Evaluation of resource arbitration methods for multi-core real-time systems*

Timon Kelter, Tim Harde, Peter Marwedel[1] and Heiko Falk[2]

1 **Department of Computer Science, TU Dortmund**
Otto-Hahn-Straße 16, D-44227 Dortmund
{timon.kelter,tim.harde,peter.marwedel}@tu-dortmund.de

2 **Institute of Embedded Systems/Real-Time Systems, Ulm University**
James-Franck-Ring, D–89081 Ulm
heiko.falk@uni-ulm.de

Abstract

Multi-core systems have become prevalent in the last years, because of their favorable properties in terms of energy consumption, computing power and design complexity. First attempts have been made to devise WCET analyses for multi-core processors, which have to deal with the problem that the cores may experience interferences during accesses to shared resources. To limit these interferences, the vast amount of previous work is proposing a strict *TDMA (time division multiple access)* schedule for arbitrating shared resources. Though this type of arbitration yields a high predictability, this advantage is paid for with a poor resource utilization. In this work, we compare different arbitration methods with respect to their predictability and average case performance. We show how known WCET analysis techniques can be extended to work with the presented arbitration strategies and perform an evaluation of the resulting ACETs and WCETs on an extensive set of realworld benchmarks. Results show that there are cases when TDMA is not the best strategy, especially when predictability and performance are equally important.

1998 ACM Subject Classification B.8.2 Performance Analysis and Design Aids, D.2.4 Software/Program Verification

Keywords and phrases WCET analysis, multi-core, arbitration, shared resources

Digital Object Identifier 10.4230/OASIcs.WCET.2013.1

1 Introduction

In the last years, many proposals on how to compute safe WCET values for programs running on multi-core systems have been made, but analyses that scale well and are precise at the same time are much harder in the multi-core case than in the single-core one. The central property, that must be accounted for, is that cores may access shared resources and these accesses will have to be arbitrated at some point. Among most of the analysis techniques that have been introduced, it is a common denominator that TDMA should be used for arbitrating the shared resources, since it allows an easy derivation of worst-case bounds for the duration of accesses. In this paper, we compare previously published arbitration methods experimentally both in terms of average and worst-case performance. For the experiments we use a static analyzer and a cycle-true system simulator on a multi-core ARM-platform with a configurable shared bus, which is arbitrated among the cores. We also examine a

* This work was partially supported by Deutsche Forschungsgesellschaft (DFG) under grant FA 1017/1-1 and EU COST Action IC1202: Timing Analysis On Code-Level (TACLe).

more flexible variant of TDMA, called *Priority Division*, and show its effects on ACET and WCET. In summary, the contributions are:

- Generalization of previously published WCET analysis techniques to the mentioned arbitration methods
- Experimental evaluation of average-case and worst-case properties of different arbitration methods on a realistic multi-core platform

The rest of the paper is organized as follows: In Section 2 we will present related work, and Section 3 introduces our system model used in the analyses. In Section 4 the overall analysis framework as well as the changes that are needed to incorporate the different arbitration methods are presented. Section 5 provides the experimental evaluation of the approaches. Finally, we provide a summary of our results and give directions for future work in Section 6.

2 Related Work

The initial scenario covered by WCET analyses was the case of a single program being run uninterruptedly on a single core. This is a well-understood problem for which structured analyses were devised [16]. For the WCET estimation to be safe the analyzer must usually operate on a binary of the analyzed program, since only then locations of code and data are fixed, which affects e.g. cache performance. The problem is separated into control-flow reconstruction, value analysis, microarchitectural analysis and path analysis. *Control-flow reconstruction* can be tackled by a combination of heuristics and data-flow analysis, whereas *value* and *microarchitectural analysis* are usually done as a pure data-flow analysis with the domain of register / memory values or abstract machine states, respectively. In these analyses it may be possible to throw away abstract states that are not inducing a local worst-case, but this is only possible if the architecture under examination is *timing-anomaly-free* [13]. For the last step in this WCET analysis pipeline, the *path analysis*, an integer linear program is formulated and solved, which models all paths through the program together with user-defined flow restrictions, which are needed to bound loops and recursions in the program.

In this work, we also conduct the WCET analysis in the presented way, where the modeling of accesses to shared resources is integrated into the microarchitectural analysis stage with the help of an existing approach for analyzing TDMA offsets [5, 1].

The Priority Division (PD) arbitration policy was first introduced in [14]. The authors provide experimental results of an FPGA-based system which uses PD to arbitrate the shared memory bus, and they state that PD is well-suited for WCET analysis, but this is not investigated further. In this paper we show how to integrate PD into existing WCET analysis frameworks.

The second major category of timing-analysis tools models the multicore system as a set of timed automata and performs model-checking of timing predicates to find the WCET [3]. The drawback of this approach, as discussed in [15], is the possible lack of scalability since the generated models quickly become intractably big. To overcome this, mixtures of abstract interpretation and model-checking have been proposed [8].

Finally, the last approach to deriving multicore WCET values it to derive arrival curves which bound events on the shared resources and compute worst-case access delays from those curves [11]. This has the advantage that the timing behavior of sequences of accesses can be analyzed more easily, therefore these approaches are also called “cumulative” in [2]. On the other hand, the abstraction towards access curves alone already loses some precision.

There have been numerous discussions of arbitration policies for real-time systems. The *PROMPT principles* for predictable multicore architectures [2] advocate "deterministic" arbitration with TDMA as one example. The comparison in [12] examined different arbitration methods for the shared bus of a Java real-time processor and TDMA arbitration resulted as the most predictable method. This tendency can be further supported by the long-standing work on the *Time-Triggered Architecture (TTA)* [6], where timing-predictable communication is implemented by customized TDMA schedules. The TTA and other works also define Networks-on-Chip (NoCs) which use TDMA to provide guaranteed delays. With minor adaptions, the analysis techniques presented in this paper are also applicable to such NoCs.

The adaptation of multicore hardware to exhibit better timing properties is related to our work, since we also propose a configurable arbiter for the shared resource, but we assume standard components for the rest of the system. In other works, architectures which can explicitly produce a measurable WCET [10] or which are at least highly predictable [7] are proposed. Compared to our architecture these approaches require more changes to the hardware.

3 System Model

The system architecture that we assume is sketched in Figure 1. It is built according to the PROMPT guidelines [2], and thus is composed from n fully timing-composable ARM7TDMI cores. Each core has local scratchpads for instructions and data and is connected to the shared memory via a shared bus with configurable arbitration logic. The shared memory finally holds RAM memories for instructions and data and a boot ROM from which the cores read their packed binaries during system startup. We have deliberately not included caches in the architecture to be able to focus on the effects of the employed arbitration methods. Otherwise, imprecisions in the cache analysis might be able to affect the results for the shared bus. Nevertheless, caches can be easily integrated into the analysis framework, since we follow the standard approach from [16]. The whole system was modeled in the cycle-accurate virtual prototyping IDE CoMET [4] to be able to perform detailed measurements. A custom implementation of the bus arbitration logic has been designed, to be able to track the bus utilization and the imposed access delays in detail.

All implemented arbitration methods share the assumptions that bus transactions are uninterruptible and that a maximum duration m_{max} of an access to any device behind the

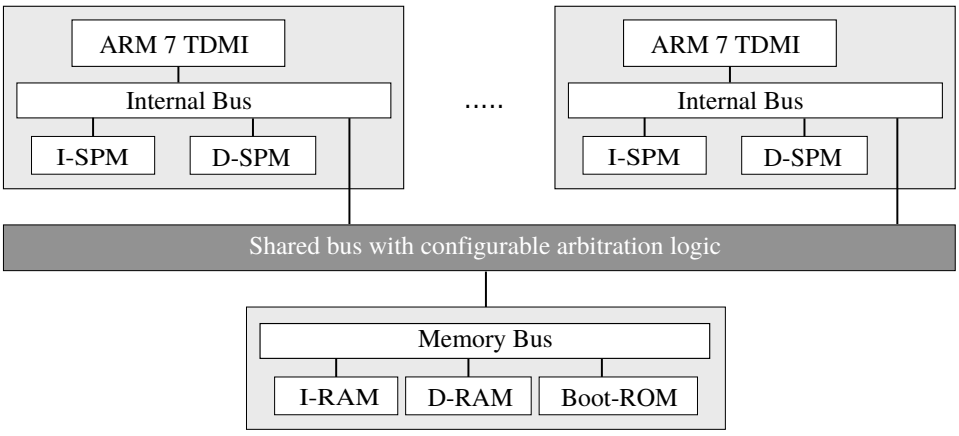

Figure 1 The employed system architecture (simplified).

bus is known. For simplicity of presentation we also assume that the bus runs at the same clock as the cores, but the offset analysis sketched in Section 4 can easily be extended to include a bus clock which is slower than the core clock, since it is tracking the schedule position in (higher precision) core ticks.

Fair arbitration (FAIR), also called *Round-Robin*, rotates the bus access among all cores. It maintains an *active core* $c_a \in \{0, \ldots, n-1\}$. When an access finishes c_a is advanced to the next core which requests the bus. Thus, each core can acquire the bus after at most $n-1$ other cores have performed their accesses.

Static priority-based arbitration (PRIO) assigns a unique priority $p_i \in \{1, \ldots, n\}$ to each core c_i and when there are multiple requests only the request from the core with the highest priority is granted. Nevertheless, since accesses are uninterruptible even the highest-priority core may have to wait until an ongoing transaction is completed.

TDMA creates a schedule consisting of n slots of size l and assigns an owner core $o_i \in \{0, \ldots, n-1\}$ to each slot. The current position in the schedule is determined by taking the current clock tick modulo nl. In each slot i only the owner is granted access to the bus and only in the interval $[il, \ldots, (i+1)l - m_{max}]$. The subtraction of m_{max} is necessary to make sure that accesses complete before the next slot begins.

Priority division (PD) is a generalization of TDMA. Instead of assigning an owner o_i it assigns unique priorities $p_{ij} \in \{0, 1, \ldots, n\}$ for each slot i and each core j. The bus is granted to the requesting core with the highest positive priority, only those with priority 0 are excluded from arbitration (this can be used to emulate TDMA behavior).

4 Analysis Framework

The general analysis framework is depicted in Figure 2. The analysis begins with a CFG reconstruction, which is based on pattern matching, and then does a simple register value analysis to identify memory access targets of load/store instructions. The main component, which is influenced by the choice of the arbitration policy, is the microarchitectural analysis. This stage computes bounds on the execution time of basic blocks. We sketch its architecture for the case with caches to show how caches can be integrated, even though our target architecture currently does not contain caches (see Section 3). The microarchitectural analysis associates an abstract pipeline state with each node in the CFG. Each abstract pipeline state is complemented by an abstract cache and bus state. The pipeline analysis is the driver of this stage: It simulates the possible processor actions on the abstract states and sends a request to the cache and bus stages whenever a memory access is performed (solid gray arrows in Figure 2). This request contains information about the target of the access and the timing of the access relative to the begin of the block's execution. If caches are present then the cache analysis may forward the request to the bus analysis, depending on whether the access is guaranteed to hit the cache or not. Similarly, the bus analysis receives the request together with its associated timing information and must decide how long it may take for this request to be granted the bus. We base this analysis on local abstract state information only, i.e. the bus analysis may not assume information about concurrently occurring accesses performed by other cores [1]. This means that for the worst-case we have to assume that concurrent accesses are occurring all the time. The bus and cache analyses must

[1] If concurrently occurring accesses are to be considered, a *Parallel Control Flow Graph* is needed together with analysis techniques which guarantee to cover all possible instruction and pipeline stage interleavings. These techniques, though principally possible, are prohibitively expensive since the enumeration of all possible interleavings leads to a state explosion [9].

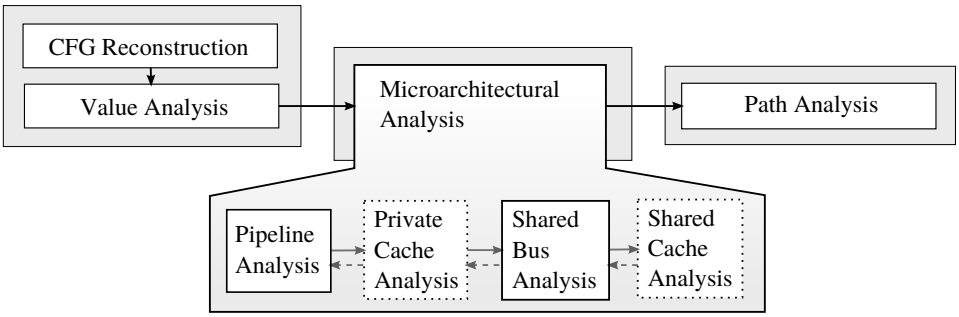

Figure 2 Analysis stages and their interaction.

then update the cache and bus state which is associated to the current abstract pipeline state and return an approximation of the timing behavior of the memory access to the pipeline analysis (dashed gray arrows in Figure 2). The updates of the basic block states (*transfer functions*) are integrated into a data-flow fixpoint iteration which converges in a safe approximation of the WCET for single executions of basic blocks (see [16]). This information is then used in the *path analysis* to compute the longest (shortest) path through the program whose length is the WCET (BCET). In the following we examine the abstract state and transfer functions for the shared bus analysis in more detail.

The *abstract bus state* for a basic block b as introduced in [5] is the set of *TDMA offsets* $O_b^{in} \subseteq \{0, \ldots, nl-1\}$, i.e. positions in the TDMA schedule with which the block execution may start. The bus analysis then computes multiple intermediate bus states O_b^i and finally a result state O_b^{out} that denotes the offsets after the block execution is finished. Initially, O_b^0 is set to O_b^{in}. The pipeline analysis then repeatedly hands accesses $a_i \in b, i \in \mathbb{N}$ to the bus analysis, together with an execution time set $T_{a_i} \in 2^{\mathbb{N}}$ that bounds the time that passed since access a_{i-1} (or the block start for $i = 1$). The bus analysis must then determine the set of possible memory access times D_{a_i} that the shared memory may need to serve this access. With these values the bus analysis computes $O_b^i = \mu_c(O_b^{i-1}, T_{a_i}, D_{a_i})$, where c is the core which issued request a_i. Also it must return a set of possible execution times $\delta_c(O_b^{i-1}, T_{a_i}, D_{a_i})$ to the pipeline analysis to describe the resulting timing for a_i.

By considering each input offset in separation we can define $\mu_c(O, T, D) = \bigcup_{o \in O, t \in T} \{\Phi_c(o + t \mod nl, D)\}$. $\Phi(o, D)$ determines the resulting offsets for an access at the offset o whose runtime is bounded by D.

$\Phi : \mathbb{N} \times 2^{\mathbb{N}} \to 2^{\mathbb{N}}$ must be defined for each arbitration method, to reflect the timing of the specific method. For TDMA and PD we define s_o as the slot containing offset o and s_c as the first slot after offset o where core c has maximum priority or which is owned by c. $s_c = s_o$ is explicitly allowed. With these definitions we have

$$\Phi_c^{TDMA}(o, D) = \begin{cases} \{o\} \oplus D & \text{if } o \in \omega(s_c) \\ \{s_c l\} \oplus D & \text{if } else \end{cases} \tag{1}$$

The first case in Equation 1 models an access inside slot s_c and the second case handles accesses outside of it. The operator $\oplus$ is defined by $\oplus(X, Y) = \{x + y | x \in X, y \in Y\}$ and $\omega(s_c) = \{s_c l, \ldots, (s_c + 1)l - m_{max}\}$ is a shorthand for the "grant window" of offsets inside

s_c where an access from core c will be immediately granted.

$$\Phi_c^{PD}(o,D) = \begin{cases} \{o\} \oplus \{0,\ldots,m_{max}\} \oplus D & \text{if } o \in \omega(s_c) \\ \left(\bigcup_{s_i \in \{s_o,\ldots,s_c-1\}} \phi_c(s_i,D) \right) \cup \Phi_c^{TDMA}(o,D) & \text{if } o \notin \omega(s_c) \\ \emptyset & \text{if } \nexists s_c \end{cases} \tag{2}$$

$$\phi_c(s,D) = \begin{cases} \{sl,\ldots,(s+1)l - m_{max}\} & \text{if } p_{sc} > 0 \\ \emptyset & \text{else} \end{cases} \tag{3}$$

The case structure for Φ_c^{PD} in Equation 2 is similar to the one for Φ_c^{TDMA}, but with the additional case that c is not the top-priority core in any slot (case three). Also it has to account for possibly ongoing transactions, even in slots in which c has the highest priority (case one). For accesses outside of s_c (case two) $\phi_c(s,D)$ as given in Equation 3 contributes all offsets which may result from the access being granted in a successive slot where c has positive but not the highest priority.

For fair and priority-based arbitration we can only supply conservative bounds, since these methods require knowledge about all possibly concurrently occurring transactions, which is hard to obtain in general, as noted above.

$$\Phi_c^{FAIR}(o,D) = \{o\} \oplus \{0,\ldots,nm_{max}\} \oplus D \tag{4}$$

$$\Phi_c^{PRIO}(o,D) = \begin{cases} \{o\} \oplus \{0,\ldots,m_{max}\} \oplus D & \text{if } \forall i \in \{1,\ldots,n\} : p_i \leq p_c \\ \emptyset & \text{else} \end{cases} \tag{5}$$

Therefore, Equations 4 and 5 are only stating that FAIR accesses may experience 0 to nm_{max} cycles delay. For PRIO accesses the two listed cases are symmetric to case 1 and case 3 in Equation 2 since PRIO is just a specialization of PD.

It is easy to extend the definition of Φ such that it does not only return the resulting offset but also the time that it took to perform the arbitration and the access. With this extension we can define δ_c similar to μ_c. In cases in which the original Φ returned an empty set, the runtime is defined to be $\{\infty\}$, which happens for accesses which have no bounded duration (e.g. PRIO).

As mentioned before, the μ_c and δ_c functions are then used in the transfer function of the microarchitectural analysis stage. The join function, which is used when control flow from at least two different predecessors joins at a basic block, is simply the set union of the incoming offset sets.

5 Evaluation

Due to the lack of standard multicore real-time benchmarks, we chose to execute independent tasks from the MRTC, UTDSP, MiBench, MediaBench and DSPstone benchmark suites on the single cores, amounting to 110 flow-fact-annotated, independent *benchmark tasks* in total. In the experiments, we grouped together benchmarks with similar runtime and executed *packages* with one benchmark per core. The packages were formed by sorting the benchmarks in the order of their single-core ACET and then having a window of size 1/2/4/8 slide over this list, collecting all 110/109/107/103 possible combinations. All cores start their assigned task synchronously and execution finishes when all tasks have been completed. Thus, since the benchmarks have different runtimes there will be some amount of inevitable *completion time jitter*, but apart from that this scenario models a system with high load. All

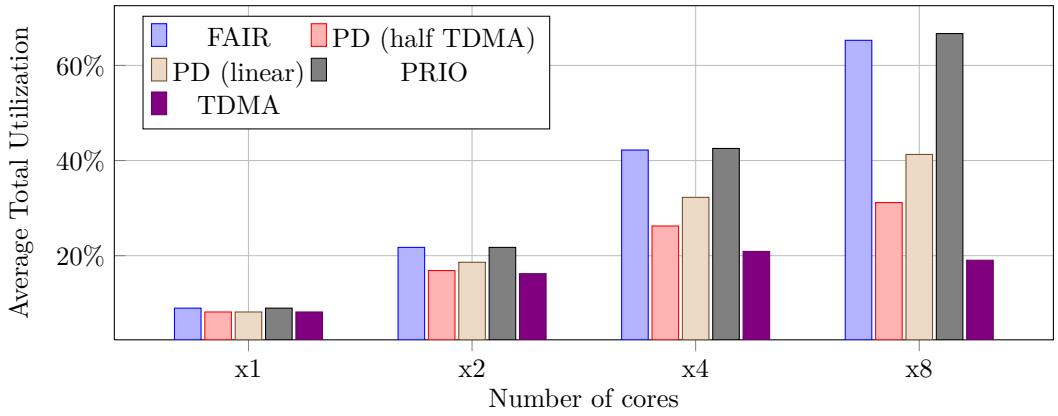

Figure 3 Average total bus utilization for different platforms.

the benchmarks read their inputs and store their outputs in the shared memory, whereas all program code as well as the stack was allocated in the scratchpad memory of the individual cores. This emulates the (reasonable) scenario that I/O is done via a shared device, whereas code and local data are kept in local memories for performance reasons. Also, all benchmarks were compiled with moderate optimization (optimization level O1). The memory access durations were set to 1 cycle for the scratchpads and $m_{max} = 3$ cycles for the shared memory. For most data-processing instructions the ARM7TDMI needs only 1 cycle, branches need 3 cycles and multiply instructions may need up to 5 cycles.

Concerning the parametrization of the arbitration methods we have selected simple heuristics to demonstrate some key impacts. For *PRIO* the priorities were assigned such that $t_i > t_j \Leftrightarrow p_i > p_j$ where t_i is the single-core runtime of the task mapped to core i. We use this strategy, also known as "largest job first", here to speed up long-running tasks and thus to decrease the completion time jitter. For *TDMA* (and also for *PD*) we set the slot size $l = m_{max}$ to keep delay times as small as possible. Our experiments have shown that higher slot lengths impose both higher WCET and ACET values. Also, for *TDMA* we set $o_i = i$ such that each core owns a single slot. For *PD (linear)* each slot i is "owned" by core i by setting $p_{ii} = n$. Priorities for all other cores are distributed in the same way as for *PRIO* i.e. in the order of single-core task runtime. The variation *PD (half TDMA)* for all slots $i \in \{0, \ldots, \lfloor n/2 \rfloor\}$ assigns $p_{ii} = 1$ and $\forall j \neq i : p_{ij} = 0$, thus effectively making these slots pure TDMA slots. All slots $i \in \{\lfloor n/2 \rfloor + 1, \ldots, n\}$ are configured in the same way as for *PD (linear)*.

Figure 3 shows the average[2] utilization resulting for different values of n. As expected, *FAIR* and *PRIO* show superior utilization that scales linearly with the number of cores, since these are *work-conserving* arbitration methods, i.e. as long as there are active requests they do not insert wait cycles. *TDMA* shows some increase in utilization with rising n, but it is stagnating at around 20% due to slots which remain unused by their owners. For $n = 8$ the utilization is actually *decreasing* again below 20%. *PD* is also not work-conserving, since it must delay requests when they cannot be served in the current slot, which may happen frequently for our setting of $l = m_{max}$. Still both *PD* configurations show a linear increase in utilization, with *PD (half TDMA)* being slightly behind *PD (linear)*. For *PD (linear)* the utilization is twice as high as for *TDMA*, which is also reflected in the average ACETs of the

[2] Since we only report relative values here, we use *average* as a synonym for the *geometric mean*.

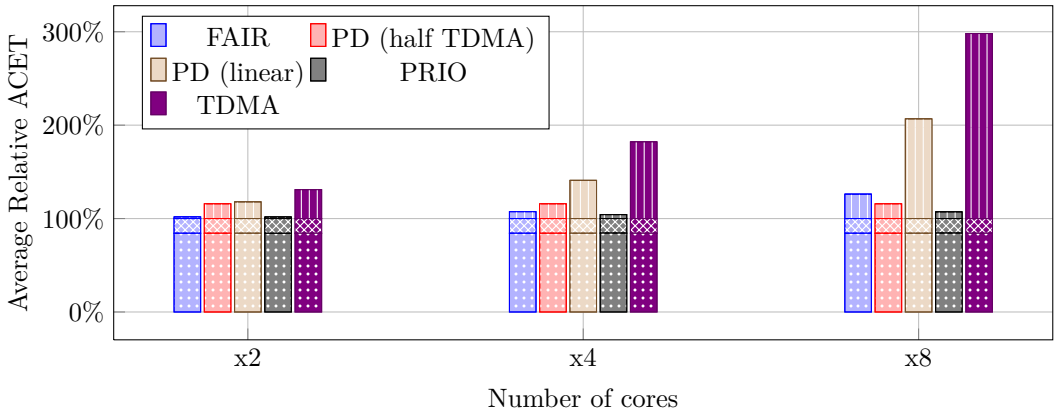

Figure 4 Average relative measured execution time (ACET) for different platforms (Baseline = execution time on single-core platform).

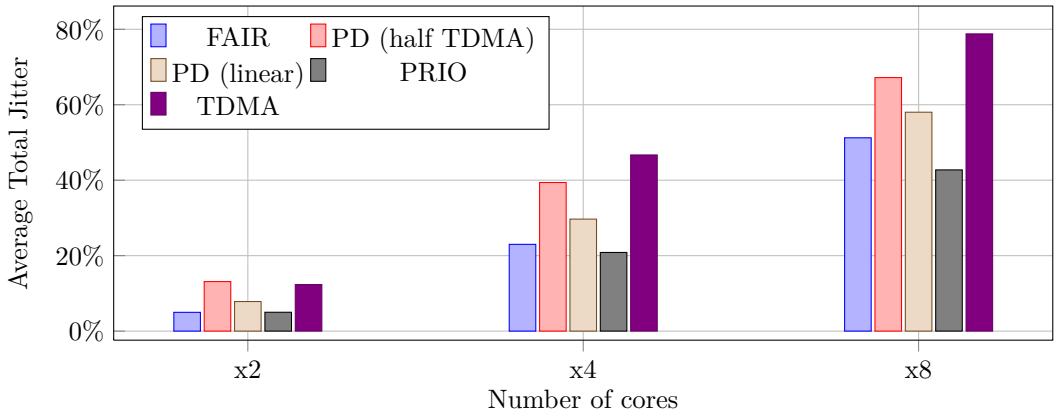

Figure 5 Average benchmark execution time jitter for different platforms.

benchmarks as shown in Figure 4.

In general, the ACET per task is inversely proportional to the achieved utilization values. The dotted areas in Figure 4 show the portion of the ACET which is used for computation and local memory accesses (stack and program code, see Section 3), the crosshatched areas show the portion in which the task is *using* the shared bus and the areas with vertical bars show the percentage of the ACET in which the task is *waiting* for the shared bus. For *TDMA* it becomes visible that e.g. for the configuration with 8 cores, the tasks are on average using more cycles for waiting than for performing computation and actual memory accesses.

The ACET and utilization values are influenced by the completion time jitter of the benchmark packages, that is the length of the time interval between the first termination of a benchmark on any core and the termination of the last benchmark. Especially for *TDMA* the jitter is problematic since it leaves slots of already terminated cores unused. Figure 5 shows the jitter as a percentage of the total runtime of the benchmark package (i.e. the runtime of the longest benchmark). It is visible that the low utilization values for *TDMA* are to some extent related to the rising jitter, but since this increase is itself triggered by the usage of *TDMA* this is an inherent drawback of the policy.

Finally, Figure 6 shows the average of the quotient of single-task WCET and single-task ACET, which is a bound on the maximum possible WCET overestimation. The highest

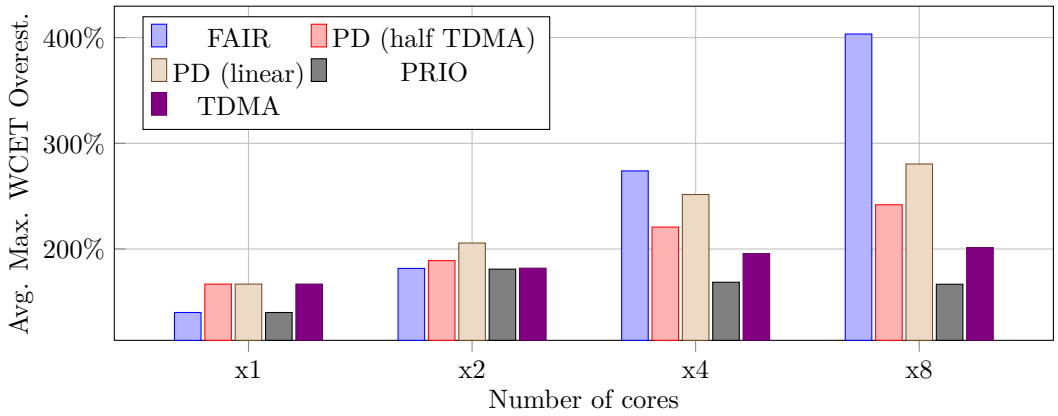

Figure 6 Average maximum task WCET overestimation for different platforms. Since the baseline corresponds to the measured ACET on the respective platform, the values represent maximum overestimation ratios.

WCET increases are attained by *FAIR* since it must always assume a worst-case delay of $(n-1)m_{max}$ cycles, but up to 2 cores *FAIR* is still competitive. Concerning the *PRIO* results in Figure 6, they are almost constant between the different configurations since here only the tasks for which a WCET can be determined at all are considered. This in all cases is only the highest-priority task whose WCET is increased by 40% to 81% on average, due to the presence of uninterruptible lower-priority accesses. *TDMA* shows the smallest overestimation ratio, but it is notable that both *PD* approaches are following very closely after *TDMA* in the WCET ranking. The small increase in WCET for e.g. *PD (linear)* is compensated by far better ACET and utilization values which makes *PD* a very appealing method for mixed-criticality systems.

6 Summary & Future Work

Current multi-core processors are mainly not timing-predictable due to a number of reasons, with one of them being accesses to shared resources. Since some amount of sharing is inevitable in multi-core systems we have demonstrated advantages and disadvantages of arbitration schemes for shared resources. Therefore, this work can serve as a basis for selecting a suitable arbitration policy, depending on the needs of the platform. For the first time, the drawbacks of TDMA in terms of average-case performance were quantified in comparison to other arbitration methods and it was shown, that priority division is a promising alternative for systems running mixed-criticality workloads, since it allows the fine-grained trading of ACET and bus utilization vs. WCET.

In the future, we would like to examine the effects of software optimizations to the achievable prediction accuracy, e.g. by grouping bus accesses or by restructuring tasks into read, execute and write phases. Another interesting perspective is the optimization of bus schedule parameters and the refinement of the presented multi-core WCET analysis.

7 Acknowledgments

We would like to thank Synopsys for the provision of the virtual prototyping IDE CoMET.

References

1 Sudipta Chattopadhyay, Abhik Roychoudhury, and Tulika Mitra. Modeling Shared Cache and Bus in Multi-cores for Timing Analysis. In *Proc. of SCOPES*, 2010.

2 Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza, Jan Reineke, Benoît Triquet, Simon Wegener, and Reinhard Wilhelm. Predictability Considerations in the Design of Multi-Core Embedded Systems. *Ingénieurs de l'Automobile*, 807:36–42, September 2010.

3 Andreas Gustavsson, Andreas Ermedahl, Björn Lisper, and Paul Pettersson. Towards WCET Analysis of Multicore Architectures using UPPAAL. In *Proc. of WCET*, July 2010.

4 Synopsys Inc. CoMET system engineering IDE. `http://www.synopsys.com/Systems/VirtualPrototyping/Pages/CoMET-METeor.aspx`.

5 Timon Kelter, Heiko Falk, Peter Marwedel, Sudipta Chattopadhyay, and Abhik Roychoudhury. Bus-Aware Multicore WCET Analysis through TDMA Offset Bounds. In *ECRTS*, pages 3–12, 2011.

6 Hermann Kopetz and Günther Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

7 Isaac Liu, Jan Reineke, David Broman, Michael Zimmer, and Edward A. Lee. A PRET Microarchitecture Implementation with Repeatable Timing and Competitive Performance. In *Proceedings of the International Conference on Computer Design*, ICCD 2012, October 2012.

8 Mingsong Lv, Wang Yi, Nan Guan, and Ge Yu. Combining Abstract Interpretation with Model Checking for Timing Analysis of Multicore Software. In *Proceedings of the Real-Time Systems Symposium*, RTSS '10, pages 339–349, Washington, DC, USA, 2010. IEEE Computer Society.

9 Robert Mittermayr and Johann Blieberger. Timing Analysis of Concurrent Programs. In Tullio Vardanega, editor, *12th International Workshop on Worst-Case Execution Time Analysis*, volume 23, pages 59–68, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

10 Marco Paolieri, Eduardo Quinones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. Hardware support for WCET analysis of hard real-time multicore systems. *SIGARCH Computer Architecture News*, 37(3):57–68, 2009.

11 R. Pellizzoni, A. Schranzhofer, Jian-Jia Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 741–746, 2010.

12 Christof Pitter and Martin Schoeberl. A real-time Java chip-multiprocessor. *ACM Transactions on Embedded Computing Systems*, 2009.

13 Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In *Proc. of WCET*, 2006.

14 H. Shah, A. Raabe, and A. Knoll. Priority division: A high-speed shared-memory bus arbitration with bounded latency. In *Proc. of DATE*, pages 1–4, 2011.

15 Reinhard Wilhelm. Why AI + ILP is good for WCET, but MC is not, nor ILP alone. In *Verification, Model Checking and Abstract Interpretation (VMCAI), LNCS 2937*, 2004.

16 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3), 2008.