PhD thesis, Vienna University of Technology, Austria, 2009.

[16] Christof Pitter and Martin Schoeberl. A real-time Java chip-multiprocessor. *Trans. on Embedded Computing Sys.*, accepted for publication 2009.

[17] Peter Puschner and Alan Burns. Writing temporally predictable code. In *Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, pages 85–94, Washington, DC, USA, 2002. IEEE Computer Society.

[18] Martin Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.

[19] Martin Schoeberl. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems*, vol. 2009, Article ID 758480:17 pages, 2009.

[20] Martin Schoeberl, Stephan Korsholm, Tomas Kalibera, and Anders P. Ravn. A hardware abstraction layer in Java. *Trans. on Embedded Computing Sys.*, accepted, 2009.

[21] Martin Schoeberl, Stephan Korsholm, Christian Thalinger, and Anders P. Ravn. Hardware objects for Java. In *Proceedings of the 11th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2008)*, Orlando, Florida, USA, May 2008. IEEE Computer Society.

[22] Martin Schoeberl and Rasmus Pedersen. WCET analysis for a Java processor. In *Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2006)*, pages 202–211, New York, NY, USA, 2006. ACM Press.

[23] Martin Schoeberl, Peter Puschner, and Raimund Kirner. A single-path chip-multiprocessor system. In *Proceedings of the Seventh IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2009)*, number LNCS 5860, pages 47–57. Springer, November 2009.