A Single-Path Chip-Multiprocessor System

Martin Schoeberl, Peter Puschner, and Raimund Kirner

Institute of Computer Engineering
Vienna University of Technology, Austria
mschoebe@mail.tuwien.ac.at, {peter,raimund}@vmars.tuwien.ac.at

Abstract. In this paper we explore the combination of a time-predictable chip-multiprocessor system with the single-path programming paradigm. Time-sliced arbitration of the main memory access provides time-predictable memory load and store instructions. Single-path programming avoids control flow dependent timing variations. To keep the execution time of tasks constant, even in the case of shared memory access of several processor cores, the tasks on the cores are synchronized with the time-sliced memory arbitration unit.

1 Introduction

As more and more speedup features are added to modern processors and we are moving from single-core to multi-core processor systems, the analysis of the timing of the applications running on these systems is getting increasingly complex. The timing of single tasks per se is difficult to understand and to analyze. Besides that, task timing can no longer be considered as an isolated issue in such systems as the competition for shared resources and interferences via the state of the shared hardware lead to mutual dependencies of the progress and timing of different tasks.

We are convinced that the only way of making these highly complex processing systems time predictable is to impose some restrictions on their architecture and on the way in which the mechanisms of the architecture are used. So far we have worked along two main lines of research aiming at real-time processing systems with predictable timing:

On the software side we have conceived the *single-path execution* strategy [1]. The single-path approach allows us to translate task code in a way that the resulting code has exactly one execution trace that all executions of the task have to follow. To this end, the single-path conversion eliminates all input-dependent control flow decisions – by applying a set of code transformations [2] and if-conversion [3] it translates all input-dependent alternatives (i.e., code with if-then-else semantics) into straight-line predicated code. Loops with input-dependent termination are converted into loops that are semantically equivalent but whose iteration count is fully determined at system construction time.

Architecture-wise we have been working on time-predictable processors and chip-multiprocessor (CMP) systems. We have developed the JOP prototype of a *time-predictable processor* [4] and built a CMP system with a number of JOP cores [5]. In this multiprocessor system a static time-division multiple access (TDMA) arbitration scheme controls the accesses of the cores to the common memory. The pre-planning of

memory access schedules eliminates the need for dynamic conflict resolution and guarantees the temporal isolation that is necessary to allow for an independent progression of the computations on the CMP cores.

So far, we have dealt with each of the two topics in separation. This paper is the first that describes our work on *combining* the concepts of the *single-path approach and our time-predictable CMP architecture*. We thus present an execution environment that provides both temporal predictability to the highest degree and the performance benefits of parallel code execution on multiple cores. By generating deterministic single-path code, running this code on predictable processor cores, and using a rigid, pre-planned scheme to access the global memory we manage to achieve completely stable, and therefore predictable execution times for each single task in isolation as well as for entire applications consisting of multiple cooperating tasks running on different cores. To the best of our knowledge this has not been achieved for any other state-of-the-art CMP system so far.

2 The Single-Path Chip-Multiprocessor System

The main goal of our approach is to build an architecture that provides a combination of good performance and high temporal predictability. We rely on chip-multiprocessing to achieve the performance goal and on an offline-planning approach to make our system predictable. The idea of the latter is to take as many control decisions as possible before the system is actually run. This reduces the number of branching decisions that need to be taken during system operation, which, in turn, causes a reduction of the number of possible action sequences with possibly different timings that need to be considered when planning respectively evaluating the system's timely operation.

2.1 System Overview

We consider a CMP architecture that hosts n processor cores, as shown in Figure 1. On each core the execution of simple tasks is scheduled statically as cyclic executive. All core's schedulers have the same major cycle that is synchronized to the shared memory arbiter. Each of the processors has a small local method cache (M$) for storing recently used methods, a local stack cache (S$), and a small local scratchpad memory (SPM) for storing temporary data. The scratchpad memory can be mapped to thread local scopes [6] for integration into the Java programming language.

All caches contain only thread local data and therefore no cache coherence protocol is needed. To avoid cache conflicts between the different cores our CMP system does not provide a shared cache. Instead, the cores of the time-predictable CMP system access the shared main memory via a TDMA based memory arbiter with fine-grained statically-scheduled access.

2.2 TDMA Memory Arbiter

The TDMA based memory arbiter provides a static schedule for the memory access. Therefore, access time to the memory is independent of tasks running on other cores. In

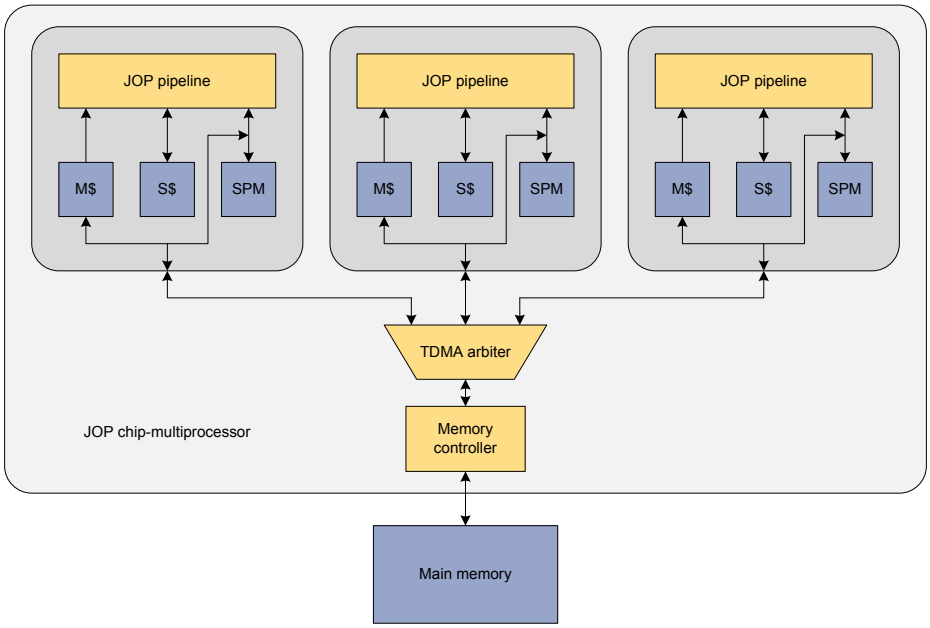

Fig. 1. A JOP based CMP system with core local caches (M$, S$) and scratchpad memories (SPM), a TDMA based shared memory arbiter, and the memory controller.

the default configuration each processor cores has an equally sized slot for the memory access. The TDMA schedule can also be optimized for different utilizations of processing cores. In [7] we have optimized the TDMA schedule to distribute slack time of tasks to other tasks with a tighter deadline.

The worst-case execution time (WCET) of a memory loads or stores can be calculated by considering the worst-case phasing of the memory access pattern relative to the TDMA schedule [8]. With single-path programming, and the resulting static memory access pattern, the execution time of tasks on a TDMA based CMP system is almost constant. The only jitter results from different phases of the task start time to the TDMA schedule.

The maximal execution time jitter, due to different phases between the task start time and the TDMA schedule, is the length of the TDMA round minus one. Thus, the TDMA arbiter very well supports time-predictable program execution. The maximal jitter due to TDMA delays is bounded and relatively small. And if one is interested to even completely avoid this short bounded execution time jitter, this can be achieved by synchronizing the task start with the TDMA schedule, using the deadline instruction described in Section 3.2.

2.3 Tasks

All tasks in our system are periodic. Tasks are considered to be simple tasks according to the *Simple-Task Model* introduced in [9]:[1] Task inputs are assumed to be available when a task instance starts, and outputs become ready for further processing upon completion of a task execution. Within its body a task is purely functional, i.e., it does neither access common resources nor does it include delays or synchronization operations.

To realize the simple-task abstraction, a task implementation actually consists of a sequence of three parts: *read inputs – execute – write outputs*. While the application programmer must provide the code for the *execute* part (i.e., the functional part), the first and the third part can be automatically generated from the description of the task interface. These *read* and *write* parts of the task implementations copy data between the shared state and task-local copies of that state. The local copies can reside in the common main memory or in the processor-local scratchpad memory. The placement depends on the access frequency and size of the local state. Care must be taken to schedule the data transfers between the local state copy and the global, shared state such that all precedence and mutual exclusion constraints between tasks are met. This scheduling problem is very similar to the problem of constructing static scheduling tables for distributed hard real-time computer systems with TDMA message scheduling in which task execution has to be planned such that task-order relations are obeyed and the message and task sequencing guarantees that all communication constraints are met. A solution to this scheduling problem can be found in [10].

Following our strategy to achieve predictability by minimizing the number of control decisions taken during runtime, all tasks are implemented in single path code. This means, we apply the single-path transformation described in [1, 2] to (a) serialize all input-dependent branches and (b) transform all loops with input-dependent termination into loops with a constant iteration count. In this way, each instance of a task executes the same sequence of instructions and has the same temporal access pattern to instructions and data.

2.4 Mechanisms for Performance and Time Predictability

By executing tasks on different cores with some local cache and scratchpad memory we manage to increase the system's performance over a single-processor system. The following mechanisms make the operation of our system highly predictable:

- Tasks on a single core are executed in a cyclic executive, avoiding cache influences due to preemption.
- Accesses to the global shared memory are arbitrated by a static TDMA memory arbitration scheme, thus leaving no room for unpredictable conflict resolution schemes and unknown memory access times.
- The starting point of all task periods and the starting point of the TDMA cycle for memory accesses are synchronized, and each task execution starts at a pre-defined offset within its period. Further, the single-path task implementation guarantees a

[1] More complex task structures can be simulated by splitting tasks into sets of cooperating simple tasks.

unique trace of instruction and memory accesses. All these properties taken together allow for an exact prediction of instruction execution times and memory access times, thus making the overall task timing fully transparent and predictable.
- As the *read* and *write* sections of the tasks may need more than a single TDMA slot for transferring their data between the local and the global memory, *read* and *write* operations are pre-planned and executed in synchrony with the global execution cycle of all tasks.

Besides its support for predictability, our planning-based approach allows for the following optimizations of the TDMA schedules for global memory accesses. These optimizations are based on the knowledge available at the planning time:

- The single-path implementation of tasks allows us to exactly spot which parts of a task's *execute* part need a higher and which parts need a lower bandwidth for accessing the global memory (e.g., a task does not have to fetch instructions from global memory while executing a method that it has just loaded into its local cache). This information can be used to adapt the memory access schedule to optimize the overall performance of memory accesses. While an adaption of memory-access schedules to the bandwidth requirements of different processing phases has been proposed before [11, 12], it seems that this technique can provide its maximum benefit when applied to single-path code – only the execution of single-path code yields a unique, and therefore fully predictable sequence and timing of memory accesses.
- A similar optimization can be applied to optimize the timing of memory accesses during the *read* and *write* sections of the task implementations. These sections access shared data and should therefore run under mutual exclusion. Mutual exclusion is guaranteed by the static, table-driven execution regime of the system. Still, the critical sections should be kept short. The latter could be achieved by an adaption of the TDMA memory schedule that assigns additional time slots to tasks at times when they perform memory-transfer operations.

Our target is a time-deterministic system, which means that not only the value of a function is deterministic, but also the execution time. It is desirable to exactly know which instruction is executed at each point in time. Execution time shall be a *repeatable* and *predictable* property of the system [13].

3 Implementation

The proposed design is evaluated in the context of the Java optimized processor (JOP) [4] based CMP system [5]. We have extended JOP with two instructions: a predicated move instruction for single-path programming in Java and a deadline instruction to synchronize application tasks with the TDMA based memory arbiter.

3.1 Conditional Move

Single path programming substitutes control decisions (if-then-else) by predicated move instructions. To avoid execution time jitter, the predicated move has to have a

constant execution time. On JOP we have implemented a predicated move for integer values and references. This instruction represents a new, system specific Java virtual machine (JVM) bytecode. This new bytecode is mapped to a *native* function for access from Java code. The semantic of the function

```
result = Native.condMove(x, y, b);
```

is equivalent to

```
result = b ? x : y;
```

without the need for any branch instruction. The following listing shows usage of conditional move for integer and reference data types. The program will print 1 and true.

```
String a = "true";
String b = "false";
String result;
int val;

boolean cond = true;

val = Native.condMove(1, 2, cond);
System.out.println(val);
result = (String) Native.condMoveRef(a, b, cond);
System.out.println(result);
```

The representation of the conditional move as a native function call has no call overhead. The function is substituted by the system specific bytecode during link time (similar to function inlining).

3.2 Deadline Instruction

In order to synchronize a task with the TDMA schedule a wait instruction with a resolution of single clock cycles is needed. We have implemented a deadline instruction as proposed in [14]. The deadline instruction stalls the processor pipeline until the desired time in clock cycles.

To avoid a change in the execution pipeline we have implemented a semantic equivalent to the deadline instruction. 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 absolute delay is written to the I/O device and the device delays the acknowledgment of the I/O operation until the cycle counter reaches this value. This simple device is independent of the processor and can be used in any architecture where an I/O request needs an acknowledgment.

I/O devices on JOP are mapped to so called *hardware objects* [15]. 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 accesses. The following code shows the usage of the deadline I/O device.

```
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.

4 Evaluation

We evaluate our proposed system within a Cyclone EP1C12 field-programmable gate array that contains 3 processor cores and 1 MB of shared memory. The shared memory is an SRAM with 2 cycles read access time and 3 cycles write access time. Some bytecode instructions contain several memory accesses (e.g., an array access needs three memory reads: read of the array size for the bounds check, an indirection through a forwarding handle,[2] and the actual read of the array value). For several bytecode instructions the WCET is minimized with a slot length of 6 cycles. The resulting TDMA round for three cores is 18 cycles.

As a first experiment we measure the execution time of a short program fragment with access to the main memory. Without synchronizing the task start with the TDMA arbiter we expect some jitter. To provoke all possible phase relations between the task and the TDMA schedule the deadline instruction was used to shift the task start relative to the TDMA schedule. The resulting execution time varies between 342 and 359 clock cycles. Therefore, the maximum observed execution time jitter is the length of the TDMA round minus one (17 cycles).

With the deadline instruction we make each iteration of the task start at multiples of the TDMA round (18 clock cycles in our example). In that case each task executes for a cycle accurate constant duration. This little experiment shows that single-path programming on a CMP system, synchronized with the TDMA based memory arbitration, results in *repeatable* execution time [13].

4.1 A Sample Application

To validate our programming model for cycle-accurate real-time computing, we developed a controller application that consists of five communicating tasks. This case study is a demonstrator that cycle-accurate computing is possible on a CMP system. Further, this case study give us some insights about the practical aspects of using the proposed programming model.

The architecture of the sample application is given in Figure 2. The application is demonstrative because of its rather complex inter-process communication pattern, which shows the need of precise scheduling decisions to meet the different precedence constraints. The application consists of the following tasks:

[2] The forwarding handle is needed for the implementation of the real-time garbage collector.

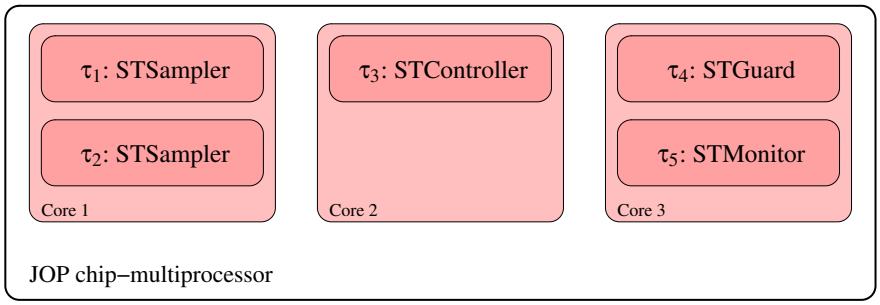

Fig. 2. Sample application: control application

- τ_1 and τ_2 are the sampling tasks that read from sensors. τ_1 samples the reference value and τ_2 samples the system value. This two tasks share the same code basis and they run at the double frequency than the controller task to allow a low-pass filtering by averaging the sensor values.
- τ_3 is the proportional-integral-derivative controller (PID controller) that gets the reference value from τ_1 and the feedback of the current system value from τ_2.
- τ_4 is a system guard similar to a watchdog timer that controls the liveness of τ_1, τ_2, and τ_3. Whenever the write phase of τ_1, τ_2, and τ_3 has not been executed between two subsequent activations of τ_4 then the system is set into an error state.
- τ_5 is a monitoring task that periodically collects the sensor values (from τ_1 and τ_2) and the control value (from τ_3). The write part of τ_5 is currently empty, but it can be used to include the code for transferring the collected system state to a host computer.

The inter-task communication of the sample application is summarized in Figure 3. It shows that this small application has a relatively complex communication pattern. Each task communicates with almost all other tasks. The communication pattern has a direct influence on the system schedule. The resulting precedence constraints have to be taken into account for scheduling the read, execute, and write phases for each task. And of course, since this is a CMP system, some of the task phases are executed in parallel, which complicates the search for a tight schedule.

Tasks τ_1-τ_5 are implemented in single-path code, thus their execution time does not depend on control-flow decisions. Since also the scheduler has a single-path implementation, the system executes exactly the same instruction sequence at each scheduling round.

All tasks are synchronized on each activation with the same phase of the TDMA based memory arbiter. Therefore, their execution time does not have any jitter due to different phase alignments of the memory arbiter.

With such an implementation style it is possible on the JOP to determine the WCET of each task directly by a single execution-time measurement (by enforcing either a cache hit or miss of the method). Table 1 shows the observed WCET values for each task, given separately for the read, execute, and write part of the tasks. The absolute

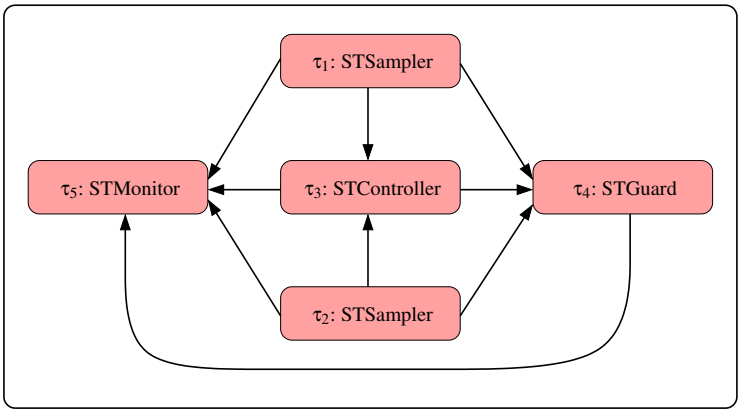

Fig. 3. Communication directions of the control application

Table 1. Measured single-path execution time in clock cycles

Task	Read	Execute	Write	Total
τ_1,τ_2	594	774	576	1944
τ_3	864	65250	576	66690
τ_4	26604	324	28422	55350
τ_5	1368	324	324	2016

WCET values are not that important to discuss, but more important is the fact that the execution time of each task is deterministic, not depending on the input data.

To summarize on the practical aspects of the programming model, it has shown that even this relatively simple application results in a scheduling problem that is rather tricky to be solved without tool support. For the purpose of our paper we solved it manually using a graphical visualization of the relative execution times and determining the activation times of each task manually. However, to successfully use this programming model for industrial production code, the use of a scheduling tool is highly advisable [10]. With respect to generating a tight schedule, it has shown that the predictable execution time of all tasks is very helpful.

5 Related Work

Time-predictable multi-threading is developed within the PRET project [14]. The processor cores are based on a RISC architecture. Chip-level multi-threading for up to six threads eliminates the need for data forwarding, pipeline stalling, and branch prediction. The access of the individual threads to the shared main memory is scheduled similar to our TDMA arbiter with the so called *memory wheel*. The PRET architecture implements the deadline instruction to perform time based, instead of lock based, synchronization for access to shared data. In contrast to our simple task model, where synchronization

is avoided due to the three different execution phases, the PRET architecture performs time based synchronization within the execution phase of a task.

The approach, which is closest related to our work, is presented in [11, 12]. The proposed CMP system is also intended for tasks according to the simple task model [9]. Furthermore, the local cache loading for the cores is performed from a shared main memory. Similar to our approach, a TDMA based memory arbitration is used. The paper deals with optimization of the TDMA schedule to reduce the WCET of the tasks. The design also considers changes of the arbiter schedule during task execution to optimize the execution time. We think that this optimization can be best performed when the access pattern to the memory is statically known – which is only possible with single-path programming. Therefore, the former approach to TDMA schedule optimization shall be combined with our single-path based CMP system.

Optimization of the TDMA schedule of a CMP based real-time system has been proposed in [7]. The described system proposes a single core per thread to avoid the overhead of thread preemption. It is argued that future systems will contain many cores and the limiting resource will be the memory bandwidth. Therefore, the memory access is scheduled instead of the processing time.

6 Conclusion

A statically scheduled chip-multiprocessor system with single-path programming and a TDMA based memory arbitration delivers repeatable timing. The repeatable and predictable timing of the system simplifies the safety argument: measurement of the execution time can be used instead of WCET analysis. We have evaluated the idea in the context of a time-predictable Java chip-multiprocessor system. The cycle accurate measurements showed that the approach is sound.

For the evaluation of the system we have chosen a TDMA slot length that was optimal for the WCET of individual bytecodes. If this slot length is also optimal for single-path code is an open question. In future work we will evaluate different slot lengths to optimize the execution time of single-path tasks. Furthermore, the change of the TDMA schedule at predefined points in time is another option we want to explore.

Acknowledgments

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement number 214373 (Artist Design) and 216682 (JEOPARD).

References

1. Puschner, P., Burns, A.: Writing temporally predictable code. In: Proc. 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems. (Jan. 2002) 85–91

2. Puschner, P.: Transforming execution-time boundable code into temporally predictable code. In Kleinjohann, B., Kim, K.K., Kleinjohann, L., Rettberg, A., eds.: Design and Analysis of Distributed Embedded Systems. Kluwer Academic Publishers (2002) 163–172 IFIP 17th World Computer Congress - TC10 Stream on Distributed and Parallel Embedded Systems (DIPES 2002).
3. Allen, J., Kennedy, K., Porterfield, C., Warren, J.: Conversion of Control Dependence to Data Dependence. In: Proc. 10th ACM Symposium on Principles of Programming Languages. (Jan. 1983) 177–189
4. Schoeberl, M.: A Java processor architecture for embedded real-time systems. Journal of Systems Architecture **54/1–2** (2008) 265–286
5. Pitter, C., Schoeberl, M.: A real-time Java chip-multiprocessor. Trans. on Embedded Computing Sys. accepted for publication. (2009)
6. Wellings, A., Schoeberl, M.: Thread-local scope caching for real-time Java. In: Proceedings of the 12th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2009), Tokyo, Japan, IEEE Computer Society (March 2009)
7. Schoeberl, M., Puschner, P.: Is chip-multiprocessing the end of real-time scheduling? In: Proceedings of the 9th International Workshop on Worst-Case Execution Time (WCET) Analysis, Dublin, Ireland, OCG (July 2009)
8. Pitter, C.: Time-predictable memory arbitration for a Java chip-multiprocessor. In: Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2008). (2008)
9. Kopetz, H.: Real-Time Systems. Kluwer Academic Publishers (1997)
10. Fohler, G.: Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In: Proceedings of the 16th Real-Time Systems Symposium. (December 1995) 152–161
11. Andrei, A., Eles, P., Peng, Z., Rosen, J.: Predictable implementation of real-time applications on multiprocessor systems on chip. In: Proceedings of the 21st Intl. Conference on VLSI Design. (Jan. 2008) 103–110
12. Rosen, J., Andrei, A., Eles, P., Peng, Z.: Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In: Proceedings of the Real-Time Systems Symposium (RTSS 2007). (Dec. 2007) 49–60
13. Lee, E.A.: Computing needs time. Commun. ACM **52**(5) (2009) 70–79
14. Lickly, B., Liu, I., Kim, S., Patel, H.D., Edwards, S.A., Lee, E.A.: Predictable programming on a precision timed architecture. In Altman, E.R., ed.: Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2008), Atlanta, GA, USA, ACM (October 2008) 137–146
15. Schoeberl, M., Korsholm, S., Thalinger, C., Ravn, A.P.: 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, IEEE Computer Society (May 2008)