Calculation of Worst-Case Execution Time for Multicore Processors using Deterministic Execution

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*—Safety critical real time systems need to meet strict timing deadlines. We use a model checking based approach to calculate the WCET, where we apply optimizations to reduce the number of states stored by the model checker. Furthermore, we used deterministic shared memory accesses to further reduce calculation time, memory and number of states needed for calculating WCET. By optimizing the model checking code, we were able to complete benchmarks which otherwise were having state explosion problems. Furthermore, by using deterministic execution, we significantly reduced the calculation time (up to 158x), memory (up to 89x) and states needed (up to 188x) for calculating WCET with a negligible increase (up to 4%) in the calculated WCET for a multicore system with 4 cores. Lastly, unlike other state-of-the-art approaches, that perform binary search to search the WCET by running several iterations, our method calculates WCET in just one iteration. Taking all these optimizations into consideration, the gain in speed was from 1775x to 2471x for 4 threads.**

I. Introduction

Adapting multicore systems to real time embedded systems is a challenging task, as a real time process, besides being error free, must also meet timing deadlines. The real time scheduler needs to know the worst-case execution time (WCET) of each task. Finding a good WCET estimate (less pessimistic) 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 increase in interest to solve the problem of finding WCET for tasks running on multicore processors, from on-chip hardware support to software solutions for commodity off the shelf (COTS) processors. But most of those do not take into account the shared memory accesses. In [6], the authors do take into account shared memory accesses, but the state explosion problem of the model checking based approach they use limits the effectiveness of that approach.

In this paper, we investigate, whether deterministic shared memory accesses [7] [8] would reduce the possible number of states used by the model checker and therefore reduce the WCET calculation time. The contributions of this paper are as follows.

- Limiting the state space explosion problem by utilizing deterministic execution when calculating the WCET of a multithreaded program running on multicores using model checking.
- Implementing optimizations to further reduce the size of the state space as well as to get a tighter WCET estimation.
- Using only one iteration to calculate the WCET rather than performing binary search as used by the current state-of-the-art approaches (which requires several iterations).

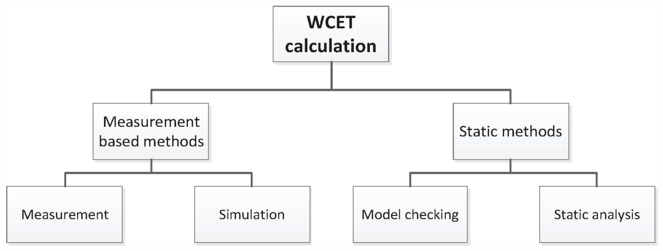

Figure 1. Methods used for WCET calculation

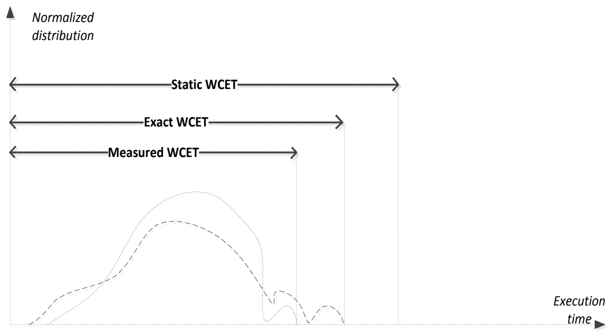

Figure 2. Measurement based vs static methods

In Section II, we discuss the background, while in Section III, we discuss the implementation. This is followed by the Section IV, which discusses the performance evaluation. We finally conclude the paper with Section V.

II. Background

Safety critical real time embedded systems need not only be functionally correct but also meet strict timing deadlines. For this purpose, it is necessary to calculate the WCET of these tasks. However, calculation of WCET is not straight forward for modern processors due to features such as multi-level caches and out of order execution.

There are two methods of calculating WCET, measurement based and static methods as shown in Figure 1. In measurement based methods, we test the execution time by giving different inputs. However, it is often very difficult to test a program with all different inputs. Not checking the program with every possible input might give an underestimated WCET, as shown in Figure 2. A more appropriate approach is to use static methods, which can be classified into static-analysis and model-checking based. In static analysis, rather than running the program with different inputs, all possible paths are statically checked for calculating WCET. In static analysis, often abstract interpretation [13] is used to model the architectural features of a processor using an approximated model. On the other hand, with model checking, one can write code for a precise model of the processor. The result is a tighter WCET, but using more computational overhead.

In this section, we first describe the problem of WCET calculation (Section II-A), and then the description of deterministic execution that can be used to reduce the number of states during model checking (Section II-B).

A. WCET calculation

Modern processors have features such as cache hierarchies and out of order execution, which are meant to improve the average-case execution time of programs running on them. However, these features make it much more difficult to determine a tight WCET. In addition, more complex architectures mean more states for a model checker to keep track of, making it more prone to state explosion problems. Despite these problems, there exist sophisticated tools, such as Chronos [14], that can guess a good WCET for programs running on single core processors. Multicore systems on the other hand have an additional complexity, due to shared resources, such as shared memories. With shared memory, tasks running on different cores also need to synchronize to access the shared data, for example by using locks. This makes it difficult to deduce tight WCET bounds for such systems. Synchronization of shared memory accesses also means that many different possible interleavings of the threads are possible, which further aggravates the problem of calculating the WCET. They can have timing anomalies due to shared resources and shared memory accesses. For example, assume that a path ABD is the worst-case path if seen separately, where A, B and D are basic blocks. In the presence of shared L2 cache however, another path, say ACD might become the 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.

Recently, there have been several papers published which deal with calculating WCET on multicore processors. A survey of those techniques is given in [12]. Some of those assume that there are no shared memory accesses by the tasks running on the different cores. In other words, they assume that tasks are running embarrassingly parallel to each other. They only cater for the problem of shared L2 cache accesses [10] [11] and the shared bus [5]. Papers like [15] and [16] do consider shared memory synchronization, but they assume simpler processor architectures which do not have any cache, but only scratchpad memories. Such kind of processors are not mainstream and require special programming techniques, since the scratchpad memories have to be manually managed by the programmer.

Table I. Comparison of different tools

Tool	Method Used	L2 cache	Shared data	L1 CC
[10][11]	Static analysis	+	-	-
[5]	Static analysis & Model checking	+	-	-
[15][16]	Static analysis	-	+	-
[9]	Model checking	+	-	-
[6], This	Model checking	+	+	+

[6] considers both cache coherence as well as synchronization operations such as spin locks for shared memory accesses. The authors use UPAAL [3] based model checking for that purpose. They do take into account shared memory accesses, but their solution suffers from state explosion problem even for very simple programs. [9] also uses model checking but do not support synchronization operations. [17] recently proposed a mathematical model to determine WCET of multicore systems with caches and cache coherence using abstract interpretation. However, they still do not consider cache coherence that is generated due to accessing the shared synchronizing objects. Moreover, they do not perform any evaluation.

All the tools described above are shown in Table I, along with the methods they use and the platforms they are made for. There is only one tool [6] which considers shared data on systems with L1 cache coherence (L1 CC). However, as explained before, it is too slow as it suffers from state explosion problems even for very small programs. Our tool improves upon that.

In this paper, we investigate whether model checking overhead used for calculating WCET can be reduced using deterministic shared memory accesses [7] [8]. We use the SPIN model checker [1] with its associated language PROMELA for model checking. Moreover, unlike [6], which uses a very simple example, we use real benchmark programs written in C. We use Chronos [14] to compile those programs into assembly and also to construct the control flow graph (CFG). The assembly code and CFG are then used to generate the PROMELA code for model checking to calculate WCET.

B. Deterministic execution

Multithreaded programs have a frequent source of non-determinism in the form of shared memory accesses. Due to this, multithreaded programs can have many possible thread interleavings for shared memory accesses, which makes it difficult to find WCET of such programs. We can remove interleavings for shared memory accesses altogether if we know the input of the program and perform deterministic execution, as deterministic execution would make sure that the threads perform the shared memory accesses always in the same sequence. Even for multiple inputs, we can still reduce the possibilities as explained by [4].

One such algorithm for deterministic execution is Kendo [7], which uses logical clocks for each thread to determine when a thread will acquire a lock. The thread with the least logical clock value gets the lock. For example, Thread 1 will be unable to acquire a lock when its logical clock (1029) is higher than that of Thread 2 (329). But, as soon as Thread 2's clock get past 1029, Thread 1 may acquire the lock. With DetLock [8], we showed that updating clocks ahead of time

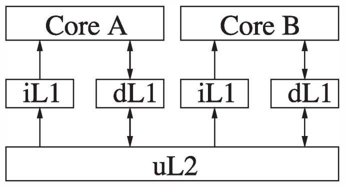

Figure 3. Architecture of the processor used

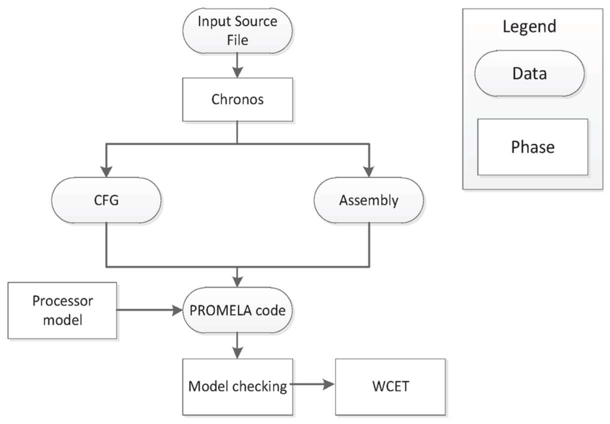

Figure 4. Steps for WCET calculation

improves the performance as compared to Kendo. Therefore, in this paper we also update the clocks ahead of time.

III. Implementation

In this section, we discuss the implementation of our tool. Firstly, in Section III-A, we discuss the architecture of the processor used. Next, in Section III-B, we discuss our method of finding WCET using deterministic execution, while in Section III-C, we discuss an optimization applied to reduce the calculated WCET with deterministic execution. Finally in section III-D, we describe our method of model checking which avoids performing binary search to calculate WCET, as done by other state of the art approaches.

A. Processor architecture

Since, the focus of this paper is to see how much reduction in analysis time we get by using deterministic execution, we assume a simple processor model, which is that every instruction takes one cycle and an L1 cache miss takes 10 cycles, while an L2 cache miss takes 80 cycles. Moreover, a taken branch causes extra 3 cycles. The architecture of the processor is shown in Figure 3. There are separate L1 caches for instruction and data, while the L2 cache is shared. We also assume that there are as many read ports for instruction cache as the number of cores. For cache coherence, we use the MESI cache coherence protocol. We also assume that every shared memory access takes place within a spinlock. For checking whether a certain memory access can cause a cache miss, we check the memory addresses. We check for shared L2 cache misses only for instructions, while assuming all the data is already there in the L2 cache. This assumption is not unreasonable since a typical L2 cache can be large enough to accommodate data for one loop cycle of a real time

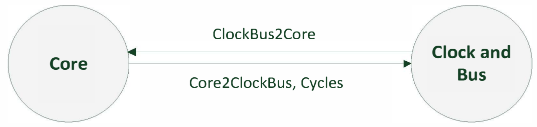

Figure 5. Block diagram of communication between a core's process and the *Clock and Bus* process

process. Similarly, for the non-shared data, we assume it has already been brought to the L1 caches of the cores, since the benchmarks we used are small enough to accommodate the local data in the L1 caches of the cores.

B. WCET calculation

As shown in Figure 4, we used the Chronos tool to extract the CFG and assembly code. This CFG and assembly code is then used to generate part of the code for our PROMELA code used to calculate the WCET.

We have two kinds of processes in our model checker. There is a core process for each of the cores while there is a *Clock and Bus* process that represents the processor's clock and the shared bus which manages cache coherency. Figure 5 shows the communication channels between the *Clock and Bus* process and a core's process. Through the *ClockBus2Core* channel, the *Clock and Bus* process tells a core to either go ahead or wait. A core's process on the other hand sends the number of clock cycles it needs to advance to the *Clock and Bus* process along with other information on the *Core2ClockBus* channel, such as the address of a shared memory access. Note that the only types of shared memory accesses we allow in our model are the shared mutexes and shared variables within locks.

```
atomic {
min_clock = get_minimum_clock();
for( pid = 0; pid < NUM_OF_PROC; i++ ) {
    if( clock[pid] == min_clock )
        advance( pid );
}}
```

Listing 1. Pseudo-code to advance clock cycles

```
atomic {
line = get_cache_line_of_addr(addr);
if (!in_l2_cache[line]) {
    in_l2_cache = true;
    st[line] = clock[pid] + l2_mt;
    wait_for_cycles( pid, l2_mt + l1_mt );
}
else if ( clock[pid] <= st[line] ) {
    to_wait = st[line] - clock[pid] + l1_mt;
    wait_for_cycles( pid, to_wait );
}
else
    wait_for_cycles( pid, l1_mt );
}
```

Listing 2. Pseudo-code to access L2 cache for instructions

Since a model in which we explicitly synchronize each thread at each clock cycle is quite costly, we have devised a method to significantly reduce the overhead without introducing errors. The C-style pseudo-code for that purpose, which is part of the *Clock and Bus* process, is shown in Listing 1. Only the cores with the minimum clocks are allowed to progress, while those which have advanced ahead have

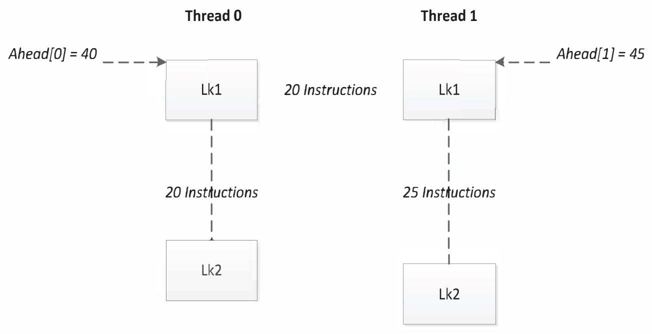

Figure 6. Optimization to improve deterministic execution, WCET and WCET calculation time

to wait. In this way, we make sure that the clocks of the cores are synchronized and yet avoid the overhead of explicit synchronization. In case of accessing instructions from the L2 cache, the C-style pseudo-code to make sure the cores progress properly, is shown in Listing 2. This code is part of a core's process. For example, if a core A experiences L2 cache miss for a cache line, the next core B reading the same cache line would read it from the L2 cache, as the core A would have already brought it into the L2 cache. However, since core B would access that cache line in a later time, to make sure this is properly modeled, we save that value in *st* (line 5), and the clock of core B is advanced using that value if its clock was less than that of core A (line 8). Here *l1_mt* is the L1 cache miss penalty while *l2_mt* is the L2 cache miss penalty. We also use a simplified cache coherence model for shared data which is read/modified only within locks, since only one core would be reading/modifying that data.

We check the WCET, both with and without deterministic execution. For deterministic execution, we used a hardware based comparator, as a totally software based deterministic execution would cause substantial cache coherence activity due to reading the shared clocks for deterministic execution. Each core writes to one of the input registers of the comparator. The comparator writes 1 to the output register whose corresponding input register contains the smallest clock value, while writing 0 to all the other output registers. In this way, a thread can know, whether to acquire a lock by just reading the value of its corresponding output register. In case of two or more input registers having the smallest value, 1 is written to the corresponding output register of the one with the least index. Through this hardware, there is no need for threads to read other cores' clocks and also no overhead is incurred due to cache invalidations that occur for maintaining cache coherence.

Since for deterministic execution, we assume a hardware based mechanism, to have a fairer comparison between the deterministic and non-deterministic methods, we also compare the deterministic execution with a method where we assume a hardware based synchronization mechanism, that is, where a lock could be acquired immediately, that is without the overhead of cache coherency for compare and swap operation on a shared variable.

C. Optimization of deterministic execution

We use the DetLock mechanism of updating clock for deterministic execution. One limitation of that method is that a thread cannot update its clock ahead of time if its waiting for a lock, even when the lock it is waiting for is different from the locks other threads are trying to acquire. For a program with two threads, Thread 0 can acquire a lock only when its clock is less than or equal to that of Thread 1 as shown by the condition below, where *dt[0]* and *dt[1]* are logical clocks for Thread 0 and Thread 1 respectively.

$$dt[0] \leq dt[1]$$

To overcome the above mentioned limitation, besides the logical clock, we introduce two more variables for each thread. These two variables are *ahead* and *nl*, where *ahead* is used by a thread waiting for a lock to tell other threads, that its not going to acquire a subsequent lock, which is different from the one it is waiting for, at least for the amount of instructions assigned to *ahead*. On the other hand, *nl* is the number of the lock, or more precisely the address of the mutex in question. The formula for lock acquisition for Thread 0, now changes to the following.

$$dt[0] \leq (dt[1] + ahead[1] \times (nl[1] \neq nl[0]))$$

This mechanism can be more easily understood by Figure 6. Here if Thread 0 has reached the place where it is trying to acquire Lk2. With the DetLock only approach, it would not be able to acquire that lock, until Thread 1 has unlocked Lk1. However, with this new optimization, it would be able to acquire Lk2 even if Thread 1 has not acquired lock Lk1 yet. Basically, when Thread 1 would reach the point where it is about to acquire Lk1, Thread 0 would know that Thread 1 is not going to acquire Lk2 until it would have executed more instructions than what Thread 0 has executed up till now. With this optimization, we can reduce the calculated WCET, albeit at the cost of slightly increased WCET calculation time.

D. Avoiding binary search

Approaches using model checking to calculate the WCET, such as [9] and [6] use assertions to find the WCET. So, they have to run the model checker several times in a binary mode fashion to reach the right WCET value. Although [6] talks about using the sup operator of UPPAAL to avoid binary search, the current stable release of UPPAAL, which is version 4.0.13, does not support the sup operator. Only versions 4.1 and greater of UPPAAL support the sup operator. That is why, the authors of [6] discuss the sup operator only in the future work section of their paper. On the other hand, our technique of avoiding binary search works perfectly on the stable release version of SPIN.

We avoid performing binary search by logging the value of the elapsed time instead. We make use of the VAR_RANGES flag in SPIN to log the ranges of the variables. However, VAR_RANGES only give ranges from 0-255. Since, SPIN generates C files which are further compiled to make the executable model checking file, it is possible to modify the C code to log the full integer value of the elapsed time. We have written a script that does exactly that by inserting code in the *logval* function to log the value of the elapsed time, the maximum value of which is taken as the WCET. Running

Table II. Benchmark characteristics

Benchmark	Basic blocks	Cond branches	Locks	Max L2 cache misses
Fluidanimate (ComputeForcesMT)	11	2	2	12
Network (thread_ippktcheck)	20	11	2	16
Radiosity (radiosity_averaging)	6	3	3	12

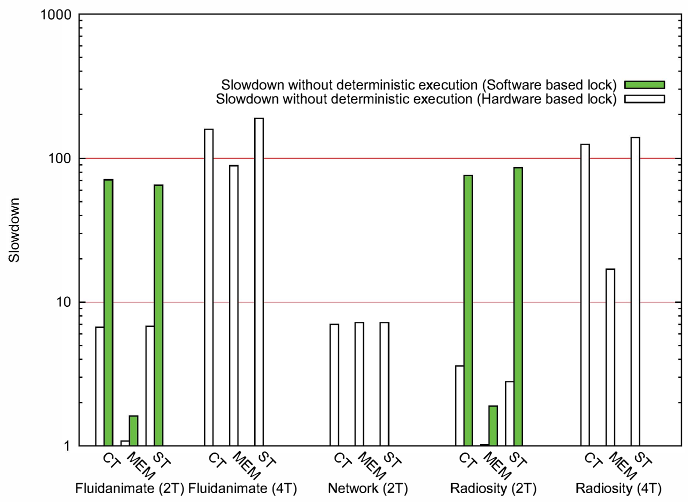

Figure 7. Slowdown with non-deterministic execution w.r.t deterministic execution (Panels 2 & 3 in Table III)

the model checker with VAR_RANGES flag increases the calculation time, but is still a much faster method than running several iterations to reach the WCET value through binary search.

IV. Performance Evaluation

In this section, we will discuss the results that we achieved by applying optimizations and using deterministic execution. Section IV-A discuss the results, while Section IV-B shows further improvement achieved by avoiding binary search to reach the WCET value.

A. Results

We selected three benchmarks. One is Fluidanimate from the PARSEC [18] benchmarks suite, another one is a Network protocol benchmark from EEMBC [2] and lastly we have Radiosity from SPLASH [19]. We only used a portion of these applications. Those portions included shared memory accesses. The characteristics of those parts of the code are shown in Table II. The names of the functions from which the code is taken are also shown. To run these benchmarks, we used a computer with 96GB RAM. We used the DCOLLAPSE flag of PROMELA for compressing memory. The results are shown in Table III.

Using only the approach of [6] and without applying the optimizations discussed in Section III-B, none of the benchmarks could complete, due to the state explosion problem. With our optimizations, most of the configurations could finish. Those configurations that still could not finish are indicated with a ∞ mark in Table III.

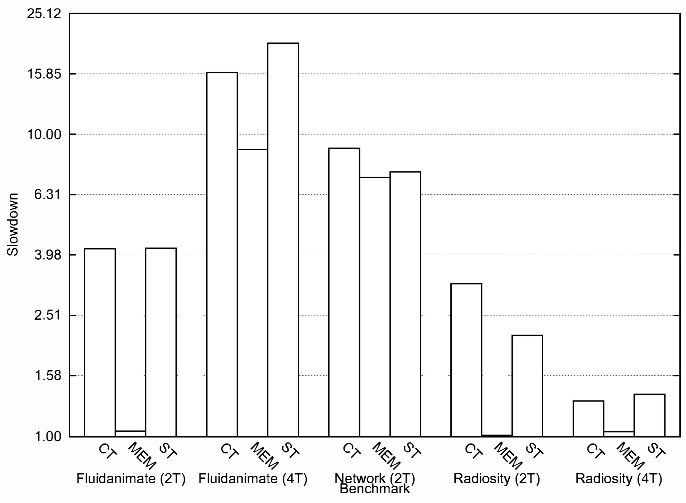

Figure 8. Slowdown of updating clocks after execution (Panel 4 in Table III)

The first panel in the Table III shows the results with deterministic execution with clocks updated ahead of time. The second panel shows the results with non-deterministic execution with hardware based lock acquisition, which do not require cache coherence for the shared mutexes. Adding hardware-based lock acquisition is done to have a fair comparison with the deterministic case, because we used hardware based deterministic execution to avoid excessive cache coherency that comes with software based deterministic execution. The third panel shows non-deterministic execution with normal software based lock acquisition that does involve cache coherence for the shared mutexes. Lastly, the fourth panel shows the results with deterministic execution that does not update the clock ahead of time but after execution, like Kendo.

From the Table III, we can see that deterministic execution with optimized clock updates (clocks updated ahead of time) gives the best results in terms of calculation time, memory consumed and number of states stored. Introducing deterministic execution does however increase the WCET slightly due to the extra code included in the programs. The increase in WCET however is not more than 4% for the selected benchmarks. The improvement in calculation time, memory consumed and number of states stored scales with the number of threads used. For example, for the Fluidanimate benchmark, the calculation time, memory consumed and number of states were reduced by as much as 158x, 89x and 188x respectively for 4 threads. From Panel 3 of Table III, we can see that the lack of hardware support causes cache coherency for shared mutexes to significantly increase calculation time, memory and states. The comparison of non-deterministic execution with respect to the deterministic version is also illustrated in Figure 7, where the bars on the left (in white color) show the overhead for non-deterministic execution with hardware support while the colored bars on the right show the overhead without hardware support. In cases where the later could not complete, we leave that column empty, that is, no bar is drawn. In that figure, CT represents calculation time, MEM represents memory consumed and ST represents the number of states used by the model checker.

Our method of updating clocks ahead of time also shows to significantly improve both the WCET and calculation time, as

Table III. PERFORMANCE RESULTS

Configuration	Param/BM	Fluidanimate (ComputeForcesMT)		Network (thread_ippktcheck)	Radiosity (radiosity_averaging)	
Number of cores		**2**	**4**	**2**	**2**	**4**
1. Deterministic with optimized clock updates and hardware support	*WCET (cycles)*	1209	1209	1276	1378	1506
	Calculation time (secs)	0.11	25.1	1050	0.1	3.42
	Memory consumed (MB)	179.8	410.6	11414	178.8	201.3
	States (millions)	0.031	2.9	218.7	0.019	0.34
2. Non-deterministic with hardware support	*WCET (cycles)*	1200 **(0.99x)**	1200 **(0.99x)**	1257 **(0.99x)**	1346 **(0.98x)**	1442 **(0.96x)**
	Calculation time (secs)	0.74 **(6.7x)**	3980 **(158x)**	7300 **(7x)**	0.36 **(3.6x)**	428 **(125x)**
	Memory consumed (MB)	194.2 **(1.08x)**	36595.4 **(89x)**	81919.9 **(7.2x)**	182.3 **(1.02x)**	3445.1 **(17x)**
	States (millions)	0.21 **(6.8x)**	544.4 **(188x)**	1576.6 **(7.2x)**	0.053 **(2.79x)**	46.96 **(138x)**
3. Non-deterministic without hardware support	*WCET (cycles)*	1254 **(1.04x)**	∞	∞	1444 **(1.05x)**	∞
	Calculation time (secs)	7.78 **(71x)**	∞	∞	7.6 **(76x)**	∞
	Memory consumed (MB)	289.5 **(1.61x)**	∞	∞	337.8 **(1.89x)**	∞
	States (millions)	2.03 **(65x)**	∞	∞	1.63 **(86x)**	∞
4. Deterministic without optimized clock updates and with hardware support	*WCET (cycles)*	1224 **(1.01x)**	1364 **(1.13x)**	1297 **(1.02x)**	1509 **(1.1x)**	1889 **(1.25x)**
	Calculation time (secs)	0.46 **(4.18x)**	414 **(16x)**	9450 **(9x)**	0.32 **(3.2x)**	4.49 **(1.31x)**
	Memory consumed (MB)	187.8 **(1.04x)**	3653.3 **(8.9x)**	81919.9 **(7.2x)**	180.8 **(1.01x)**	208.9 **(1.04x)**
	States (millions)	0.13 **(4.19x)**	57.8 **(20x)**	1645.37 **(7.5x)**	0.041 **(2.16x)**	0.47 **(1.38x)**

compared to updating clocks after execution. The improvement in WCET happens due to the fact that by updating clocks ahead of time, we reduce the waiting time of a thread waiting for a lock. That waiting also increases the possible number of states, thus increasing the calculation time and memory consumed. The slowdown caused by updating clocks after execution is illustrated in Figure 8.

In Table IV, we discuss the improvement in WCET that we observed by applying the optimization that we discussed in Section III-C, that is, by using the *ahead* and *nl* variables to allow a thread to proceed with lock acquisition even when another thread is waiting for a lock but has a lesser value of logical clock. We show the numbers for the Radiosity benchmark for both 2 and 4 threads. The other two benchmarks do not have different mutexes, so we could not apply this optimization to them. In the *W/o opt* column, we use the basic DetLock mechanism of updating the clock ahead of time, while in *With opt*, we also use *ahead* and *nl* variables. In the *With opt* column, we also show improvement (>1x) or degradation (<1x) for all the parameters as compared to the *W/o opt* column. From the table, for 4 threads, we can see improvement in WCET at the cost of increased number of states, memory consumption and calculation time. However, these numbers are still much better than the non-deterministic case (see Table III).

B. Improvement by avoiding binary search

In Section III-D, we discussed how we can avoid binary search by modifying the C code generated by SPIN to include the code to log the elapsed time value. This method avoids performing binary search as done by the state of the art approaches that use model checking to calculate WCET, such as [9] and [6]. Table V shows the overall speedup for 4 threads, including that which comes from avoiding the binary search. The column titled *W/o VR* shows the calculation time without using VAR_RANGES (See Section III-D to see discussion about the VAR_RANGES flag), while the *With VR* column shows the calculation time by using it. Next we show the speedup that we achieved with deterministic execution (DE) with clocks updated ahead of time, followed by the number of iterations used to reach WCET if binary search (BS) is used.

Table IV. IMPROVEMENT BY UPDATING CLOCK AHEAD OF A LOCK ACQUISITION FOR RADIOSITY BENCHMARK

Parameters	**2 threads**		**4 threads**	
Configuration	*W/o opt*	*With opt*	*W/o opt*	*With opt*
WCET (cycles)	1440	1378 **(1.04x)**	1728	1516 **(1.15x)**
Calc time (secs)	0.17	0.1 **(1.7x)**	3.33	3.42 **(0.97x)**
Mem consumed (MB)	179.0	178.8 **(1x)**	192.28	201.3 **(0.96x)**
States (millions)	0.021	0.019 **(1.11x)**	0.21	0.34 **(0.62x)**

Table V. IMPROVEMENT BY AVOIDING BINARY SEARCH (FOR 4 THREADS)

Benchmark	W/o VR (secs)	With VR (secs)	DE speedup	BS iterations	Overall speedup
Fluidanimate	18.7	25.1	158x	21	**2471x**
Radiosity	2.70	3.42	125x	18	**1775x**

The overall speedup is then calculated by using the following formula.

$$(Without\_VR/With\_VR) \times DE\_speedup \times BS\_iters$$

From the table, we can see that for the Fluidanimate benchmark, the overall speedup is as high as 2471x.

V. CONCLUSIONS

In this paper, we used model checking for estimating the WCET for portions of the applications where shared memory accesses occurred. We showed that by using deterministic execution, we can reduce calculation time and memory usage significantly at the cost of negligible increase of the calculated WCET. We significantly reduced the time (up to 158x), memory (up to 89x) and states (up to 188x) for calculating WCET with a negligible increase (up to 4%) in the calculated WCET for a multicore system with 4 threads. We also showed an improvement in all the parameters, if we update the deterministic execution clock ahead of time, as in the case of DetLock. Moreover, we avoid performing binary search to calculate the WCET, which involves running several iterations of the model checker, by modifying the C code generated by SPIN to log the value of the elapsed time instead. The total combined gain in speed was found to be as high as 2471x. The state explosion problem still poses a challenge to this solution for practical purposes though. Future work will focus on an approach which combines static analysis with model-checking might be used to overcome that problem.

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://spinroot.com/spin/whatispin.html

[2] http://www.eembc.org/

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

[4] 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.

[5] Sudipta Chattopadhyay, Abhik Roychoudhury, and Tulika Mitra. Modeling shared cache and bus in multi-cores for timing analysis. In *SCOPES*, 2010.

[6] 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*, 2010.

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

[8] Hamid Mushtaq, Zaid Al-Ars and Koen 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.

[9] 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.

[10] Jun Yan and Wei Zhang. Wcet analysis for multi-core processors with shared l2 instruction caches. In *RTAS*,2008.

[11] Wei Zhang and Jun Yan. Accurately estimating worst-case execution time for multi-core processors with shared direct-mapped instruction caches. In *RTCSA*, 2009.

[12] Hamid Mushtaq, Zaid Al-Ars and Koen Bertels. Accurate and efficient identification of worst-case execution time for multicore processors: A survey. In *IDT 2013*.

[13] S. Bygde. Static WCET Analysis Based on Abstract Interpretation and Counting of Elements. Lic. dissertation, School of Innovation, Design and Engineering, March 2010.

[14] Xianfeng Li, Yun Liang, Tulika Mitra, Abhik Roychoudhury. Chronos: A Timing Analyzer for Embedded Software. Science of Computer Programming, Special issue on Experimental Software and Toolkit, 69(1-3), December 2007.

[15] Haluk Ozaktas, Christine Rochange, and Pascal Sainrat. Automatic WCET Analysis of Real-Time Parallel. In *13th International Workshop on Worst-Case Execution Time Analysis*, 2013.

[16] Mike Gerdes, Theo Ungerer and Rudolf Knorr. Timing Analysable Synchronisation Techniques for Parallel Programs on Embedded Multi-Cores. Phd Thesis.

[17] Sudipta Chattopadhyay. Time-predictable Execution of Embedded Software on Multi-core Platforms. Phd Thesis.

[18] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: characterization and architectural implications. In *PACT*, 2008.

[19] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA*, 1995.