Automatic WCET Analysis of Real-Time Parallel Applications

Haluk Ozaktas, Christine Rochange, Pascal Sainrat

Automatic WCET Analysis of Real-Time Parallel Applications*

Haluk Ozaktas, Christine Rochange, and Pascal Sainrat

IRIT – Université de Toulouse
France
firstname.lastname@irit.fr

Abstract

Tomorrow's real-time embedded systems will be built upon multicore architectures. This raises two challenges. First, shared resources should be arbitrated in such a way that the WCET of independent threads running concurrently can be computed: in this paper, we assume that time-predictable multicore architectures are available. The second challenge is to develop software that achieves a high level of performance without impairing timing predictability. We investigate parallel software based on the POSIX threads standard and we show how the WCET of a parallel program can be analysed. We report experimental results obtained for typical parallel programs with an extended version of the OTAWA toolset.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases WCET analysis, parallel programming, thread synchronisation

Digital Object Identifier 10.4230/OASIcs.WCET.2013.11

1 Introduction

Future real-time embedded systems will have to follow the global trend towards multicore computing units, which is mainly guided by power efficiency considerations. Designing time-predictable multicore architectures is at the heart of several research projects, e.g. T-CREST[1] and parMERASA[2]. Now, when hardware solutions are available, software will have to be carefully designed to optimise the usage of resources. In some cases, the target is a high task throughput: it can be achieved by co-scheduling independent tasks on the cores. Other applications, e.g. command-control functions in cyber-physical systems, instead require shortened response times. For some of them, that exhibit intrinsic data or control parallelism, the execution time of individual tasks can be reduced by applying parallel programming techniques: a task is decomposed into threads that are run in parallel, each of them processing one part of the workload. In this paper, we focus on this class of programs.

Several parallel programming paradigms can be considered depending on the problem decomposition (task- or data-parallelism) and on the way threads can communicate, which is highly related to the target hardware architecture. Various programming languages and APIs can be used to develop parallel programs. We focus on the POSIX threads standard which is widely used in the industry.

In real-time systems, special attention must be paid to task scheduling: it must be guaranteed that critical tasks will meet their hard deadlines in any situation. Real-time

* The research leading to these results has received funding from the European Union Seventh Framework Programme under grant agreement no. 287519 (parMERASA).

[1] www.t-crest.org

[2] www.parmerasa.eu

task scheduling is a hot research topic. All the proposed strategies rely on estimations of the worst-case execution times (WCET) of critical tasks. Several approaches have been proposed in the past to determine WCET upper-bounds considering sequential tasks running on uni-processors. If these techniques are still to be used when independent tasks run on time-predictable multicores, specific solutions have to be developed for parallel applications, composed of synchronising threads. This is the purpose of this paper.

The paper is organised as follows. In Section 2, we discuss the scope of our work and we give an overview of related work. Section 3 introduces our approach to the automatic WCET analysis of POSIX-based parallel tasks. Experimental results are reported in Section 4 and Section 5 concludes the paper.

2 Scope of the paper and related work

2.1 Time-predictable multicores

Various classes of parallel architectures exist, from chip multicores to clusters and grids of computers, or from general-purpose processing units to specialised accelerators like GPUs. However, to be considered as candidates to build hard real-time systems, these architectures should enforce timing analysability: it should be possible to compute the WCET of a critical task running in parallel with other tasks. The main difficulty comes from inter-task conflicts: they make the latencies of accesses to shared resources hard to predict. This mainly concerns shared caches, memory controllers and interconnection networks.

Two kinds of approaches have been proposed. A first group of solutions consist in considering all together the tasks that might be running at the same time in order to estimate their possible interactions and their impact on the worst-case execution times. This strategy has been considered for the analysis of shared caches [9, 11, 14] and shared busses [2, 20]. A second class of approaches aim at designing hardware that enforces spatial and timing isolation for critical tasks. Cache locking and partitioning schemes [21, 16] belong to this category. Timing isolation can also be supported by appropriate arbitration mechanisms, e.g. for a shared bus [19, 10] or a memory controller [1, 15]. More globally, several European projects have been launched to design time-predictable multicores, e.g. MERASA [22], PREDATOR [4], T-CREST and parMERASA.

In the following, we assume we have a time-predictable shared-memory multicore architecture and the related WCET analysis tool capable of analysing sequential applications: modelling hardware-level thread interactions is out of the scope of our work.

2.2 Real-time and WCET-aware parallel applications

Various parallel programming models exist and are supported by a large number of programming languages and APIs. In this work, we focus on applications developed on the widely-used POSIX threads standard.

Programs written with POSIX threads are characterised by explicit thread control (creation and join) and explicit thread synchronisation through mutexes, condition variables and barriers. Figure 1 shows a sample program that we will use as a running example in the paper. Determining the WCET of such a parallel program comes up to computing the WCET of the main thread, taking into account the costs for thread control and thread synchronisations. The predictability of these costs highly depend on the implementation of the system software. This is discussed in Section 2.3.

```
int main() {
  for (int i=0; i<2; i++)
    CREATE_THREAD(&work);
  ...
  BARRIER(&bar,3);  // ID=bar
  ...
  for (int i=0; i<2; i++)
    JOIN(i+1);  // ID=join
}
```

```
void work() {
  ...
  BARRIER(&bar,3);  // ID=bar
  ...
  MUTEX_LOCK(&lock);  // ID=cs
  ... // critical section
  MUTEX_UNLOCK(&lock);  // ID=cs
  ...
}
```

Figure 1 Example code.

Real-time parallel applications should be designed with time predictability in mind. As we will see later, stall times at synchronisations impact the WCET. Then the main recommendations come from the way these stall times can be estimated: synchronisation operations as well as the involved threads should be easy to identify. As a consequence, the number of threads should be statically fixed and synchronisation patterns should make it possible to determine how long one thread may be stalled by another one. The latter can be achieved by using standard synchronisation patterns, like critical sections and barriers.

In the following we also consider that the number of threads is lower than or equal to the number of cores so that all the threads can execute in parallel, each on a different core. In practice, it could be accepted that the number of threads exceed the number of cores. In such a case, however, the scheduling of threads and their mapping to cores must be decided statically [17]. This way, the timing analysis can determine how to compose their individual WCETs. This option is not considered in the paper.

2.3 Time-predictable system software

The analysis of stall times requires the synchronisation to be implemented with time-predictable primitives. Mainly, these primitives should allow upper bounding the stall time of a thread at a synchronisation. Ticket locks should, for example, ensure that threads reaching a critical section will be granted access in a First-Come First-Served fashion. The design of such predictable primitives is discussed in [23, 5]. We assume that the applications are developed using such routines. In addition, timing analysis either needs an upper-bound in the latency of a thread creation or a hardware mechanism that enforces a synchronous start of created child threads.

2.4 Related work

As mentioned earlier, using multicores to build hard real-time systems is not common yet. Research on WCET analysis on multicores has essentially focused on the predictability of accesses to shared resources, as overviewed in Section 2.1. There have been very few contributions to the analysis of parallel programs. The timing analysis of a parallelised control-loop style application was reported in [6]. In [18], a first attempt to *manually* compute the WCET of an industrial parallel program with static analysis techniques was reported. Individual code segments were analysed using the OTAWA toolset, then their WCETs were combined outside the toolset by hand to determine the WCET of the whole application.

In [7], the authors propose a method based on timed automata to model the behaviour of a parallel program. Model checking techniques are used to determine the WCET of the whole program by verification. In [8], they consider a simplified parallel programming language

and introduce an approach based on abstract interpretation to perform simultaneous timing analysis of the different threads. The predictability of various parallel programming models, e.g. GPU and data parallel programming, is investigated in [13].

3 Approach to the WCET analysis of parallel applications

The execution time of a parallel program is the execution time of its longest thread. In our model, the main thread creates child threads and later joins them. Then determining the WCET of a parallel program comes to computing the WCET of its main thread. This time is impacted by the child threads:

- The latency of the thread creation operation must be accounted for;
- The main thread may have to wait for other threads when it reaches a barrier or the lock acquisition operation before a critical section. The worst-case stall time must be estimated.
- When joining the child threads, the main thread has to wait for their termination

3.1 Timing analysis of synchronisations

We distinguish two kinds of synchronisations: critical sections, guarded by locks, and progress synchronisations, implemented by barriers or conditions (wait and signal). In both cases, a thread that reaches a synchronisation primitive may be forced to wait before proceeding. Its worst-case stall time (WCST) must be estimated.

3.1.1 Worst-case stall times

Critical section

Entering a critical section is typically achieved by acquiring a lock. If no other thread requests the lock at the same time, then the synchronisation does not generate any stall. But in the worst case, all possible contenders try to acquire the lock simultaneously, and the thread has to wait for all other threads to release the lock (provided locks are granted in a First-Come First-Served fashion). This is illustrated in the left-side part of Figure 2.

The WCST at the critical section for the leftmost thread, denoted by S, is computed assuming the two other threads already have requested the lock. Then S is the sum of the times during which each of them holds the lock, i.e. the WCETs of their critical sections:

$$S = w_1 + w_2$$

Progress synchronisation (barrier)

The right-side part of Figure 2 illustrates the stall time of a thread at a barrier. The WCST is determined by considering the previous collective synchronisation point, i.e. the previous point where all the involved threads did synchronise before the barrier.

The (actual) stall time of one thread (thread i) at a barrier to be reached by a single other thread (thread j) would be given by $\max(0, t_j - t_i)$, where t_i and t_j are the actual execution times of threads i and j to reach the barrier from the previous synchronisation point: either $t_i \geq t_j$ and thread i does not have to wait, or the stall time is the difference between their execution times.

Now, threads generally exhibit variable execution times: $t_i \in [b_i, w_i]$ where b_i and w_i are the best- and worst-case execution times for thread i (similarly, $t_j \in [b_j, w_j]$). Then attention should be paid to how the difference between their execution times is computed.

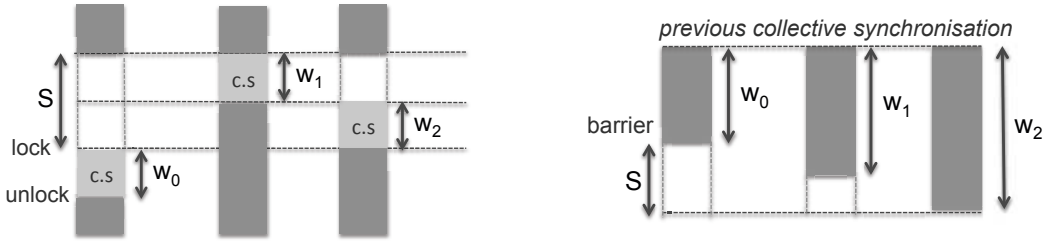

Figure 2 Stalls due to synchronisations.

Theoretically, the longest stall time by thread i when $t_i < t_j$ is given by $w_j - b_i$ (difference between the worst-case execution time of thread j and the best-case execution time of thread i). However, computing the WCST for thread i is done in the context of determining its WCET. As a result, the worst-case value for $t_j - t_i$ is computed as $w_j - w_i$.

Generalising to several threads, as in the example shown in Figure 2 (right side), we get:

$$S = \max(0, (w_1 - w_0), (w_2 - w_0))$$

3.1.2 Abstract view of synchronisation primitives

While a synchronisation operation is simply seen as a call to a system-software primitive, things are a bit more complex from the point of view of WCET analysis which is done at cycle-/instruction-level. The main issue is to identify key locations in the code of the primitives: the point where a thread may be stalled and the point where a thread may signal other threads (allowing them to resume their execution). Finding out these locations is a hard task and having it done automatically is still challenging. This is the reason why we need the parallel application to use known primitives, that have been previously analysed manually (as described in [18]). We plan to release this constraint by designing a specific format to describe synchronisation routines, so that the user could use his own primitives and provide a description for them.

3.1.3 Computation of the global WCET

The WCET of the whole application is computed as the WCET of the main thread to which WCSTs at synchronisations are added. In our example (see Figure 3), two WCSTs must be estimated for the main thread. The first one, S_1, is related to a barrier. As explained in Section 3.1.1, it is determined considering the threads' WCETs from the previous collective synchronisation, which is the creation of the child threads, to the barrier (we assume that the cost of thread creation is known). The second stall time, S_2, at the join with child threads, can be analysed similarly. The previous collective synchronisation is the barrier. However, the code executed by the child threads from the barrier to the exit includes a critical section. Then S_2 depends on S_3, which can be determined as shown earlier.

Figure 5 depicts the global procedure to perform the timing analysis of a parallel program. First, synchronisation patterns must be identified. This may be a complex task. To make it simpler, we rely on user-provided annotations that we will describe in Section 3.2. The second step determines the dependencies among the stall times and builds a WCST tree, as the one shown in Figure 4. This is done from the root down to the leaves: a branch ends when a WCST can be computed from partial execution paths that do not include any synchronisation. WCSTs can then be estimated by climbing up the tree from the leaves to the

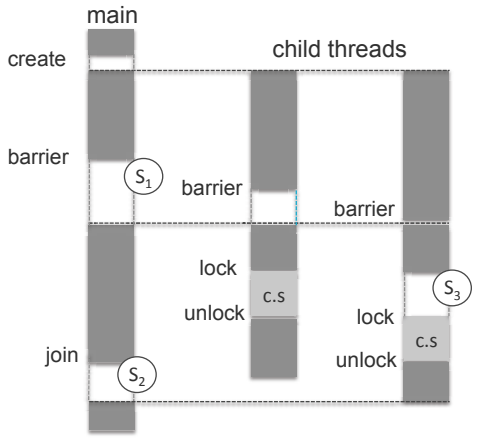

Figure 3 Example program.

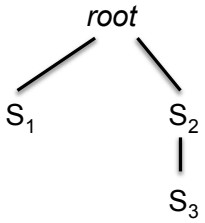

Figure 4 WCST tree.

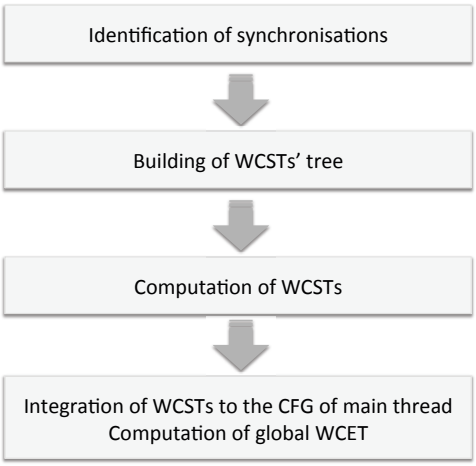

Figure 5 Global procedure.

```
<barrier id="bar">
  <thread id="0-2">
    <last_sync ref="BEGIN"/>
  </thread>
</barrier>
<csection id="cs">
  <thread id="1-2"/>
</csection>
<sync id="join">
  <thread id="0">
    <wait id="1-2">
      <sync ref="END"/>
      <last_sync ref="bar"/>
    </wait>
  </thread>
</sync>
```

Figure 6 Annotations for the example code.

root. They are added to the WCETs of the corresponding basic blocks in the program CFG. The final stage integrates the WCSTs into the ILP formulation of the WCET computation (IPET method [12]).

3.2 Annotations of parallel programs

To help the analysis of parallel programs, and in particular of their synchronisations, we have designed an annotation format. It can be used to provide information on synchronisation patterns. The annotation format includes two parts:

- A set of identifiers annotated in the source code, to allow further reference to specific points in the program, i.e. calls to synchronisation primitives. Identifiers are specified as C comments (`// ID=...`), as can be seen in the example code (see Figure 1).
- Additional information, e.g. the threads involved in a synchronisation, are provided in a separate XML-based file. Some elements of this file are described below[3].

[3] Due to space limitations, only a subset of our annotation language is described in this paper. The full language supports more complex synchronisation patterns.

Figure 6 shows the annotation file that describes our example code. It specifies three synchronisations that are likely to generate stall times: a barrier, a lock-based protection for a critical section, and joining the child threads for the main thread.

The `barrier` element refers to a barrier identifier put in the source code. The inner `thread` element indicates which threads should meet at the barrier (`0` is the main thread). For these threads, the previous collective synchronisation is specified in the `last_sync` element, with the `BEGIN` built-in value that refers to the start of each thread. The `csection` element provides details on the synchronisation at the entry of the critical section. The inner `thread` element shows that the two child threads may compete for the lock. Finally, the `sync` element refers to the join operation executed by the main thread as specified by the nested `thread` element. The innermost `wait` element indicates that it should wait for threads 1 and 2 to reach the end of their execution (as specified with the built-in value `END`).

4 Experiments

4.1 Methodology

Our solution to automatically analyse the WCET of parallel programs helped with user-provided annotations has been implemented on top of the OTAWA toolset [3]. OTAWA provides an API to build WCET computation tools based on static analysis techniques. We have extended the library with utilities to parse annotation files, to retrieve synchronisations in the binary code, to build the WCST tree, to analyse the WCSTs and then to integrate them in the linear program used to determine the global WCET.

Since we focus on software interactions, we have considered a simple architecture in which each instruction executes in a single cycle with a configurable additional latency for memory accesses. We have found that the results presented below do not depend on the value of the latency (raw values do, but not the shape of curves).

4.2 Benchmarks

We have analysed two different parallel implementations of a kernel solving a partial differential equation on a 2D-grid. The first version uses the iterative Gauss-Seidel method where each point is computed based on its immediate north and west neighbours. There is no dependency between the points belonging to the same anti-diagonal: they can be computed in parallel. However, dependencies among anti-diagonals should be respected. The algorithm iterates until convergence. Our parallel implementation first divides the grid into compartments such that the main anti-diagonal has the same number of compartments as the number of threads. It exploits the independence of the compartments within a same anti-diagonal. This implementation includes three barriers and one critical section. The main thread participates in the computation and execute the same function as the child threads.

The second version implements the Jacobi method where each point can be computed independently of other points: this improves the intrinsic parallelism but generally requires a larger number of iterations to converge. Our parallel implementation of this algorithm assigns a block of lines to each thread. Threads execute in parallel within an iteration. The code contains two barriers and one critical section. For both methods, we have defined a maximum number of iterations in order to be able to compute a WCET value.

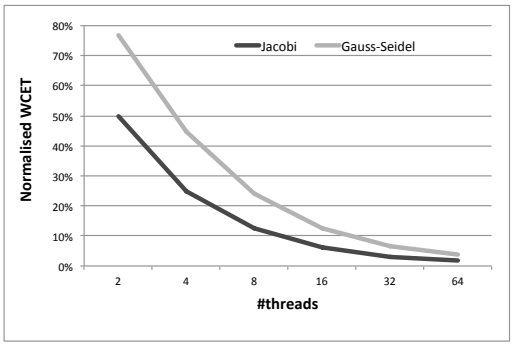

Figure 7 Normalised WCET.

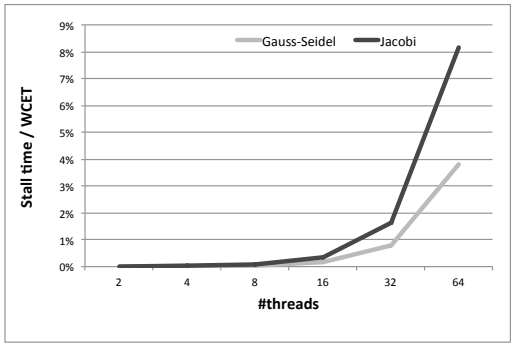

Figure 8 Impact of stall times.

	Gauss-Seidel	Jacobi
2	0.559	0.379
4	1.046	0.705
8	2.177	1.446
16	3.718	2.679
32	8.796	5.782
64	17.999	11.855

Figure 9 Computation times in seconds.

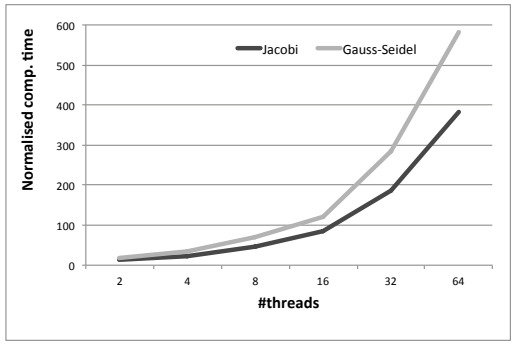

Figure 10 Normalised computation times.

4.3 Results

We report experiments carried out for the two algorithms described above, considering both sequential and parallel (from 2 up to 64 threads) versions.

Figure 7 shows the WCETs of parallel implementations normalised to the WCET of the sequential code. For the same number of threads, the Jacobi algorithm gets higher speed-ups than the Gauss-Seidel method: this was expected since there is no dependence between points in Jacobi method which yields to higher parallelism.

Figure 8 plots the contribution of stall times to the WCET of the application. For up to 32 threads, the impact of worst-case stall times is negligible. For 64 threads, stall times contribute from 4% (Gauss-Seidel) to 8% (Jacobi) of the WCET. These low contributions are mainly due to the fact that all the threads run the same code. Then their worst-case arrival times at barriers are equal. Thus the stall times are only due to the critical section. They rapidly increase with the number of threads because, in the worst case, a thread is stalled until all the possible contenders execute the critical section and release the lock. This shows the importance of limiting the number of contending threads to optimise the WCET.

Figure 9 provides the raw values of the computation time (in seconds) of the automatic WCET analysis of the parallel codes. In Figure 10 these times are normalised to the WCET computation time of the sequential version (which is 0.031 seconds). Analysing a parallel application is noticeably longer than analysing its sequential version. This was somewhat expected since the WCET estimation of a parallel program requires many small WCET analyses on partial paths. Now, in these experiments, the WCET was analysed as if the threads did execute different functions, to reflect a pessimistic situation. In our

two benchmark codes, all the threads instead share the same function. As a result, the real computation cost would be that of the parallel program with two threads (the main and one child), i.e. about 18 times the computation cost for the sequential version for the Gauss-Seidel algorithm (about 12 times for the Jacobi method).

5 Conclusion

With the emergence of multicore architectures in the embedded systems market, one strategy to get high computing power will be to parallelise software. Now, for hard real-time systems, timing predictability is a key issue. It requires specific solutions at the hardware level, since interactions among concurrent threads must be controlled in some way to make their timing analysis possible. This point is at the core of several terminated and ongoing research projects and was considered as solved in this paper. Parallel programming introduces software-level interactions between threads through synchronisation operations. These synchronisations engender stall times that must be accounted for when analysing the worst-case execution times of tasks. This is the problem we have tackled in this work.

We have introduced an approach for an automatic timing analysis of parallel applications. It consists in estimating the synchronisation-related stall times of each individual thread and in considering them as extra-costs for the associated basic blocks in the CFG. This way, the stall times are accounted for within the WCET computation process.

Determining the worst-case stall times due to synchronisations requires a detailed analysis of the synchronisation patterns and of the binary code of synchronisation primitives. To perform this task we rely on annotations that must be generated by the user. Once synchronisation operations are identified, WCSTs are recursively computed.

We have implemented our algorithm on top of the OTAWA library and experimented it on parallelised versions of the Gauss-Seidel and Jacobi algorithms. Experimental results show that the worst-case impact of synchronisation stalls on WCET estimates remains limited (8% for 64 threads). The cost of analysing a parallel code remains reasonable when all the threads execute the same function (around 12 to 18 times the computation cost of the sequential version) but rapidly increases with the number of threads when they run different codes.

As future work, we plan to apply our approach to larger applications and to analyse the impact of parallel programming patterns to the worst-case performance of programs. We will also investigate automatic extraction of synchronisation patterns from the binary code.

References

1 B. Akesson, K. Goossens, and M. Ringhofer. Predator: a predictable SDRAM memory controller. In *5th Int'l Conf. on Hardware/Software Codesign and System Synthesis*, 2007.

2 B. Andersson, A. Easwaran, and J. Lee. Finding an upper bound on the increase in execution time due to contention on the memory bus in COTS-based multicore systems. In *WiP of Real-Time Systems Symposium (RTSS)*, 2009.

3 C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. OTAWA: an open toolbox for adaptive WCET analysis. In *Workshop on Software technologies for Embedded and Ubiquitous Systems (SEUS)*, 2011.

4 C. Cullmann, C. Ferdinand, G. Gebhard, D. Grund, C. Maiza, J. Reineke, B. Triquet, and R. Wilhelm. Predictability considerations in the design of multi-core embedded systems. In *Int'l Conf. on Embedded Real Time Software and Systems*, 2010.

5 M. Gerdes, F. Kluge, T. Ungerer, and C. Rochange. The split-phase synchronisation technique: Reducing the pessimism in the WCET analysis of parallelised hard real-time

programs. In *Int'l Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2012.

6 M. Gerdes, J. Wolf, I. Guliashvili, T. Ungerer, M. Houston, G. Bernat, S. Schnitzler, and H. Regler. Large drilling machine control code—parallelisation and WCET speedup. In *Int'l Symp. on Industrial Embedded Systems (SIES)*, 2011.

7 A. Gustavsson, A. Ermedahl, B. Lisper, and P. Pettersson. Towards WCET analysis of multicore architectures using uppaal. In *Workshop on WCET Analysis*, 2010.

8 A. Gustavsson, J. Gustafsson, and B. Lisper. Toward static timing analysis of parallel software. In *Workshop on WCET Analysis*, 2012.

9 D. Hardy, T. Piquet, and I. Puaut. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *Real-Time Systems Symposium (RTSS)*, 2009.

10 T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury. Bus-aware multicore WCET analysis through TDMA offset bounds. In *Euromicro Conf. on Real-Time Systems (ECRTS)*, 2011.

11 Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury. Timing analysis of concurrent programs running on shared cache multi-cores. In *Real-Time Systems Symposium (RTSS)*, 2009.

12 Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *ACM SIGPLAN Notices*, volume 30, 1995.

13 B. Lisper. Towards parallel programming models for predictability. In *Workshop on WCET Analysis*, 2012.

14 M. Lv, W. Yi, N. Guan, and G. Yu. Combining abstract interpretation with model checking for timing analysis of multicore software. In *Real-Time Systems Symposium (RTSS)*, 2010.

15 M. Paolieri, E. Quiñones, F. J Cazorla, and M. Valero. An analyzable memory controller for hard real-time CMPs. *Embedded Systems Letters, IEEE*, 1(4), 2009.

16 P. Paolieri, E. Quiñones, F. Cazorla, G. Bernat, and M. Valero. Hardware support for WCET analysis of hard real-time multicore systems. In *Int'l Symp. on Computer Architecture (ISCA)*, 2009.

17 M. Pelcat, P. Menuet, S. Aridhi, and J.-F. Nezan. Scalable compile-time scheduler for multi-core architectures. In *Design, Automation and Test in Europe (DATE)*, 2009.

18 C. Rochange, A. Bonenfant, P. Sainrat, M. Gerdes, J. Wolf, T. Ungerer, Z. Petrov, and F.šek Mikulu. WCET analysis of a parallel 3D multigrid solver executed on the MERASA multi-core. In *Workshop on WCET Analysis*, 2010.

19 J. Rosen, A. Andrei, P. Eles, and Z. Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *Real-Time Systems Symposium (RTSS)*, 2007.

20 S. Schliecker, M. Negrean, and R. Ernst. Bounding the shared resource load for the performance analysis of multiprocessor systems. In *Design, Automation and Test in Europe (DATE)*, 2010.

21 V. Suhendra and T. Mitra. Exploring locking & partitioning for predictable shared caches on multi-cores. In *Design Automation Conference (DAC)*, 2008.

22 T. Ungerer, F. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, C. Rochange, E. Quiñones, M. Gerdes, M. Paolieri, J. Wolf, H. Cassé, S. Uhrig, I. Guliashvili, M. Houston, F. Kluge, S. Metzlaff, and J. Mische. MERASA: Multicore execution of hard real-time applications supporting analyzability. *IEEE Micro*, 30(5), 2010.

23 J. Wolf, M. Gerdes, F. Kluge, S. Uhrig, J. Mische, S. Metzlaff, C. Rochange, H. Cassé, P. Sainrat, and T. Ungerer. RTOS support for parallel execution of hard real-time applications on the MERASA multi-core processor. In *Int'l Conf. on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, 2010.