TOWARDS WCET ANALYSIS OF MULTICORE ARCHITECTURES USING UPPAAL[1]

Andreas Gustavsson[2], Andreas Ermedahl[2],
Björn Lisper[2], and Paul Pettersson[2]

Abstract
To take full advantage of the increasingly used shared-memory multicore architectures, software algorithms will need to be parallelized over multiple threads. This means that threads will have to share resources (e.g. some level of cache) and communicate and synchronize with each other. There already exist software libraries (e.g. OpenMP) used to explicitly parallelize available sequential C/C++ and Fortran code, which means that parallel code could be easily obtained.

To be able to use parallel software running on multicore architectures in embedded systems with hard real-time constraints, new WCET (Worst-Case Execution Time) analysis methods and tools must be developed. This paper investigates a method based on model-checking a system of timed automata using the UPPAAL tool box. It is found that it is possible to perform WCET analysis on (small) parallel systems using UPPAAL. We also show how to model thread synchronization using spinlock-like primitives.

1. Introduction

The execution of hard real-time systems must be predictable in order to ensure a certain system behavior. In particular, the WCETs (Worst-Case Execution Times) of the hard real-time tasks are assumed to be known and given as input to different real-time system scheduling algorithms [4, 10, 17]. The WCET of a task is dependent both on the properties of the software which is executed as well as the underlying hardware. Today, there are algorithms and tools which strive to derive a safe and tight bound on the WCET of a task, using the task code and a model of the (single-core) target hardware. Some examples of such tools are aiT [9, 27], SWEET [8, 27] and RapiTime [23, 27].

Over the past years, there has been (and there will probably continue to be) a rapid increase in the usage of multicore architectures in embedded real-time systems. These architectures have several independent processing units (cores) on each chip. The cores typically share some resources (e.g. some level of on-chip cache) which introduces dependencies among the cores. Thus the cores could experience delays due to simultaneous access to these shared resources; e.g., if the L1 caches are non-shared and the L2 cache is shared, two simultaneous misses in the L1 caches will cause one of the cores to delay while the other core is granted access to the L2 cache. If there are one or more levels of core-individual (non-shared) caches, some memory coherence and consistency model will probably be implemented. This means that a line in the local cache of one core may be invalidated by another core's cache, thus introducing a cache miss if the line is again referenced [1].

[1]This work was funded by VR through the project 2008-4650 Worst-Case Execution Time Analysis of Parallel Systems.
[2]School of Innovation, Design and Engineering, Mälardalen University, Box 883, S-721 23 Västerås, Sweden.
{andreas.sg.gustavsson,andreas.ermedahl,bjorn.lisper,paul.pettersson}@mdh.se

To take full advantage of these new kinds of architectures, algorithms will need to be parallelized over multiple threads. This means that the threads will have to share resources and communicate and synchronize with each other. There already exist software libraries used to explicitly parallelize sequential code – one example available for C/C++ and Fortran code running on shared-memory machines is OpenMP [20]. The conclusion is that parallel software running on parallel hardware is already available today and will probably be the standard way of computing in the future.

This means that new algorithms, methods and tools for WCET analysis are needed to guarantee the schedulability and predictability of this new kind of systems, where a task could consist of several cooperating threads running in parallel on individual cores. This paper presents a method for WCET analysis of parallel (or sequential) code executing on shared-memory multicore (or single-core) architectures, using verification techniques (model-checking) on a system of timed automata. The paper shows that it is possible to model and analyze the impact on the WCET from having a memory hierarchy consisting of core-individual L1 instruction and data caches, and a shared L2 cache. It also shows how a mutual exclusion software primitive similar to a spinlock could be modeled.

The organization of the rest of this paper is as follows. Section 2 presents some related research performed on analysis of multicore architectures. Section 3 contains an introduction to timed automata and the modeling tool box UPPAAL [5]. Section 4 describes the models and verification queries used to calculate the WCET estimate of an example program. Section 5 contains a discussion of the proposed method. It also suggests several aspects of the method that should be further investigated.

2. Related Work

The idea of using model-checking to perform WCET analysis has been investigated and shown to be adequate for analyzing parts of a single-core system in [14] and [19]. However, to the best of our knowledge, no prior research has been conducted regarding multicores with complete (and non-perfect) memory hierarchies. This aspect is investigated in this paper.

In [18] and [28], model-checking is used to perform WCET analysis. Both papers are closely related to the work presented herein, but mainly propose methods to reduce the state space by altering the program model without affecting the true WCET of the program. Our approach is more focused on analyzing the impact on the WCET from allowing synchronizing tasks. In [28], a perfect data cache is assumed (i.e., all accesses are assumed to be hits), which is generally not the case. In contrast, this paper assumes a complete and non-perfect memory hierarchy. In [29] and [30], static analyses of shared L2 instruction caches are presented. Also in these papers, perfect L1 data caches are assumed.

Other than this, to the best of our knowledge, there mainly exist different techniques used to increase the predictability and analyzability (e.g. to tighten the WCET estimate) of multicore systems. In an extension to the method presented in [29], memory bits for each instruction are used to determine whether the instruction should be cached or not [12] – e.g., to avoid pollution of the shared cache, “Static Single Usage” [12] instructions should not be cached. This generates the possibility to determine a tighter WCET estimate.

In [21], arbiters (hardware circuits) are added to a shared-memory multicore processor to synchronize the memory accesses in order to increase the timing-predictability of the system. The result is a multicore architecture that can be analyzed with existing single-core WCET analysis tools.

GAMC [22] is an SDRAM controller which upper bounds the delay a core can suffer from memory-interferences from other cores. This is an important aspect since the largest memory access latency will occur when accessing the main memory. The result is a tight WCET estimate which only differs at most 13% from the largest measured execution time. Similarly, in [4] and [24], TDMA-based memory bus access policies are introduced to make all memory access latencies predictable, regarding the WCET.

3. Timed Automata & UPPAAL

Timed automata[3] [3] can be used to model real-time systems. An automaton can be viewed as a state machine with locations and edges [15]. A state represents certain values of the variables in the system and which location of an automaton is active, while the edges represent the possible transitions from one state to another [15]. (Continuous) time is expressed as a set of real-valued variables modeling clocks. In UPPAAL, all clocks are initialized to zero and then increase with the same rate [7].

A transition is enabled (i.e., it is possible to perform the particular transition from one state to another) if its accompanying guard is satisfied. A guard can simply be viewed as a boolean expression (which can include variables and clocks) which enables or disables the edge. The guard cannot force the transition to be taken however [7]. When a transition is taken, actions can be performed (e.g., variables can be updated and clocks can be reset to zero).

UPPAAL[4] [5, 16, 26] is a tool used to model, simulate and verify *networks* of timed automata [5, 7, 15]. The automata can synchronize via channels on transitions. Only two automata are allowed to synchronize via a given regular channel at a time. Channels can also be declared as being broadcast, which means that one issuing automaton can synchronize with an arbitrary number (including zero) of waiting automata. Another possibility is to declare a channel as being urgent, which means that when a transition is enabled, it will be performed without allowing any time to pass.

Locations in an UPPAAL timed automaton can have special properties as well; urgent or committed. When a location with one of these properties is active, time is not allowed to pass. The difference between urgent and committed locations is that if there are committed locations active, an outgoing transition from one such location must be taken in the next step – if such a transition does not exist or is not enabled, the system will deadlock. A location in the automaton can have an invariant associated with it. An invariant is a clock constraint which limits the amount of time for which the location is allowed to be active.

Some other features of UPPAAL are a C-like programming interface to ease the modeling task, and meta-variables [5]. If the only difference between two states is the values of variables declared as meta, then the states are considered to be the same. This is useful for reducing the size of the state space while verifying properties of the system. Care should be taken to avoid using meta-variables in a way that could eliminate states from the analysis that actually should be taken into account, though. Verification of system properties (requirements) is performed by formulating queries used by the UPPAAL verifier. The query language is described in e.g. [5] or in the help session accompanying the UPPAAL binaries [26].

[3]The formal syntax and semantics of timed automata can be found in e.g. [2] and [15].

[4]An introduction to UPPAAL and the formal semantics of networks of timed automata are given in [5] and [15] respectively.

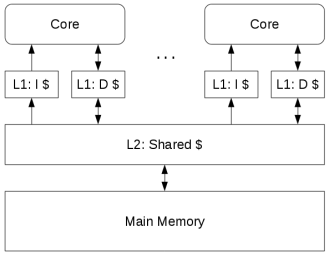

Figure 1: The modeled architecture.

Property	L1-I	L1-D	L2
Lines	4	4	8
Words/Line	2	2	4
Sets	2	4	2
Latency	1	1	10
Replacement Policy	LRU	LRU	LRU

Table 1: Cache Properties.

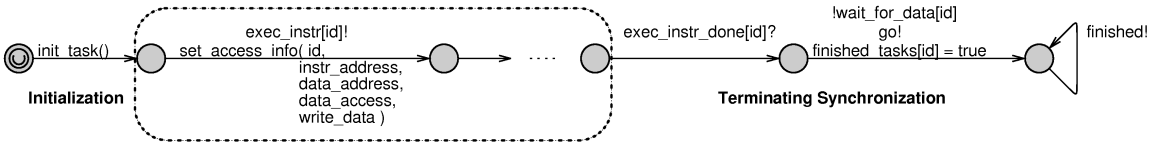

Figure 2: Model of the task interface.

4. WCET Analysis Using UPPAAL

To model a fictitious shared-memory multicore architecture, a network of timed automata is created in UPPAAL[5]. The architecture is assumed to have the properties depicted in Figure 1; i.e., core-individual L1 instruction and data caches, and a shared L2 cache. In the figure, the arrows between the cores and the caches show the possible flow of memory contents (i.e., instructions and data). The core is assumed to be very simple, only incorporating a pipeline similar to a basic five-stage, in-order RISC-pipeline. The caches are assumed to have the properties found in Table 1.

The resulting models are presented in Figure 3. For a multicore architecture with n cores, there will be n sets of the models in Figures 3a–3c (i.e., one set per core) but only 1 set of the models in Figures 3d–3g[6]. For the current approach, no value analysis is used. Therefore, in the below given models, no actual memory contents is ever transferred or kept track of in the memory hierarchy. The only thing considered is what memory locations (addresses) are referenced by the program. A limitation of this approach is that dynamic memory references cannot be easily modeled.

4.1. The Program Model Interface

The interface for modeling a thread is shown in Figure 2. The "Initialization" part is optional and the `init_task()` function could simply be empty. The "Terminating Synchronization" part ensures that no time is missed by the WCET analysis. If the pipeline should be emptied at the end, a delay should be inserted to account for this in this part of the model.

The middle (framed) part depicts the instruction execution interface. The instructions are assumed to be assembly instructions and are executed one by one. An instruction is executed by synchronizing with the core automaton via the `exec_instr[id]` urgent channel and setting information about the access via the function call `set_access_info()`. The arguments should be interpreted as: `id` – the core on which the instruction should be executed; `instr_address` – the memory address where

[5]UPPAAL version 4.0.10 (rev. 4417) has been used in this paper.
[6]With one exception regarding the Lock handler automaton – there is one Lock handler per lock, i.e., per critical section.

the instruction is stored; `data_address` – the address in memory on which the data accessed by the instruction is stored (only used for instructions such as LOAD and STORE etc.); `data_access` – a boolean telling whether the instruction is a data accessing instruction (e.g., a LOAD or STORE etc.); `write_data` – a boolean distinguishing between read and write instructions (i.e., whether the instruction is a LOAD or STORE etc.).

Other types of instructions, such as branch instructions and instructions not referencing memory locations, should be accounted for by adapting the structure of the automata modeling the program. Thus, the structure of the program should be represented by the structure of the automata. This representation could be automatically generated using flow facts generated by a static analysis tool, such as SWEET [8]. The translation would be close to 1:1 of the instruction-level CFG (Control Flow Graph) [18]. To account for hazards, extra stalls can be inserted into the pipeline by setting the `stalls[id]` variable to the desired value before executing the instruction.

To account for the possible memory locations that a given instruction could reference, a value analysis could be used [27]; and to account for the possible values of different variables affecting the execution pattern of the program, a control flow analysis could be used [27]. The structure of the automata modeling the program could then be adapted accordingly (e.g. by adding one transition for each possible memory reference or variable value). This means that UPPAAL will automatically account for the (global) worst-case memory reference or variable value. This approach could also avoid unwanted effects from timing anomalies since UPPAAL searches the entire state space when finding the WCET estimate.

4.2. The Model of the Core

The model of the core is depicted in Figure 3a. This automaton represents the timing model of the core (the pipeline etc.) and is the automaton with which the program-automaton synchronizes to execute instructions. When an instruction should be executed, the core accesses the memory hierarchy to fetch it and then steps the pipeline. If the instruction accesses data, the pipeline is stepped (stalls are inserted) until the memory access stage is reached, then the data is accessed. This leads to an over-approximation of the execution time. However, to avoid further over-approximation (which could be much larger), another instruction can be fetched while the data is accessed.

The `exec_instr_done[id]` channels are declared as broadcast so that the program-automata do not have to synchronize via these channels before a request to execute a new instruction can be issued. This is to minimize the number of locations in the program-automaton (to make the interface as clean as possible and to minimize the state space).

4.3. The Models of the Caches

The models of the L1 instruction and data caches are depicted in Figure 3b and 3c respectively. The main difference between these cache models is that a data cache has the ability to invalidate a line in the other data caches. Otherwise the models are quite straightforward. All the cache content handling is performed by the `access_cache_L1_{instr,data}()`, `update_cache_L1_data()` and `invalidate_L1()` functions.

If the accessed data is not available in the L1 cache, it is fetched from the L2 shared cache, which is depicted in Figure 3d. This model is even more straightforward – all the cache content handling

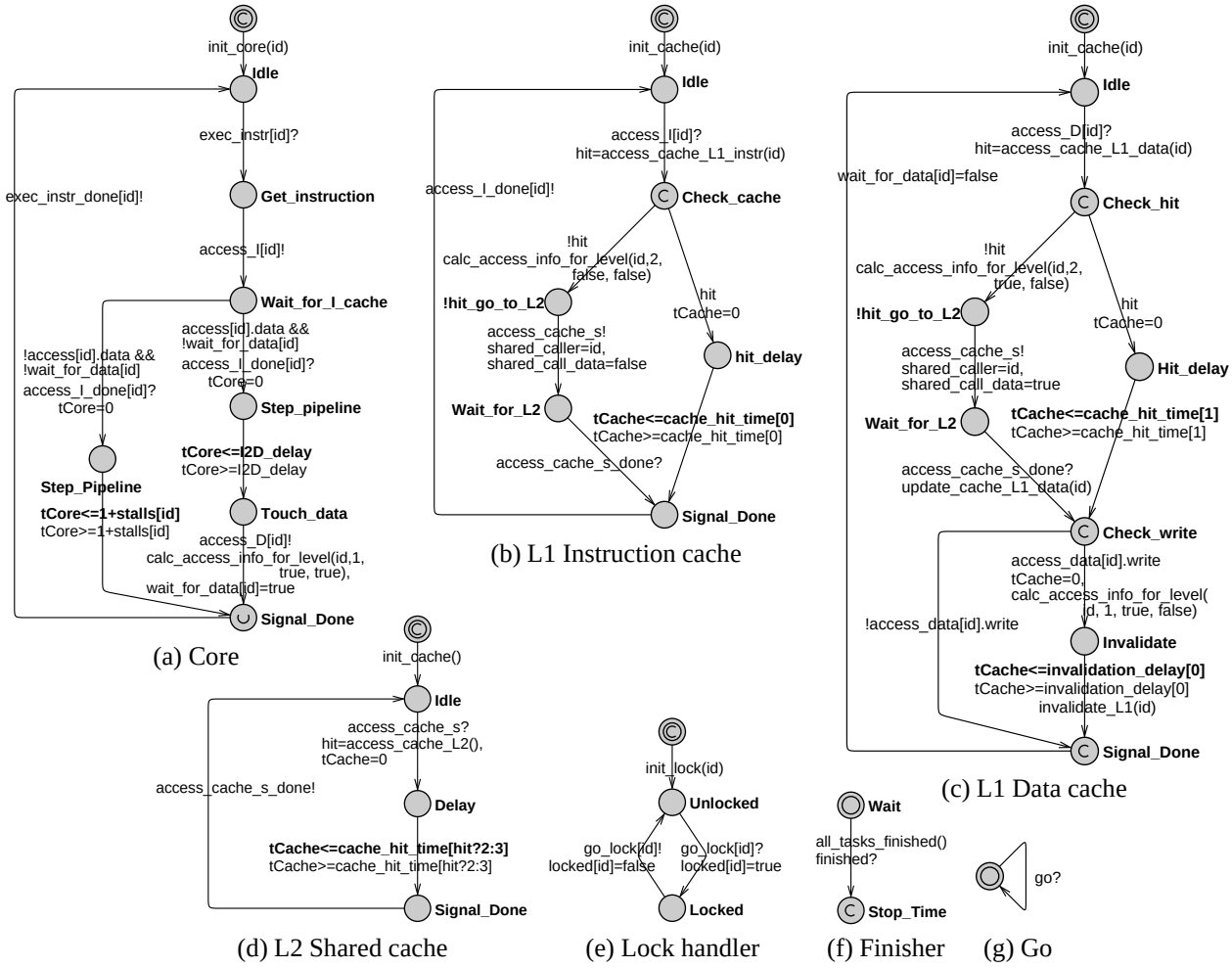

Figure 3: Timing model of the considered multicore architecture.

is performed by the `access_cache_L2()` function. If the accessed data is not located in the L2 cache, it is fetched from the main memory (which is assumed to always hit).

All the caches in the system can be individually defined, regarding set-associativity, cache size, block size and replacement policy (the used cache properties can be found in Table 1).

4.4. The Auxiliary Automata

These automata, depicted in Figures 3e–3g, are implementation specific. The Lock handler-automaton can be (and is) used to implement spinlocks. The Finisher-automaton is used to stop the time and deadlock the system when all tasks have finished executing. And finally, the Go-automaton is very versatile. It simply waits to synchronize via an urgent channel (thus not allowing any time to pass when the transition is enabled). This can be viewed as a trick to achieve the desired system behavior (e.g. to achieve system progress).

4.5. WCET Analysis by Verification

Given the above described network of timed automata, UPPAAL can verify if different properties hold for the system. The verification property that is used to find the WCET estimate looks like[7]: `A[] t <= x`. This property should be interpreted as: "For every possible state, the value of the clock `t` is always less than or equal to `x`". The WCET analysis is easily performed by running the model-checker (verifier) in a binary search style by altering the value of `x` until the WCET estimate is found[8].

In order for this approach to work, some other properties of the system must also be verified; otherwise there might exist some amount of time that is not accounted for when calculating the WCET estimate, or the overall system behavior could be incorrect. It must be verified that: whenever the system is in a deadlock state, the Finisher automaton is in its Stop_Time location; the system will always reach a state where the Finisher automaton is in its Stop_Time location; when the Finisher automaton is in its Stop_Time location, all other automata modeling the hardware are in their Idle locations, and all automata modeling the program have finished; and mutual exclusion is guaranteed on critical sections. By using similar verification properties to the one above, UPPAAL can check these properties automatically[9].

4.6. Experimental Evaluation

An example model of a program (using the interface given in Figure 2) is given in Figure 4. The task of the modeled program is very simple; it just acquires a spinlock-like lock and then writes to a shared variable before releasing the lock, and it executes this procedure three times before finishing its execution.

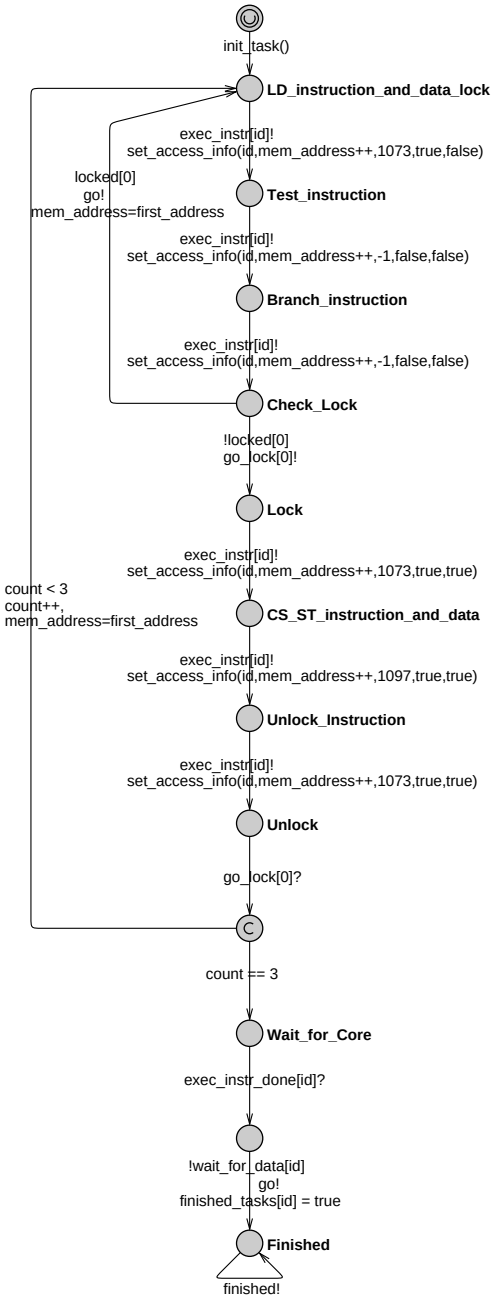

Figure 4: Model of a program with spinlock-like synchronization.

The same task is run on two cores (both tasks are released at the same time) and the result of the analysis is a WCET estimate equal to 636 clock cycles (the other properties mentioned above are also satisfied); using the specific values of the cache sizes (Table 1) and latencies etc. The main memory is assumed to have a latency of 80 clock cycles. Each step in the binary search approach is performed within 1 second and the total number of steps is 11 (this is dependent on the initial values of `x` in the verification property from section 4.5, however).

An initial investigation of some potential problems regarding the scalability of the model-checking

[7]The UPPAAL verifier syntax can be found in [5] or in the online help session accompanying the UPPAAL binaries [26].
[8]Similar approaches to WCET analysis using model-checking are described in [18], [19] and [28].
[9]To guarantee a safe verification, the UPPAAL option "Extrapolation" should be set to "None".

approach has been conducted. By increasing the number of cores to four and running one instance of the same example program as above on each core, we get a large slowdown in the analysis time. Another investigation, where the release time of the second task is made general in the interval $[0, 1000]$, has also been performed. The same result, a large slowdown in the analysis time, was observed. Increasing the sizes of the (meta-declared) caches to 2048 lines for the L1 caches and 8192 lines for the L2 cache, does not seem to have an equally large impact on the analysis time though. The memory usage increases drastically, however. The required times for performing one binary search step are summarized in the table below (a dual-core processor, running at 2.66GHz, with 4GB of RAM was used). The "2 Cores" column represents the original experiment and is the base for comparison. The total time is an approximation of the total time needed to perform the analysis, assuming 11 iterations, and that the binary search strategy for invoking the UPPAAL verifier is handled by a script.

	2 Cores	4 Cores	Release Time	$ Sizes
Time	<1s	>3h (aborted)	44s	14s
Total Time	11s	>33h	500s	150s

A consequence of these results is that the complexity of the models and the size of the analyzed program (and thus the achievable tightness of the WCET estimate) have to be balanced to avoid making the state space explode. The case with 4 cores was aborted after approximately 3 hours when the virtual memory demands exceeded the available amount of RAM (4GB).

5. Discussion & Future Work

Modeling systems is very easy using UPPAAL, which also offers a useful interface for performing model-checking. This paper has shown that WCET analysis of parallel code and hardware can be performed using the model-checking techniques available in e.g. UPPAAL. There are some limitations imposed by using UPPAAL to perform the WCET analysis, however. The C-style interface is a bit limited regarding function calls; e.g., an array-argument must have a known size – this limits the level to which the code can be written in a generical way. However, the UPPAAL C-functions are meant to be very simple and small and the C-style interface offered by UPPAAL is in general very rich, so the pros very much outweighs the cons.

Another drawback is the binary search strategy that has to be used for finding the WCET estimate. This could lead to unnecessarily large overheads in the analysis. One way to avoid the binary search approach is to use the new `sup`[10]-operator, implemented in (and described in the help session accompanying) the development version (4.1) of UPPAAL [26]. The `sup`-operator finds the maximum value of an expression evaluating to either an integer or a clock. To find the WCET estimate using the `sup`-operator, the following property could simply be verified: `sup: t`. This property should be interpreted as: "Find the maximum value of the clock `t`". This approach works for the proposed system model since the system is deadlocked and the time is stopped when all tasks have finished executing. The reason to why this approach is not used in this paper is because of the development (unstable) state of the UPPAAL-version (4.1) in which the `sup`-operator is implemented.

However, an initial investigation using the `sup`-operator has been performed on the system described in section 4.6. By verifying the property `sup: t`, it is found that the WCET estimate is 636 clock cycles (the same result as achieved by using the binary search approach). The total time needed to verify the property is in the order of 1 second – this is superior to the binary search approach where

[10] `sup` is an abbreviation of suprema.

approximately 1 second (plus the overhead needed to adjust the parameters) is needed for each binary search step.

An investigation of the sup-operator's impact on the scalability has also been conducted for the same system setups that were described in section 4.6. The result is presented in the table below.

	2 Cores	4 Cores	Release Time	$ Sizes
Time	1s	>3h (aborted)	42s	14s
Total Time	1s	>3h	42s	14s

As for the binary search approach, the case with 4 cores was aborted after approximately 3 hours when the virtual memory demands exceeded the available amount of RAM (4GB). As can be seen, the total time needed to perform the entire analysis using the sup-operator is quite comparable to the time needed to perform one binary search step (excluding any parameter adjustment overhead). This makes the sup-operator a very promising feature of UPPAAL; since the entire analysis can be performed automatically (in one step) and the implied overhead, if any, is negligible.

Further investigations should be performed, regarding how well this method (model-checking) scales with the size of the modeled program and the complexity of the hardware model. It would also be worth investigating the impact on the size of the state space (and thus the analysis time) by transferring more of the cache handling functionality from the cache automata to the cache handling C-functions, and vice versa. On one extreme, all the cache handling could be done by the C-functions, while the automaton only is used to perform the cache access delay.

Another way of (hopefully) increasing the scalability of the method is to extend the use of scalars. When scalars are used, UPPAAL can apply symmetry reduction on the model [13], which can lead to a dramatic decrease in the size of the state space. Symmetry reduction eliminates redundant paths in the model. Considering the models presented in section 4, there are lots of redundant paths. The same program is executed on several homogenous cores with a homogenous memory hierarchy. This means that the same execution pattern exists several times in the state space, the only difference is which program (and core and caches) it concerns. As a simple example, either program 0 is considered to start before program 1, or vice versa – only one of the possibilities needs to be considered since the models are equal; this is what symmetry reduction tries to achieve. Scalars and symmetry reduction are also described in more detail in the UPPAAL help session[11].

The granularity of the proposed interface in this paper is on the instruction level. This increases the size of the state space compared to using a basic block granularity. One way of reducing the size of the state space, and keep the instruction level granularity (when considering non-preemptive tasks at least), could be to merge instructions on the same cache line that do not access data and add some additional delay in the program model to represent the merging. This would be possible since the lines in the (non-shared) instruction cache never are invalidated by another cache; if one instruction is available, all other instructions in the same line are also available. This approach can be viewed upon as a manually performed partial order reduction [6, 11].

The static WCET analysis tool SWEET[12] is already capable of generating models in the UPPAAL

[11]The UPPAAL help session accompanies the UPPAAL binaries, available at [26]. It is also available at `http://www.uppaal.org/help.php?file=WebHelp` (for the official release of UPPAAL).

[12]SWEET uses basic blocks by default [8], but does also have the capability of using an instruction level granularity. This makes interaction possible.

syntax on a special format [25]. Performing minor changes to this generation could adapt SWEET to also being able to create models on the format specified by this paper. This means that benchmarks could be easily translated and analyzed together with the hardware models presented herein.

Other suggestions for future work are to implement a more detailed timing model to avoid over-approximating the WCET, to implement a model of a real-world multicore architecture, such as e.g. the ARM Cortex, and to investigate the possibilities of implementing models of more synchronization primitives, e.g. mutexes and condition variables.

A final and very important conclusion is that WCET analysis of the inter-thread communication and interferences on shared resources can be made quite simple using the suggested model-checking method, compared to static analysis (see e.g. [29]). However, it will probably be quite difficult to make the model-checking method scale well.

References

[1] ADVE, S. V., AND GHARACHORLOO, K. Shared Memory Consistency Models: A Tutorial. Tech. rep., Rice University and Western Research Laboratory, 1995.

[2] ALUR, R. Timed Automata. In *Computer Aided Verification* (Jan. 1999), vol. 1633/1999, Springer Berlin / Heidelberg.

[3] ALUR, R., AND DILL, D. L. A theory of timed automata. *Theoretical Computer Science 126*, 2 (Apr. 1994), 183 – 235.

[4] ANDREI, A., ELES, P., PENG, Z., AND ROSÉN, J. Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. *International Conference on VLSI Design 21* (2008), 103–110.

[5] BEHRMANN, G., DAVID, A., AND LARSEN, K. G. A Tutorial on UPPAAL.

[6] BENGTSSON, J., JONSSON, B., LILIUS, J., AND YI, W. Partial Order Reductions for Timed Systems. In *CONCUR'98 Concurrency Theory* (Feb. 1998), vol. 1466/1998, Springer Berlin / Heidelberg, pp. 41–49.

[7] BENGTSSON, J., AND YI, W. Timed Automata: Semantics, Algorithms and Tools. In *Lectures on Concurrency and Petri Nets* (July 2004), vol. 3098/2004, Springer Berlin / Heidelberg, pp. 87–124.

[8] ERMEDAHL, A. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Dept. of Information Technology, Sweden, June 2003.

[9] FERDINAND, C., HECKMANN, R., AND FRANZEN, B. Static Memory and Timing Analysis of Embedded Systems Code. In *3^{rd} European Symposium on Verification and Validation of Software Systems (VVSS'07)* (Mar. 2007), pp. 153–163.

[10] GUAN, N., STIGGE, M., YI, W., AND YU, G. New Response Time Bounds for Fixed Priority Multiprocessor Scheduling. In *Proc. 30^{th} IEEE Real-Time Systems Symposium (RTSS'09)* (Dec. 2009), pp. 387–397.

[11] HÅKANSSON, J., AND PETTERSSON, P. Partial Order Reduction for Verification of Real-Time Components. In *Formal Modeling and Analysis of Timed Systems* (Sept. 2007), vol. 4763/2007, Springer Berlin / Heidelberg, pp. 211–226.

[12] HARDY, D., PIQUET, T., AND PUAUT, I. Using Bypass to Tighten WCET Estimates for Multi-Core Processors with Shared Instruction Caches. In *Proc. 30^{th} IEEE Real-Time Systems Symposium (RTSS'09)* (2009), pp. 68–77.

[13] HENDRIKS, M., BEHRMANN, G., LARSEN, K. G., NIEBERT, P., AND VAANDRAGER, F. Adding Symmetry Reduction to UPPAAL. In *Formal Modeling and Analysis of Timed Systems* (May 2004), vol. 2791/2004, Springer Berlin / Heidelberg, pp. 46–59.

[14] HUBER, B., AND SCHOEBERL, M. Comparison of Implicit Path Enumeration and Model Checking Based WCET Analysis. In *Proc. 9th International Workshop on Worst-Case Execution Time Analysis (WCET'2009)* (2009).

[15] KATOEN, J.-P. Concepts, Algorithms, and Tools for Model Checking. In *Lecture Notes of the Course "Mechanised Validation of Parallel Systems" (course number 10359)* Semester 1998/1999, at Friedrich-Alexander Universität Erlangen-Nürnberg.

[16] LARSEN, K. G., PETTERSSON, P., AND YI, W. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer (STTT) 1*, 1-2 (Dec. 1997), 134–152.

[17] LIU, C. L., AND LAYLAND, J. W. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM 20*, 1 (1973), 46–61.

[18] LV, M., GUAN, N., YI, W., DENG, Q., AND YU, G. Efficient Instruction Cache Analysis with Model Checking. In *Proc. 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'10), Work-in-Progress Session* (Apr. 2010), pp. 33–36.

[19] METZNER, A. Why Model Checking Can Improve WCET Analysis. In *Computer Aided Verification* (July 2004), vol. 3114/2004, Springer Berlin / Heidelberg, pp. 298–301.

[20] OPENMP. OpenMP Application Program Interface, Version 3.0, May 2008. `http://www.openmp.org/mp-documents/spec30.pdf`.

[21] PAOLIERI, M., QUIÑONES, E., CAZORLA, F. J., BERNAT, G., AND VALERO, M. Hardware support for WCET analysis of hard real-time multicore systems. In *Proc. 36th International Symposium on Computer Architecture (ISCA 2009)* (2009), pp. 57–68.

[22] PAOLIERI, M., QUIÑONES, E., CAZORLA, F. J., AND VALERO, M. GAMC: A Generic Analyzable Memory Controller for Hard Real-Time Multicore Processors. Tech. rep., Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, May 2009.

[23] RAPITIME. Rapitime white paper, 2009. `www.rapitasystems.com/system/files/RapiTime-WhitePaper.pdf`.

[24] ROSÉN, J., ANDREI, A., ELES, P., AND PENG, Z. Bus Access Optimization for Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. In *Proc. 28th IEEE Real-Time Systems Symposium (RTSS'07)* (2007), pp. 49–60.

[25] SUNDMARK, D. *Structural System-Level Testing of Embedded Real-Time Systems*. PhD thesis, Mälardalen University, Department of Computer Science and Electronics, Sweden, 2008.

[26] UPPAAL. UPPAAL Website, 2010. `http://uppaal.org`.

[27] WILHELM, R., ENGBLOM, J., ERMEDAHL, A., HOLSTI, N., THESING, S., WHALLEY, D., BERNAT, G., FERDINAND, C., HECKMANN, R., MITRA, T., MUELLER, F., PUAUT, I., PUSCHNER, P., STASCHULAT, J., AND STENSTRÖM, P. The Worst-Case Execution Time Problem — Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems (TECS) 7*, 3 (2008), 1–53.

[28] WU, L., AND ZHANG, W. Bounding Worst-Case Execution Time for Multicore Processors through Model Checking. In *Proc. 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'10), Work-in-Progress Session* (Apr. 2010), pp. 17–20.

[29] YAN, J., AND ZHANG, W. WCET Analysis for Multi-Core Processors with Shared L2 Instruction Caches. In *Proc. 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'08)* (June 2008), pp. 80–89.

[30] YAN, J., AND ZHANG, W. Accurately Estimating Worst-Case Execution Time for Multi-Core Processors with Shared Direct-Mapped Instruction Caches. In *Proc. 15th International Conference on Real-Time Computing Systems and Applications (RTCSA'09)* (Aug. 2009), pp. 455–463.