Accurate and Efficient Identification of Worst-Case Execution Time for Multicore Processors: A Survey

Hamid Mushtaq, Zaid Al-Ars, Koen Bertels
Computer Engineering Laboratory
Delft University of Technology
Delft, the Netherlands
{H.Mushtaq, Z.Al-Ars, K.L.M.Bertels}@tudelft.nl

***Abstract*—Parallel systems were for a long time confined to high-performance computing. However, with the increasing popularity of multicore processors, parallelization has also become important for other computing domains, such as desktops and embedded systems. Mission-critical embedded software, like that used in avionics and automotive industry, also needs to guarantee real time behavior. For that purpose, tools are needed to calculate the worst-case execution time (WCET) of tasks running on a processor, so that the real time system can make sure that real time guarantees are met. However, due to the shared resources present in a multicore system, this task is made much more difficult as compared to finding WCET for a single core processor. In this paper, we will discuss how recent research has tried to solve this problem and what the open research problems are.**

I. Introduction

For a long time, single core processors ruled the desktop and embedded market. The popularity of the single core processors could be attributed to the portability they provided. A program written for one processor, could be ported to the faster version of the same processor without changing a single line of code. However, at one point, it was no more possible to build faster single processors due to the huge amount of power they would need. That is the point, where multicore processors came into existence, as they are more power efficient. Nowadays, multicore processors are common in desktops, laptops and mobile phones. However, industries which use mission critical embedded software, such as avionics and the automotive industry have been reluctant to employ multicore systems. The reason being that such software also needs to meet timing deadlines for real time performance. For guaranteeing real time performance, the real time scheduler needs to know the worse-case execution time (WCET) of each task. Finding a good estimate (less pessimistic) of WCET, of a task is much simpler if it runs on a single core processor than if it runs on a multicore processor concurrently with other tasks. This is because those tasks can share resources, such as shared cache or shared bus, and/or may need to concurrently read and/or write shared data.

Recently, there has been an increasing interest to solve the problem of finding WCET for tasks running on multicore processors, from hardware solutions to software solutions for Commodity Off The Shelf (COTS) processors. In this paper, we discuss the research done in this context and also point out the open issues. In Section II, we provide the necessary background to help reader understand the problem of WCET. This is followed by Section III which discusses the WCET calculation techniques employed for single core processors. Next, we discuss the research that has been done for calculating WCET of multicore processors in Section IV. This is followed by Section V on open issues. We finally conclude the paper with Section VI.

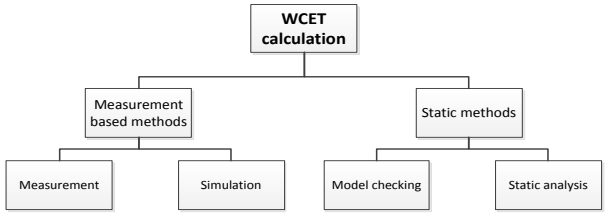

Figure 1. Methods used for WCET calculation

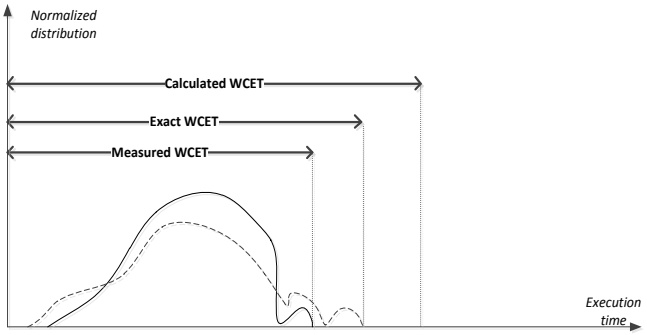

Figure 2. Measurement based vs static methods

II. Background

Multicore processors can be useful in embedded systems, such as automotive systems, as that would mean that software could be made more centralized. This translates to less cable usage in cars, and therefore less fuel consumption, as more cable length is directly proportional to fuel consumption in cars. Moreover, with processors with more cores, more functionality could be added, for example, we could have an improved braking system, which uses more sensors [27].

Mission critical embedded systems perform hard real time tasks, which need to complete within a certain time period. To be able to guarantee that those tasks finish within that time period, their WCET should be known. For single core processors, techniques to find WCET are well known and there are several tools available to perform that. Those techniques and tools can be found in the literature [30].

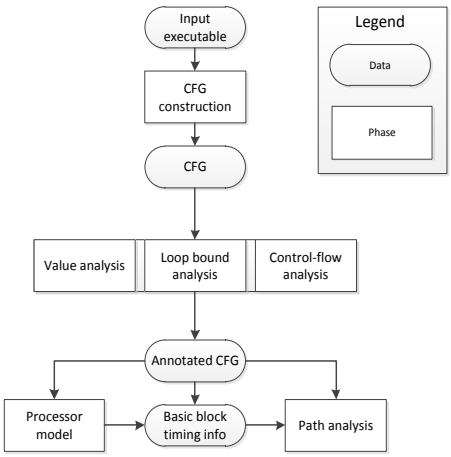

Figure 3. Steps for WCET calculation using static analysis

As seen from Figure 1, there are two main methods of finding WCET, measurement based methods and static methods. In measurement based methods, the execution time is measured either through direct measurement or simulation of the code by giving different inputs. The obvious drawback of this method is that the WCET can be underestimated in this way, as not all possible paths could be tested with the limited number of inputs. This fact is shown in figure 2, where the curve for the measurement-based experiments is shown with a solid line, while the curve with all possible inputs possible is shown with a dotted line. To overcome this, one could put a safety margin over the measured WCET. However, the safety margin is still just a guess and the picked WCET could end up less than the real WCET. One way to have better estimations is to measure the worst-case execution time of each basic block and then try to find the path with the worst-case time by adding the time taken by these blocks. However, this would only work for very simple processors. In the presence of advanced features, such as pipeline, branch predication, out-of-order execution and caches, this would not work. In the presence of these advanced features, the worst-case execution time of a block is dependent on the path followed by the program. For example, the cache misses in a basic block will be dependant upon from which path the program reached that basic block.

Due to the fact that measurement based methods underestimate WCET, we limit ourselves to only static methods. There are two major kinds of static methods used for calculating WCET, namely static analysis and model checking. The combination of these two is also employed in some cases. The details of these methods is discussed in detail in the next two sections.

The major steps taken in calculating WCET are shown in Figure 3 [1]. The input executable is read to construct a control flow graph (CFG). Afterwards, control-flow analysis (CFA) is performed which performs steps such as removing infeasible paths, trying to find loop bounds and determine frequencies of execution of paths, etc. At that point, the user could also provide information such as loop bounds which could not be found by CFA, or known values at certain locations in the program, so that CFA can more precisely find infeasible paths. The annotated CFG with the micro-architectural analysis is then used to find the WCET of each basic block. Finally path

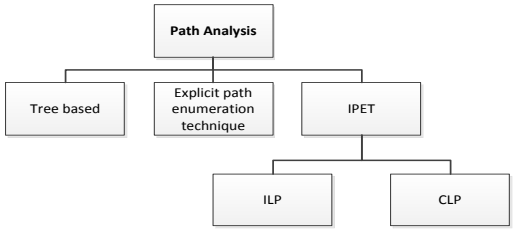

Figure 4. Path analysis methods used for WCET calculation

analysis is performed to find the WCET. Since the goal is to calculate a WCET which is at least equal to the real WCET, the calculated WCET is almost always greater than the real WCET, as shown in Figure 2. Therefore, the quality of a tool measuring WCET is assessed by how close WCET it calculates with respect to the real WCET, in other words how much tighter the WCET it calculates.

III. WCET FOR SINGLE CORE PROCESSORS

While the main focus of this paper is on finding WCET for multicore processors, it is first important to discuss the techniques applied to single core processors. This is because, even for single core processors, finding WCET is not straight forward, as typical single core processors are designed to have good average-case execution time, through features such as pipelining, cache memories, out-of-order execution, speculation and branch prediction. All these features, make accurate timing analysis a difficult problem [4].

These performance-enhancing features, also introduce timing anomalies. For example, one may assume that it would be safe to use a cache miss time for WCET calculation. However [21] showed that for out-of-order execution, it is possible that in some cases a cache hit would increase the time as compared to a cache miss.

There exist two main techniques for finding WCET for single core processors, namely static analysis and model checking. Static analysis is more computationally efficient in finding the WCET as compared to model checking, as model checking can have state-space explosion problems. However, model checking can find tighter WCET estimates. This is because, model checking can more accurately model a processor, while static analysis can just approximate the processor model, as it needs to find the WCET without actually running the program. These two techniques can be combined though to achieve the best results. In Section III-A, we will discuss related work using static analysis, while in Section III-B, we will discuss techniques which employ model checking for finding WCET.

A. Static analysis

Static analysis techniques try to find the WCET without actually running the program. Since they do not run the program, they need to approximate the processor model. Therefore, static analysis techniques are divided into two steps. First step is performing CFA on the CFG, and performing value analysis to find loop bounds, values to eliminate infeasible paths and addresses to help in finding cache hits and misses, followed by processor modeling to obtain the WCET of each basic

block in the program. There are three techniques to do this, abstract interpretation (AI), integer linear programming (ILP) and constraint logic programming (CLP). The second step is to find the WCET using WCET of the basic blocks. Different techniques employed for this purpose (Path analysis) are shown in Figure 4.

1) Tree based: The tree based method for path analysis traverses the CFG in bottom-up fashion, combining the WCETs of the basic blocks along the way (see [30] for more detail). This method is quite efficient but suffers from some limitations. For example, it is not possible to represent goto statements. Also, it is difficult to eliminate infeasible paths. An example of a paper using this technique is [9], which employs this technique for a processor with pipeline, branch prediction and an instruction cache.

2) Explicit path enumeration technique: This technique tries to find the longest path in terms of execution time by looking for all the possible paths in the program. It first tries to eliminate all infeasible paths in the program. This method suffers from low performance, as the number of paths that need to be examined increases exponentially with the program size.

3) ILP: Due to the problem associated with explicit-path-enumeration technique, authors in [19] propose integer linear programming (ILP) to solve the WCET problem implicitly. That is why this techniques is known as implicit-path-enumeration technique (IPET). Equation 1 is the basic equation of calculating WCET with this technique. Here c is the cost of basic block i and x is the number of times that basic block is executed. The WCET is given by finding the maximum value of Equation 1.

$$\sum_{i=1}^{n} c_i x_i. \tag{1}$$

The authors of [19] extended their work to also account for architectures with instruction caches in [20]. Both modeling of the instruction cache (processor modeling) and calculating of WCET (path analysis) is done using ILP. A basic block is further divided into line blocks, where each line block represents contiguous instructions which use the same cache line. Also, information is kept for a line block whether it incurs a cache miss or is a cache hit. This method might work for simple models, but for more complex processor architectures, which include pipelining, speculative execution, branch prediction and out-of-order execution for example, it becomes prohibitively difficult to use ILP due to its restrictive nature. For those purposes, Abstract Interpretation (AI), which is discussed next, is much more feasible.

4) AI+ILP: Abstract Interpretation (AI) is a dataflow technique to approximate model of a processor. AI can be used for example to get a set of possible values for a variable. However, since AI is an approximate method, it might also include values in a set, which would not occur in the program. Therefore, techniques using AI overestimate WCET at the cost of finding WCET in less time as compared to model checking.

That is why authors in [29] separate processor modeling and path analysis steps. For processor modeling, they use AI. Through AI, they model pipeline and caches. Through AI, they can classify an instruction as always hit, always miss, persistent (miss for first time and then always hit) or unclassified. In the case of unclassified, both scenarios are considered, that is cache hit and cache miss, as previously discussed that due to timing anomalies, it is not enough to consider cache miss as the worst case scenario.

While [29] checks for always miss, always hit and unclassified instructions in a global scope, [15] also consider local scopes like loops and functions. They argue that the same cache line block used in different scopes might not interfere with each other and therefore would be mutually exclusive, so in this way we could have blocks which are classified as persistent only in that local scope. This method reduced estimates of the WCET by upto 74%.

5) CLP: Another alternative of processor modeling and path analysis using ILP is to use constraint logic programming (CLP). The drawback of ILP is that we are limited to only using linear constraint with ILP, and for representing disjunction, we have to duplicate a block. For example, if a block can be reached from two different paths, it has to be duplicated into two blocks, each having a different WCET, but with CLP, we can actually define through constraint equations the value of WCET values for that block for different paths. Authors in [22] showed that using CLP significantly reduced WCET calculation time as compared to ILP, as there are much less blocks required due to the ability of representing disjunctions through constraint equations.

B. Model checking

The problem with static analysis is that it is an approximate method, due to the approximate nature of the processor modeling steps. With model checking on the other hand, we can build a more concrete model of a processor, and therefore have tighter WCET estimates. For example, when cache accesses cannot be classified with AI, we have to check execution time with both cache miss and cache hit. On the other hand, with model checking, some of those instructions which were unclassified in AI, could be classified, thus tightening up the WCET estimates. [14] is an example which supports model checking for finding WCET for Java processors. The authors use UPPAAL [3] model checker for that purpose. The authors noticed that model checking was enough for typical tasks in embedded systems. However, for larger applications, it was too slow. The authors recommended that model checking could be combined with static analysis in such a way, that the more important code fragments could be analyzed with model checking while the remainder of the application with static analysis.

[23] also uses a model checker. Instead of using each instruction in the model checker, the authors use basic blocks, thus reducing the number of states for model checking.

[17] combine model checking with static analysis to find WCET. The model checking is useful in deriving loop bounds, which change dynamically. For example, for loop bounds that depend upon two variables, the user just has to feed a range of values of these variables and the model checker can extract the loop bounds from them.

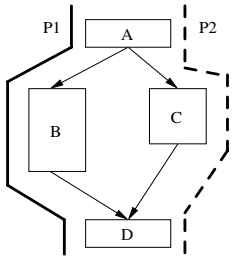

Figure 5. Example of timing anomaly in a multicore processor [32]

IV. WCET FOR MULTICORE PROCESSORS

We discussed previously that single core processors can have timing anomalies in the presence of complex performance enhancing features. Multicore processors have another source of timing anomalies due to shared resources, such as shared cache memory. An example is shown in Figure 5. Let us assume the path ABD is the worst-case path if seen separately. In the presence of shared L2 cache however, ACD might become worst-case path if a thread running on another core evicts more instructions from C than B in the L2 cache. Therefore, whenever analyzing WCET for a multicore, we always need to consider all the tasks running on different cores together, which can significantly increase the complexity of timing analysis.

In Section IV-A, we discuss the WCET calculation methods used for mutlicore systems, which are static analysis and model checking, whereas, in Section IV-B, we discuss techniques that assist in WCET estimation.

A. WCET calculation methods

Like in case of single core processors, WCET calculation techniques can also be divided into static analysis and model checking. These two techniques for multicore systems and their comparison is discussed next.

1) Static analysis: The first work done in this regard was by [32], which extends [29] to a multicore processor with private L1 caches but shared L2 cache. Through AI, this method tries to find out which instructions are always cache hits or always cache hits after the first time. It considers all other instructions as cache misses. This method first checks for L1 cache misses separately. An L1 cache miss implies either an L2 cache hit or an L2 cache miss. The basic idea is to check if the same cache block will be used by another thread running on another core. If that is the case, the basic block is marked as to have an L2 cache miss if the other thread is using that block with a loop, otherwise it is marked as always-except-one-hits. If the cache block is not used by the other core, then it is marked as always-hit. The WCET is found by solving the linear constraints formed by AI. The authors of this paper extend this method to also include data caches in [33]. The drawback of this method though is that it considers the effect of caches in isolation, that is not including performance enhancing features such as branch prediction and speculative execution which can cause timing anomalies. In case of timing anomalies, it is not enough to assume cache miss as the worst case.

To solve this problem [7] include pipelining, branch prediction and speculative execution in their analysis. Although they only consider instruction L2 caches. With timing anomaly in consideration, the timing analysis becomes more complex, as we cannot just assume a cache miss, if we are not sure about a cache access being a cache miss or a cache hit. This is the reason, the authors classify cache accesses as always-hit, always-miss and unclassified. For unclassified, both a cache hit and a cache miss are tried to find out the WCET.

[13] uses a technique similar to [7] but improves WCET calculation by employing bypassing of caches. Bypass of a load instruction for example, for a cache level means that if there is a miss, the memory block would not be brought into the cache, while if there is a cache hit, age of no cache block would be altered. In this way, instructions which are rarely used in a program could be bypassed, thus reducing inter-core conflicts and therefore improving timing analysis. The instructions to be bypassed are chosen by the compiler at compile time. [13] only considers bypassing for instruction caches, while [18] does it for both instruction and data caches. It has to be noted though that not every processor has a bypass instruction and therefore this method is not portable.

2) Model checking: Besides, static analysis, model checking is also a viable approach for calculating WCET for a multicore processor. We can either use model checking alone or combine it with static analysis. When model checking is used alone, both the processor modeling and path analysis is done using the model checker. The user can query the model checker with a guess WCET, to see if the maximum time calculated by the model checker is less than or equal to the guessed WCET. The guess is refined until the WCET is matched with the maximum time calculated by the model checker.

[31] uses model checking to estimate WCET. The model checking language used is PROMELA, which is the language of SPIN [2] model checker. The approach works for shared L2 caches. The authors show that using model checking improves the tightness of WCET as compared to static analysis only approaches. This is because model checking can check every possible interleaving of threads running on different cores, and therefore some cache accesses which cannot be classified as cache miss or cache hit, can be properly classified with a model checker. One problem with using a model checker is the state-space explosion problem. The authors of [31] tried to reduce this by first finding L1 and L2 cache hits and misses for a task by assuming it is running on a single core processor. This information is then imported for model checking the real scenario, that is tasks running on a multicore processor. In this way, only the L2 hits need to be taken care of, as L2 misses would still be misses on a multicore.

[8] combines static analysis with model checking to calculate WCET. AI is used to model the shared cache, but model checking is used to model the shared bus, which is the bus that is used to read from and write to the main memory. The main reason of using model checking for the shared bus is because it is much simpler to model it with a model checker as compared to modeling it with AI. Furthermore, since it is more accurate, it also gives tighter WCET estimates.

[12] uses UPPAAL [3] to model a multicore system with private L1 caches and a shared L2 cache. This method also works for tasks communicating with each other through shared memory using spin locks. Since, each instruction is modeled,

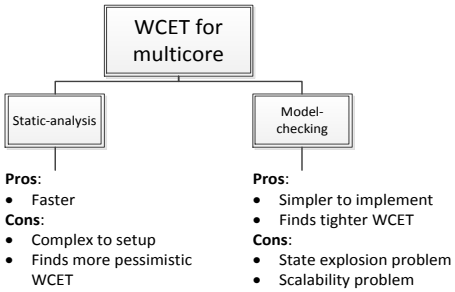

Figure 6. Model checking vs static analysis comparison

the state space is large, and therefore this method only works for small programs. Even for those programs, a WCET can only be calculated with two cores. For more cores, state space explosion is observed.

[6] uses model checking and static analysis. This method supports shared memory communication among the tasks. A program is divided into communication and execution slices. At the start of an execution slice, data is loaded into the private caches of the cores and at the end of the communication slice, data is put back in the main memory. State-space explosion occurs when there is too much communication involved. Also, this method slows down execution, as data has to be read from and written to the main memory at each execution and communication slice.

3) Model checking vs static analysis: Figure 6 compares static analysis with model checking for finding WCET for multicore processors. We can see that static analysis is faster but finds more pessimistic WCET as compared to model checking based approaches. Moreover, it is also more difficult to implement. The problem with model checking is that it suffers from scalability problem, as with more cores, there are more states possible, thus causing state-space explosion for larger programs. The good thing though is that model checking can be aided by static analysis to reduce those states, as done by some papers discussed in Section IV-A2. However, none of those papers used a processor with more than 2 cores, suggesting that even by combining static analysis with model checking, it is still difficult to find a scalable method.

B. Assisting WCET estimation

Hardware approaches can ease in estimating tighter WCET. For example, [10] proposes hardware mechanism to allow execution of synchronization operations such as mutex locks in bounded time. The logic for the hardware synchronization primitives (such as test-and-set, fetch-and-increment/decrement) is nested in the memory controller.

Cache locking and cache partitioning [28] can make the task of WCET calculation much easier. Cache partitioning means that the tasks running on different cores use a separate portion of the shared cache, while cache locking allows a user to load certain data in the cache and lock it, that is, prevent it from being replaced. The benefit of cache partitioning is that one could perform WCET calculation for tasks running on separate cores separately. While cache partioning can ease the calculation of WCET, it can also reduce performance, as due to less cache space available to a task, more cache misses could occur.

[16] proposes synchronized cache management to ease finding a tighter WCET. This is done by using page coloring. Physical pages of different colors do not cause cache conflict. Moreover, there are limited number of pages of the same color. Accesses within the same colored memory by different cores cause conflict only when the number of cache ways are exhausted. The authors view each color as a shared resource, where a lock is required to access that shared resource. For locking, the authors implemented a synchronization protocol.

[26] propose an interference-aware arbiter, through which the maximum time to access a shared resource by a hard real-time task (HRT), such as shared memory has an upper bound. The system assumes that both HRT and non-real time (NRT) tasks are running concurrently on the system and makes sure that the access to a shared resource by an HRT is bounded in time to ease WCET calculation.

V. Open issues

For single core processors, there are several tools available for estimating WCET of tasks as discussed in [30]. However, there is no such tool available yet for multicore processors, as we saw that timing analysis for multicore processors is much more complex due to increased number of states possible due to access of shared resources. Here, we discuss the still open issues for solving the problem of estimating WCET for multicore systems.

A. Scalability and precision

Although the scalability of static analysis is better as compared to model checking, the static analysis methods are still not very scalable, as the possible number of states with a multicore processor is still much more than that of a single core one. There are no papers yet that use more than two cores for experiments. All of the static analysis approaches that we discussed use ILP for path analysis. It would be interesting to use CLP instead, because [22] showed that CLP is much faster than ILP on single core.

A combination of model checking and static analysis methods could represent the most appropriate solution. One way to solve the scalability issue would be to only perform model checking on the compute-intensive part of the code and use static analysis for the rest.

B. Synchronization

Almost all the methods that we discussed for finding WCET on multicore processors ignore the problem of data sharing between the cores, and those that do consider it have some limitations. [6] writes back data to the main memory after every communication slice and reads it back from the main memory at the start of each execution slice, thus incurring a much larger overhead as compared to keeping the shared data in cache. [10] describes a hardware approach to bound the time of synchronization operations, but the obvious drawback of this method is that it needs hardware modifications, thus impacting portability.

One possible solution is to use determinsitic execution [25] [24], where locks for shared memory access are acquired in such a way that there is only one schedule possible. Although

this method ensures determinism of shared memory accesses for only a given input, [5] showed that even when an exhaustive set of inputs is considered, deterministic execution can have a smaller schedule space than non-deterministic approaches. However, to employ this method, the problem of global clock reading that [25] and [24] employ would need to be solved first, as global clock reading can cause cache evictions and therefore increase cache coherence activity.

VI. Conclusion

In this paper, we discussed the challenges of finding WCET for multicore processors and discussed some recent approaches in that direction. However, none of the existing approaches have been tried and tested for more than two cores, thus raising the concern of scalability of such approaches. Also, most of these approaches ignore the fact that data could be shared between the cores. Those that do, suffer from performance or portability problems. We also gave suggestions on how this scalability and synchronization problem could be solved.

Acknowledgment

This work is carried out under the BENEFIC project (CA505), a project labeled within the framework of CATRENE, the EUREKA cluster for Application and Technology Research in Europe on NanoElectronics.

References

[1] http://compilation.gforge.inria.fr/2010_12_Aussois/programpage/pdfs/MAIZA.Claire.aussois2010.pdf

[2] http://spinroot.com/spin/whatispin.html

[3] Gerd Behrmann, Re David, and Kim G. Larsen. A tutorial on uppaal. pages 200–236. Springer, 2004.

[4] Christoph Berg, Jakob Engblom, and Reinhard Wilhelm. Requirements for and design of a processor with predictable timing. In *Perspectives Workshop: Design of Systems with Predictable Behaviour, 16.-19. November 2003, volume 03471 of Dagstuhl Seminar Proceedings. IBFI, Schloss Dagstuhl*, 2004.

[5] Tom Bergan, Joseph Devietti, Nicholas Hunt, and Luis Ceze. The deterministic execution hammer: How well does it actually pound nails? In *2nd Workshop on Determinism and Correctness in Parallel Programming*, 2011.

[6] Frédéric Boniol, Hugues Cassé, Eric Noulard, and Claire Pagetti. Deterministic execution model on cots hardware. In *Proceedings of the 25th International conference on Architecture of Computing Systems*, pages 98–110, Berlin, Heidelberg, 2012.

[7] S. Chattopadhyay, C.L. Kee, A. Roychoudhury, T. Kelter, P. Marwedel, and H. Falk. A unified wcet analysis framework for multi-core platforms. In *Proceedings of the 18th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 99–108, 2012.

[8] Sudipta Chattopadhyay, Abhik Roychoudhury, and Tulika Mitra. Modeling shared cache and bus in multi-cores for timing analysis. In *Proceedings of the 13th International Workshop on Software & Compilers for Embedded Systems*, pages 6:1–6:10, New York, NY, USA, 2010.

[9] Antoine Colin and Isabelle Puaut. A modular and retargetable framework for tree-based wcet analysis. In *Proceedings of the of the 13th Euromicro Conference on Real-Time Systems*, pages 37–44, 2001.

[10] M. Gerdes, F. Kluge, T. Ungerer, C. Rochange, and P. Sainrat. Time analysable synchronisation techniques for parallelised hard real-time applications. In *Design, Automation Test in Europe Conference Exhibition 2012*, pages 671–676.

[11] Sylvain Girbal, Miquel Moretó, Arnaud Grasset, Jaume Abella, Eduardo Quiñones, Francisco J. Cazorla, and Sami Yehia. On the convergence of mainstream and mission-critical markets. In *Proceedings of the 50th Annual Design Automation Conference*, pages 185:1–185:10, New York, NY, USA, 2013.

[12] Andreas Gustavsson, Andreas Ermedahl, Björn Lisper, and Paul Pettersson. Towards wcet analysis of multicore architectures using uppaal. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, pages 103–113. Österreichische Computer Gesellschaft, 2010.

[13] D. Hardy, T. Piquet, and I. Puaut. Using bypass to tighten wcet estimates for multi-core processors with shared instruction caches. In *Proceedings of the 30th Real-Time Systems Symposium*, pages 68–77, 2009.

[14] Benedikt Huber and Martin Schoeberl. Comparison of implicit path enumeration and model checking based wcet analysis. In *In Proceedings of the 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2009.

[15] Bach Khoa Huynh, Lei Ju, and A. Roychoudhury. Scope-aware data cache analysis for wcet estimation. In *Real-Time and Embedded Technology and Applications Symposium*, pages 203–212, 2011.

[16] Christopher J. Kenna, Jonathan L. Herman, Bryan C. Ward, and James H. Anderson. Making shared caches more predictable on multicore platforms. In *Euromicro Conference on Real-Time Systems*, 2013.

[17] Sungjun Kim, Hiren D. Patel, and Stephen A. Edwards. Using a model checker to determine worst-case execution time. Technical report, Columbia University Computer Science Technical Reports, 2009.

[18] Benjamin Lesage, Damien Hardy, and Isabelle Puaut. Shared Data Caches Conflicts Reduction for WCET Computation in Multi-Core Architectures. In *Proceedings of the 18th International Conference on Real-Time and Network Systems*, page 2283, Toulouse, France, 2010.

[19] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, pages 456–461, New York, NY, USA, 1995.

[20] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Performance estimation of embedded software with instruction cache modeling. In *ACM TRANSACTIONS ON DESIGN AUTOMATION OF ELECTRONIC SYSTEMS*, pages 257–279, 1999.

[21] T. Lundqvist and P. Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, pages 12–21, 1999.

[22] Amine Marref and Guillem Bernat. Predicated worst-case execution-time analysis. In *Proceedings of the 14th Ada-Europe International Conference on Reliable Software Technologies*, pages 134–148, Berlin, Germany, 2009.

[23] Alexander Metzner. Why model checking can improve wcet analysis. In Rajeev Alur and DoronA. Peled, editors, *Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 334–347. Springer Berlin Heidelberg, 2004.

[24] H. Mushtaq, Z. Al-Ars, and K. Bertels. Detlock: Portable and efficient deterministic execution for shared memory multicore systems. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 721–730, 2012.

[25] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: efficient deterministic multithreading in software. *SIGPLAN Not.*, 44(3):97–108, March 2009.

[26] Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. Hardware support for wcet analysis of hard real-time multicore systems. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 57–68, New York, NY, USA, 2009. ACM.

[27] H. Shah, A. Raabel, and A. Knoll. Challenges of wcet analysis in cots multi-core due to different levels of abstraction. In *Workshop on High-performance and Real-time Embedded Systems (HiRES 2013)*, 2013.

[28] V. Suhendra and T. Mitra. Exploring locking partitioning for predictable shared caches on multi-cores. In *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pages 300–303, 2008.

[29] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise wcet prediction by separated cache and path analyses. *Real-Time Syst.*, 18(2/3):157–179, May 2000.

[30] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case-execution-time problem-overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.

[31] Lan Wu and Wei Zhang. A model checking based approach to bounding worst-case execution time for multicore processors. *ACM Trans. Embed. Comput. Syst.*, 11(S2):56:1–56:19, August 2012.

[32] Jun Yan and Wei Zhang. Wcet analysis for multi-core processors with shared l2 instruction caches. In *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS '08. IEEE*, pages 80–89, 2008.

[33] Wei Zhang and Jun Yan. Accurately estimating worst-case execution time for multi-core processors with shared direct-mapped instruction caches. In *Embedded and Real-Time Computing Systems and Applications, 2009. RTCSA '09. 15th IEEE International Conference on*, pages 455–463, 2009.