An Overview of Approaches Towards the Timing Analysability of Parallel Architectures

Christine Rochange[1]

1 **Institut de Recherche en Informatique de Toulouse**
University of Toulouse, France
`rochange@irit.fr`

Abstract

In order to meet performance/low energy/integration requirements, parallel architectures (multi-threaded cores and multi-cores) are more and more considered in the design of embedded systems running critical software. The objective is to run several applications concurrently. When applications have strict real-time constraints, two questions arise: a) how can the worst-case execution time (WCET) of each application be computed while concurrent applications might interfere? b) how can the tasks be scheduled so that they are guarantee to meet their deadlines? The second question has received much attention for several years [4, 8]. Proposed schemes generally assume that the first question has been solved, and in addition that they do not impact the WCETs. In effect, the first question is far from been answered even if several approaches have been proposed in the literature. In this paper, we present an overview of these approaches from the point of view of static WCET analysis techniques.

1998 ACM Subject Classification C.3 [Special-purpose and application-based systems]: Real-time and embedded systems

Keywords and phrases WCET analysis, multicore, time predictability

Digital Object Identifier 10.4230/OASIcs.PPES.2011.32

1 Introduction

Parallel architectures, including multithreaded processors (MT) and multi-cores (MC), are being increasingly used in embedded systems because they fulfill various requirements like high performance, reduced energy consumption and thermal dissipation, and high integration. This is achieved through resource sharing among tasks: space sharing in instruction queues (MT) or caches (MT&MC), and time sharing in the pipeline (MT) or on the shared bus to the memory hierarchy (MC).
Now, in hard real-time systems, some tasks have strict deadlines and they must be carefully scheduled to meet them. Task scheduling algorithms rely on the knowledge of the WCET of each task. Research on timing analysis has been carried out for more than fifteen years. The proposed approaches range from testing techniques, that estimate the worst-case execution time from observed execution times (either on the target hardware or on a cycle-accurate simulator) which is clearly unsafe for critical software, to solutions based on static software analysis techniques that compute safe WCETs provided the model of the target hardware is correct. In this paper, we focus on static WCET analysis which is the most appropriate when considering hard real-time tasks but also the most sensible to non deterministic instructions timings.

Until recently, static WCET analysis has assumed that the task under analysis could not be impacted by any external event (either related to another task or to hardware-level devices like timer interrupts or memory refreshes). Unfortunately, resource sharing in a parallel architecture questions this assumption since it induces tasks interferences that are likely to impact instructions timings. Such interferences include conflicts to access a shared resource, which are solved by stalling all the requesting tasks but one, as well as corruption in memories when a task invalidates part of the contents that was used by another task.

Recent work has focused on these issues and different kinds of approaches have been proposed: some intend to take the possible interferences into account when computing the WCET of a task, others aim at controlling the interactions to make the WCET analysis easier. In the latter category, some solutions require detailed knowledge of all the tasks that may execute concurrently to the task to be analyzed, while other solutions make it possible to determine the WCET without knowing anything about the concurrent tasks. In this paper, we review all these approaches and we discuss their relevance from the point of view of static WCET analysis.

The paper is organized as follows. Section 2 gives a short overview of static WCET analysis techniques with special focus on hardware-specific parts and shows how resource sharing may impact instruction timings. A general overview of the approaches that have been proposed to deal with inter-task interferences is given in Section 3. In Sections 4 and 5, techniques related to handling storage and bandwidth resource sharing respectively are presented. Concluding remarks are given in Section 6.

2 Static WCET analysis and impact of resource sharing

2.1 Static WCET analysis

Techniques for static WCET analysis have been investigated for the last fifteen years. The proposed solutions rely on a number of assumptions: the WCET is computed for a task considered alone, that is not impacted by any other task or external event, that cannot be preempted by the system scheduler (except for specific works on the effects of preemptions, like [3]) and that cannot be interrupted.

Static WCET analysis typically requires three steps. The flow analysis builds the Control Flow Graph of the application from its executable code, and determines flow facts like loop bounds and infeasible paths from the source code [10, 15, 21]. The low-level analysis computes the worst-case execution costs of basic blocks taking into account the specifications of the target hardware and will be detailed below. Finally the WCET computation combines the flow facts and the execution costs to find out the longest path and its execution time: one popular method for this computation is the Implicit Path Enumeration Technique (IPET) [17] based on integer linear programming techniques.

The low-level analysis step breaks down into two sub-steps. The first one examines the behavior of history-based components, mainly the instruction and data caches: the most popular approaches are based on abstract interpretation techniques [6] and assign a category to each access to the cache (`ALWAYS_MISS`, `ALWAYS_HIT`, `PERSISTENT` or `NOT_CLASSIFIED`). Existing solutions consider set-associative instruction and data caches [11], or multi-level cache hierarchies [13]. The second part of low-level analysis computes the execution cost of each basic block when executed in the pipeline [34, 18, 32]. When examining the way a basic block is processed through the pipeline, any possible context (initial pipeline state) must be considered. The existing algorithms differ in how this context is expressed: as a worst-case pipeline state [18], as an abstract state built by abstract interpretation [34] or as a set of parameters that represent the availability of every pipeline resource [32]. But they are in agreement on the fact that they derive block costs from relative (instead of absolute) start and finish times. The impact of the cache latencies (related to the previously determined categories) may be taken into account when estimating the block costs or considered globally in the WCET computation step (which is likely to be less precise, and even unsafe for processors that make timing anomalies [20, 31] possible).

2.2 Impact of resource sharing on instructions timings

Simultaneous multithreading (SMT) processors execute several threads concurrently to improve the usage of hardware resources (mainly functional units) [38]. Common resources (instruction queues, functional units, but also instruction and data caches and branch predictor tables) are shared between concurrent threads. Some of these resources (instruction queues and buffers, caches) are referred to as *storage* resources because they keep information for a while, generally for several cycles. On the contrary, *bandwidth* resources (e.g. functional units or commit stage) are typically reallocated at each cycle [5]. A similar terminology can be used for the shared resources in a multicore architecture: a cache that is shared among the cores is a storage resource while a common bus to the memory hierarchy is a bandwidth resource.

Resource sharing is likely to impact the instructions timing. For a bandwidth resource, possible conflicts between concurrent threads to access the resource may delay some of the threads. As a result, some instruction latencies are lengthened. In an SMT core, delayed instructions may spend more time than expected in some of the pipeline stages. In a multicore, the latency of an access to the main memory may be increased because of the waiting time to the bus.
The effects of sharing storage resources are two-fold. On the one hand, the resource capacity that is usable by a thread may be less than expected since some entries may be occupied by other threads. In an SMT core, this may result in instructions being stalled in a pipeline stage because their destination queue is full. On the other hand, shared memories like caches or branch predictor tables may have their contents corrupted by other threads which could produce either destructive or constructive effects. A destructive effect is observed when another thread degrades the memory contents from the point of view of the thread under analysis: for example, another thread replaces a cache line that had been loaded by the analyzed thread and is still useful. On the contrary, a constructive effect improves the situation for the thread under analysis: for example, a cache line that it requires has been brought into the cache by another thread (this may happen when the threads share parts of code or data). However, even what is seen as constructive in the average case might impair the results of WCET analysis if the processor suffers from timing anomalies [20, 31] (in that case, a miss in the cache does not always lead to the worst-case execution time).

It is absolutely *unsafe* to ignore the effects of resource sharing when computing WCETs. Although we focus on static WCET analysis throughout this paper, we also insist that it is at least equally unsafe to rely on measurement-based timing analysis on a parallel architecture since it is very unlikely that all the possible threads interferences can be observed. In the next section, we review various approaches that have been investigated to cope with these difficulties.

3 General approaches to WCET analysis/analysability of concurrent applications

We have found three kinds of approaches to the problem of accounting for parallel tasks interferences when computing the WCET of one of these tasks. They differ from each other by the way they consider that the impact of concurrent tasks should be taken into account. In the following, τ represents a task under WCET analysis while T stands for the set of its concurrent tasks.

In this section, we give the main principles of these approaches. How they have been instantiated in the literature is described later in the paper.

3.1 Joint WCET analysis of tasks

A first category of approaches to the WCET analysis of a task executed in parallel to other tasks includes the solutions that consider the set of tasks altogether in order to determine their possible interactions. As far as storage resources are concerned, this means analyzing the code of each task in $T \cup \{\tau\}$ to determine possible conflicts, and then accounting for the impact of these conflicts on τ's WCET. For bandwidth resources, identifying conflicts generally requires considering all the possible task interleavings which is likely to be complex with fine-grained interleavings (e.g. at instruction- or memory access-level).

The feasibility of joint analysis techniques relies on all the co-running tasks being known at analysis time. This might be an issue when considering a mixed-criticality workload for which non critical tasks are dynamically scheduled (then any non critical tasks in the system should be considered as a potential opponent). In addition, it may happen that the non critical tasks have not been developed with WCET-analysis in mind and they may not be analyzable, e.g. due to tricky control flow patterns. Also, even with an homogeneously critical workload, the set of tasks that may be co-scheduled with the task under analysis depends on the schedule which, in turn, is determined from the tasks WCETs. This issue might be tackled through an iterative process but we are not aware of any work on this topic.

3.2 Statically-controlled resource sharing

Acknowledging the difficulty of analyzing storage and bandwidth conflicts accurately, a number of solutions have been proposed to statically master the task interferences so that they might be more easily taken into account in the WCET analysis. The techniques in this category all require having knowledge of the complete workload.

Controlling interferences in storage resources generally consists in limiting such interferences by restricting accesses to the shared resource. As we will see in the next sections, the proposed techniques of this kind really tend to meet the requirements of static WCET analysis techniques in terms of reduced complexity, but the solutions basic on static control proposed for bandwidth resources do not fit the principles of static WCET analysis.

3.3 Task isolation techniques

The third category of approaches includes all those that intend to make it possible to analyze the WCET of a task/thread without any knowledge about the concurrent tasks/threads. This is achieved through the design of hardware schemes that exhibit predictable behavior for shared resources. For storage resources, a common approach is to partition the storage among the tasks, so that each critical task has a private partition. For bandwidth resources, an appropriate arbitration is needed, that guarantees upper bound delays independently of the workload.

In the following, we review the techniques that have been proposed so far and that belong to these three categories.

4 Approaches to analyze storage resource sharing

4.1 Joint analysis of memories

Several recent papers focus on the analysis of the possible corruption of L2 shared instruction caches by concurrent tasks [40, 41, 12]. The general process is the following: L1 and L2 instruction cache analysis is first performed for each task in $T \cup \{\tau\}$ independently, ignoring interferences, using usual techniques [11]; then the results of the analysis of the L2 cache for task τ are modified considering that each cache set used by another task in T is likely to be corrupted. For a direct-mapped cache, as studied by Yan and Zhang [40], any access to a conflicting set is classified as `ALWAYS_MISS` (should be `NOT_CLASSIFIED` if timing anomalies may occur). For a set-associative cache, as considered by Li *et al.* [41] and Hardy *et al.* [12], possible conflicts impact the ages of cache lines.

The main concern with this general approach is its scalability to large tasks: if the number of possible concurrent tasks is large and if these tasks span widely over the L2 cache, we expect most of the L2 accesses to be `NOT_CLASSIFIED` which may lead to an overwhelmingly overestimated WCET. For this reason, Li *et al.* [41] refine the technique by introducing an analysis of tasks lifetimes, so that tasks that cannot be executed concurrently (according to the scheduling algorithm, which is non-preemptive and static priority-driven in this paper, and to inter-tasks dependencies) are not considered as possibly conflicting. Their framework involves an iterative worst-case response time analysis process, where each iteration (i) estimates the BCET and WCET of each task according to expected conflicts in the L2 cache; (ii) determines the possible tasks schedules, which may show that some tasks cannot overlap (the initial assumption is that all tasks overlap). This approach is likely to reduce pessimism but may not fit independent tasks with a more complex scheduling scheme. Another solution to the complexity issue has been proposed by Hardy *et al.* [12]: they introduce a compiler-directed scheme that enforces L2 cache bypassing for single-usage program blocks. This sensibly reduces the number of possible conflicts. Lesage *et al.* [16] have recently extended this scheme to shared data caches.

4.2 Storage partitioning and locking schemes

Cache partitioning and locking techniques have first been proposed as a means to simplify the cache behavior analysis in single-core non-preemptive systems [27, 26, 30, 25]. Recently, these techniques have been investigated by Suhendra and Mitra [37] to assess their usability in the context of shared caches in multicore architectures. They consider combinations of (static or dynamic) locking schemes and (core-based or task-based) partitioning techniques. They find out that (i) core-based partitioning strategies (where each core has a private partition and any task can use the entire partition of the core it is running on) outperform task-based algorithms; (ii) dynamic locking techniques, that allow reloading the cache during execution, lead to lower WCETs than static approaches.

Paolieri *et al.* [23] investigate software-controlled hardware cache partitioning schemes. They consider columnization (each core has a private write access to one or several ways in a set-associative cache) and bankization (each core has a private access to one or several cache banks) techniques. In both cases, the number of ways/banks allocated to each core can be changed by software, but it is assumed to be fixed all along the execution of a given task. They show that bankization leads to tighter WCET estimates.

Techniques to achieve timing-predictability in SMT processors are also based on partitioning instructions queues [1, 22].

5 Approaches to analyze bandwidth resources sharing

5.1 Joint analysis of conflict delays

Crowley and Baer have considered the case of a network processor running pipelined packet handling software [7]. The application includes several threads, each one implementing one stage of the computation. The processor features fine-grained multithreading: it provides specific hardware to store the architectural state of several threads, which allows fast context switching, and switches to another thread whenever the current thread is stalled on a long-latency operation. The time during which a thread is suspended depends on the time the other threads can execute before, in turn, yielding control so that the first thread can resume its execution. The proposed approach consists in determining the overall WCET of the application (set of concurrent threads) by considering the threads altogether. The Control Flow Graphs used for static WCET analysis are augmented with *yield nodes* at the points where the threads will yield control. *Yield edges* link each yield node of a given thread to all the return-from-yield nodes of any other thread that is likely to be selected when it is suspended. This results in a complex global Control Flow Graph which, in addition to the control flow of each thread, expresses the possible control flow from one thread to another. From this CFG, an integer linear program is built and used to determine the overall WCET of the application, using the IPET method [17]. Our feeling is that such an approach is not scalable and cannot handle complex applications.

5.2 Statically-scheduled access to shared bandwidth resources

To improve the analysability of latencies to a shared bus in a multicore architecture, Rosén *et al.* [33] introduce a TDMA-based bus arbiter. A *bus schedule* contains a number of slots, each allocated to one core, and is stored in a table in the hardware. At run-time, the arbiter periodically repeats the schedule and grants the bus to the core the current slot has been assigned to. The idea behind this scheme is that a predefined bus schedule makes the latencies of bus accesses predictable for WCET analysis. This relies on the assumption that it is possible, during the low level analysis, to determine the start time of each node (basic block) in the CFG so that it can be decided whether an access to the bus is within a bus slot allocated to the core or is to be delayed. This assumption does not hold for static WCET analysis techniques. It would require unrolling all the possible paths in the CFG which clearly goes against the root principles of static analysis. Moreover, in the case of multiple possible paths (which is the common case), a block is likely to exhibit a large number of possible start times which will noticeably complicate the WCET computation. Alternatively, the delay to get access to the bus could be upper bounded by the sum of the other slots lengths. This would come to the simple round-robin solution discussed below if slots are as short as the bus latency, but would probably severely degrade the worst-case performance with longer slots. For these reasons, we believe that static WCET analysis can get advantage of static bus scheduling only for applications that exhibit a very limited number of execution paths, as targeted by the single-path programming paradigm [28].

5.3 Task-independent bandwidth partitioning schemes

Solutions to make the latencies to shared bandwidth resources predictable reside in bandwidth partitioning techniques. This is what we call *task isolation*: an upper bound of the shared resource latency is known (it does not depend on the nature of the concurrent tasks) and can be considered for WCET analysis.

Mische *et al.* [22] introduce CarCore, a multithreaded embedded processor that supports one hard real-time thread (HRT) together with non critical threads. Temporal thread isolation is ensured for the HRT only, in such a way that its WCET can be computed as if it was executed alone in the processor (i.e. its execution time cannot be impacted by any other thread).

When considering multiple critical threads running simultaneously either in an SMT core or in a multi-core architecture (with one hard real-time thread per core), most of the approaches are based on Round-Robin-like arbitration which allows considering an upper bound on the latency to the shared resource: $D = N \times L - 1$ where L is the latency of the resource and N is the number of competing tasks. Barre *et al.* [1] propose an architecture for an SMT core supporting several critical threads: to provide time-predictability, the storage resources (e.g. instruction queues) are partitioned and the bandwidth resources (e.g. functional units) are scheduled by such a round-robin scheme. Paolieri *et al.* [23] propose a round-robin-like bus arbiter to the shared memory hierarchy in a multi-core architecture. This scheme is completed by a time-predictable memory controller [24] that also guarantees upper bounds on the main memory latencies. Bourgade *et al.* [2] introduce a multiple-bandwidth bus arbiter where each core is assigned a priority-level that defines its upper-bound delay to get access to the bus. This scheme better fits workloads where threads exhibit heterogeneous demands to the main memory.

The MERASA project [39] funded by the European Community (FP7 program) has designed a complete time-predictable multicore architecture with SMT cores, that implements some of the mechanisms mentioned above.

The PRET architecture [19] is built around a thread-interleaved pipeline: it includes private storage resources for six threads and each of the six pipeline stages processes an instruction from a different thread. To prevent long-latency instructions from stalling the pipeline and thus impacting the other threads, these instructions are replayed during the thread's slots until completion. Each thread has private instruction and data scratchpad memories and the off-chip memory is accessed through a *memory wheel* scheme where each thread has its own access window.

6 Conclusion

Parallel architectures are more and more frequently used in embedded system designs. However, they raise timing-analysability issues for critical applications for which worst-case execution time must be computed. Recent research on WCET analysis techniques and real-time systems design address this topic.

We have found three kinds of approaches in the literature. Some of them intend to consider the concurrent tasks altogether to get insight into their possible interferences. Unfortunately, these techniques would probably not be feasible for a real-size system. The second category of approaches includes those that exploit the knowledge of the whole set of concurrent tasks to statically partition accesses to storage and bandwidth resources. This seems to be sound for storage resources, even if it requires a preliminary analysis of conflicts that may be costly in time. But fine-grained static-scheduling schemes for bandwidth resources do not fit static WCET analysis techniques. For these reasons, approaches belonging to the third category, that aim at making the WCET of one task computable independently of the nature of concurrent tasks, seem to be the most relevant today. However, existing schemes probably do not scale well and will have to be improved to allow wider parallelism.

Research on WCET analysis and WCET-aware design of parallel architectures is still in early stages. We expect these topics to receive more and more attention in the next years. We believe that future critical system designs will favor task isolation at various levels to keep the problem of determining the WCETs of tasks tractable even on large-scale architectures. Task isolation may be enforced using hardware arbitration schemes in a hierarchical architecture where each resource is shared by only a limited number of nodes. In addition, the software should be designed in such a way that conflicts can only occur in well-delimited parts of the task codes. Such a behavior can be achieved considering appropriate resource access models, where a task can access a shared resource only in dedicated phases, as proposed in [36]. Provided the hardware and software conjunctly limit the conflicts between tasks, the techniques that have been proposed to analyse the WCETs considering the possible task interaction may be usable and useful to take into account the remaining possible conflicts.

References

1 J. Barre, C. Rochange, P. Sainrat. *An Architecture for the Simultaneous Execution of Hard Real-Time Threads.* Int'l Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation (IC-SAMOS), 2008.

2 R. Bourgade, C. Rochange, M. de Michiel, P. Sainrat. *MBBA: a Multi-Bandwidth Bus Arbiter for hard real-time.* 5th Int'l Conf. on Embedded and Multimedia Computing (EMC), 2010.

3 C. Burguiere, J. Reineke, S. Altmeyer. *Cache Related Preemption Delay for Set-Associative Caches.* 9th Int'l Workshop on WCET Analysis, 2009.

4 J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, S. Baruah. *A Categorization of Real-time Multiprocessor Scheduling Problems and Algorithms.* In Handbook of Scheduling: Algorithms, Models, and Performance Analysis, Joseph Y-T Leung (ed). Chapman Hall/ CRC Press, 2004.

5 F. Cazorla, A. Ramirez, M. Valero, P. Knijnenburg, R. Sakellariou, E. Fernandez. *QoS for High-Performance SMT Processors in Embedded Systems.* IEEE Micro, 24(4), 2004.

6 P. Cousot, R. Cousot. *Static determination of dynamic properties of programs.* 2nd International Symposium on Programming, 1976.

7 P. Crowley, J.-L. Baer. *Worst-Case Execution Time estimation for Hardware-assisted Multithreaded Processors.* 2nd Workshop on Network Processors, 2003.

8 R.I. Davis, A. Burns. *A Survey of Hard Real-Time Scheduling for Multiprocessor Systems.* Accepted for publication in ACM Computing Surveys.

9 S.A. Edwards, S. Kim, E.A. Lee, H.D. Patel, M. Schoeberl. *Reconciling Repeatable Timing with Pipelining and Memory Hierarchy?* Workshop on Reconciling Performance with Predictability (RePP), 2009.

10 A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, B. Lisper. *Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis.* 7th Int'l Workshop on WCET Analysis, 2007.

11 C. Ferdinand, R. Wilhelm. *Fast and efficient cache behavior prediction for real-time systems.* Journal on Real-Time Systems, 17(2/3), Springer, 1999.

12 D. Hardy, T. Piquet, I. Puaut. *Using Bypass to Tighten WCET Estimates for Multi-Core Processors with Shared Instruction Caches.* IEEE Real-Time Systems Symp. (RTSS), 2009.

13 D. Hardy, I. Puaut. *WCET Analysis of Multi-level Non-inclusive Set-Associative Instruction Caches*. IEEE Real-Time Systems Symposium (RTSS), 2008.

14 A. Hansson, K. Goossens, M. Bekooij, J. Huisken. *Compsoc: A template for composable and predictable multi-processor system on chips*. ACM Transactions on Design Autom. Electron. Syst., 14(1), 2009.

15 N. Holsti. *Analysing Switch-Case Tables by Partial Evaluation*. 7th Int'l Workshop on WCET Analysis, 2007

16 B. Lesage, T. Hardy, I. Puaut. *Shared Data Cache Conflicts Reduction for WCET Computation in Multi-Core Architectures*. Int'l Conf. on real-Time Networks and Systems, 2010.

17 Y.-T. S. Li, S. Malik. *Performance Analysis of Embedded Software using Implicit Path Enumeration*. Workshop on Languages, Compilers, and Tools for Real-time Systems, 1995.

18 X. Li, A. Roychoudhury, T. Mitra. *Modeling out-of-order processors for WCET analysis*. Journal of Real-Time Systems, 34(3), Springer, 2006.

19 B. Lickly, I. Liu, S. Kim, H.D. Patel, S.A. Edwards, E.A. Lee. *Predictable programming on a precision timed architecture*. Int'l Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES), 2008.

20 T. Lundqvist, P. Stenström. *Timing Anomalies in Dynamically Scheduled Microprocessors*. IEEE Real-Time Systems Symposium (RTSS), 1999.

21 M. de Michiel, A. Bonenfant, H. Cassé, P. Sainrat. *Static loop bound analysis of C programs based on flow analysis and abstract interpretation*. IEEE Int'l Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2008.

22 J. Mische, I. Guliashvili, S. Uhrig, T. Ungerer. *How to Enhance a Superscalar Processor to Provide Hard Real-Time Capable In-Order SMT*. 23rd Int'l Conf. on Architecture of Computing Systems (ARCS), 2010.

23 M. Paolieri, E. Quinones, F. Cazorla, G. Bernat, M. Valero. *Hardware Support for WCET Analysis of Hard Real-Time Multicore Systems*.36th Int'l Symp. on Computer Architecture (ISCA), 2009.

24 M. Paolieri, E. Quinones, F. Cazorla, M. Valero. *An Analyzable Memory Controller for Hard Real-Time CMPs*. IEEE Embedded Systems Letters, 1(4), 2009.

25 S. Plazar, P. Lokuciejewski, P. Marwedel. *WCET-aware Software based Cache Partitioning for Multi-task Real-time Systems*. 9th Int'l Workshop on WCET Analysis, 2009.

26 I. Puaut. *WCET-centric software-controlled instruction caches for hard real-time systems*. 6th Int'l Workshop on WCET Analysis, 2006.

27 I. Puaut, D. Decotigny. *Low-complexity Algorithms for Statc Cache Locking in Multitasking Hard Real-Time Systems*. IEEE Real-Time Systems Symposium (RTSS), 2002.

28 P. Puschner, A. Burns. *Writing temporally predictable code*. 7th IEEE Int'l Workshop on Object-Oriented Real-Time Dependable Systems, 2002.

29 P. Puschner, M. Schoeberl. *On Composable System Timing, Task Timing, and WCET Analysis*. 8th Int'l Workshop on WCET Analysis, 2008.

30 R. Reddy, P. Petrov. *Eliminating inter-process cache interference through cache reconfigurability for real-time and low-power embedded multi-tasking systems*. Int'l Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES), 2007.

31 J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, B. Becker. *Definition and Classification of Timing Anomalies*. 6th Int'l Workshop on WCET Analysis, 2006.

32 C. Rochange, P. Sainrat. *A Context-Parameterized Model for Static Analysis of Execution Times*. Transactions on High-Performance Embedded Architectures and Compilers, 2(3), Springer, 2007.

33 J. Rosén, A. Andrei, P. Eles, Z. Peng. *Bus Access Optimization for Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip*. 28th IEEE Int'l Real-Time Systems Symposium, 2007.

34 J. Schneider, C. Ferdinand. *Pipeline behavior prediction for superscalar processors by abstract interpretation.* SIGPLAN Notices, 34(7), ACM, 1999.

35 M. Schoeberl, P. Puschner. *Is Chip-Multiprocessing the End of Real-Time Scheduling?.* 9th Int'l Workshop on WCET Analysis, 2009.

36 A. Schranzhofer, R. Pellizzoni, Jian-Jia Chen, L. Thiele, M. Caccamo. *Worst-case response time analysis of resource access models in multi-core systems.* Design Automation Conference (DAC), 2010.

37 V. Suhendra, T. Mitra. *Exploring Locking and Partitioning for Predictable Shared Caches on Multi-cores.* 45th Conf. on Design Automation (DAC), 2008.

38 D. Tullsen, S. Eggers, H. Levy. *Simultaneous Multithreading: Maximizing On-Chip Parallelism.* 22nd Int'l Symposium on Computer Architecture (ISCA), 1995.

39 T. Ungerer, F. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, *et al.*. *MERASA: Multi-Core Execution of Hard Real-Time Applications Supporting Analysability.* IEEE Micro, 30(5), 2010.

40 J. Yan, W. Zhang. *WCET Analysis for Multi-Core Processors with Shared L2 Instruction Caches.* IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS), 2008.

41 Yan Li, V. Suhendra, Yun Liang, T. Mitra, A. Roychoudhury. *Timing Analysis of Concurrent Programs Running on Shared Cache Multi-Cores.* IEEE Real-Time Systems Symp. (RTSS), 2009.