Combining Abstract Interpretation with Model Checking for Timing Analysis of Multicore Software

Mingsong Lv[1], Nan Guan[2], Wang Yi[2], Ge Yu[1]
[1]*Northeastern University, Shenyang, China*
[2]*Uppsala University, Uppsala, Sweden*

***Abstract*—It is predicted that multicores will be increasingly used in future embedded real-time systems for high performance and low energy consumption. The major obstacle is that we may not predict and provide any guarantee on real-time properties of software on such platforms. The shared memory bus is among the most critical resources, which severely degrade the timing predictability of multicore software due to the access contention between cores. In this paper, we study a multicore architecture where each core has a local L1 cache and all cores use a shared bus to access the off-chip memory. We use Abstract Interpretation (AI) to analyze the local cache behavior of a program running on a dedicated core. Based on the cache analysis, we construct a Timed Automaton (TA) to model the precise timing information of the program on when to access the memory bus (i.e. when a cache miss occurs). Then we model the shared bus also using timed automata. The TA models for the bus and programs running on separated cores will be explored using the UPPAAL model checker to find the WECTs for the respective programs. Based on the presented techniques, we have developed a tool for multicore timing analysis, which allows automatic generation of the TA models from binary code and WCET estimation for any given TA model of the shared bus. Extensive experiments have been conducted, showing that the combined approach can significantly tighten the estimations. As examples, we have studied the TDMA and FCFS buses. In both cases, the WCET bounds can be tightened by up to 240% and 82% respectively, compared with the worst-case bounds estimated based on cache misses and maximal delays for bus access.**

***Keywords*-abstract interpretation, model checking, WCET, multicore, shared bus**

I. Introduction

It is predicted that multicores will be increasingly used in future embedded real-time systems for high performance and low energy consumption. The major obstacle is that we may not predict and provide any guarantee on real-time properties of software on such platforms. The shared memory bus is among the most critical resources, which severely degrade the timing predictability of multicore software due to access contention between cores. In single-core WCET analysis, it is usually assumed that it takes constant time to access the main memory (given perfect memory controller). But for multicores, this assumption never holds, since memory accesses suffer extra delays as a result of access conflicts on the shared memory bus. For example, for a duo-core processor with first-come-first-serve (FCFS) bus arbitration, the situation could be any memory access request is delayed by a request just issued from the other core. In this case, the worst-case memory access time will be doubled compared to that without bus contention. Since the number of processor cores on a chip continues to increase, the amount of traffic on the shared memory bus increases accordingly, and this problem is expected to be even worse in the future [1]. Since the time to access main memory is much larger than either cache access time or instruction execution time, assuming worst-case delay will leads to very pessimistic estimations. So techniques that can precisely bound the delays on shared buses are very important to obtain useful timing guarantees for multicore real-time systems.

In this paper, we study a multicore architecture where each core has a local L1 cache and all cores shared the memory bus. We intend to find the timing bounds of the programs considering variable memory access delays due to both cache misses and memory bus contention. In order to precisely estimate those delays, one has to analyze how programs' behaviors affect the timing of the conflicts, which is not a trivial task.

We present an approach which combines abstract interpretation and model checking to solve this problem. We use abstract interpretation to analyze the local cache behavior of a program running on a dedicated core. Based on the cache analysis, we construct a timed automaton to model the precise timing information of the program on when to access the memory bus. Then we model the shared bus also using timed automata. The TA models for the bus and programs running on separated cores will be explored using the UPPAAL model checker [2] to find the WECTs for the respective programs.

Based on the presented techniques, we have developed a tool for multicore timing analysis, which allows automatic generation of the TA models from binary code and WCET estimation for any given TA model of the shared bus. Extensive experiments have been conducted, showing that the combined approach can significantly tighten the estimations. In general, we can use this approach to model a broad range of shared buses. As examples, we have studied the TDMA and FCFS buses. In both cases, the WCET bounds can be

Corresponding author: Wang Yi. E-mail address: yi@it.uu.se; Post: Box 337,751 05, UPPSALA; Tel: +46(0)18 4713110.

tightened by up to 240% and 82% compared with the worst-case bounds estimated based on cache misses and maximal delays for bus access.

The rest of this paper is organized as follows. In Section II, we give an overview of WCET analysis with an emphasis on bus analysis for multicores. The system model and the assumptions are detailed in Section III. The overall analysis framework and its main components are presented in Section IV. Section V introduces the WCET analysis tool and presents experimental results and evaluation. The paper is concluded in Section VI.

II. Related Work

To obtain the WCET of a program, one first reconstruct the control flow graph (CFG) from the program's binary [3]. The nodes of a CFG are called basic blocks, with each one containing a number of instructions executed sequentially. In the second step, the WCET of the basic blocks are estimated by considering the effects of micro-architecture features, such as caches [4], pipelines [5], etc. After that, one can use techniques such as integer linear programming (ILP) [6] or model checking to find the path that leads to the maximum execution time. Micro-architecture analysis may have a huge impact on the analysis precision.

WCET analysis of multicores is challenging due to the difficulty in bounding interferences on shared resources. Yan and Zhang analyzed direct-mapped shared instruction caches in multicores [7]. A first pass applies abstract interpretation to obtain the cache hit/miss classification (CHMC) assuming private use of shared cache, and a second pass analyzes the effects of inter-core cache conflicts. The work was later improved by discovering the timing order of the potential inter-core conflicts using Cache Conflict Graphs [8]. Li et al. proposed a method to estimate the worst case response time (WCRT) of concurrent programs running on multicores with shared L2 caches [9]. They proposed an iterative analysis where the lifetime of the tasks is explored and the BCETs and WCETs are tightened in each iteration. The work was later extended by adding an AI-based TMDA bus analysis technique to bound the memory access delay[10]. Hardy et al. proposed a method for timing analysis of shared instruction caches of multi-cores and a selective caching technique (called bypass) to reduce inter-core interferences and also to tighten the WCET estimations [11].

Bare analysis of shared caches is very difficult, but this problem is made easier if the shared cache is intelligently partitioned among cores [12], [13] (cache partitioning), or one selectively cache the program and temporarily disable cache replacement [13] (cache locking). The purpose of such techniques is to completely avoid conflicts on shared resources to ensure predictable timing behaviors.

Besides shared caches, bounding the time for shared bus access is another challenging task. TDMA buses are one of the focuses. Andrei et al. proposed an approach for WCET analysis of programs on multiprocessors considering variable memory access time due to bus contention assuming TDMA bus arbitration [14]. Their approach explicitly enumerates all feasible paths of a task and the WCRT is chosen as the maximum among the traces' execution time. Based on Andrei's work, Rosén et al. proposed algorithms for optimization of bus schedules to reduce schedule length [15]. Schranzhofer et al. proposed a method to analyze the worst-case response time of real-time tasks on systems with shared resources and TDMA arbitration policies [16]. The analysis method suffers some very restrictive assumptions: first, the fully timing compositional architecture where execution time and communication time can be decoupled is assumed; second, tasks are specified by superblocks that execute in some statically pre-defined order. These assumptions make the technique hard to be applied to COTS architectures and real-life programs with arbitrary control flows.

Other research focuses on analyzing non-TMDA buses. Staschulat et al. presented a method to compute the WCET of tasks under bus contention at the system level [17]. A traditional WCET analysis technique is extended by considering the latencies due to bus contention for each basic block. The results are pessimistic since a conservative minimum distance of memory accesses is assumed. Andersson et al. studied the problem of bounding the execution time due to contention on memory buses [18]. The key point of their work is finding an upper bound on the number of bus requests that a task may suffer. The main drawback implied in their approach is that they assume the worst-case scenario where one task has to wait for bus transactions from all the other tasks, which could lead to very pessimistic estimations. Although they do not restrict to specific arbitration policies, their work only applies to work-conserving buses.

Related work has also considered variable bus access time due to peripherals and memory controllers. Pellizzoni et al. studied the impact of peripherals' interference on WCET analysis, and presented a theoretical framework to model shared resource contention and to compute the bounds on the delay [19]. Bourgade et al. presented a technique to bound the memory latencies in WCET estimation [20]. An abstract interpretation based method is used to estimate hits or misses in the row buffer given the open page policy.

Our shared bus analysis method differs from existing work in that we adopt abstract interpretation for cache analysis, the result of which is utilized in bus analysis by model checking. By modeling both the programs and buses as timed automata, precise bus access information is preserved and the timing of bus conflicts can be precisely explored. Our framework can model a broad range of bus arbitrations.

III. System Model

In this paper, we assume a duo-core processor with private L1 cache for each core and a shared memory bus

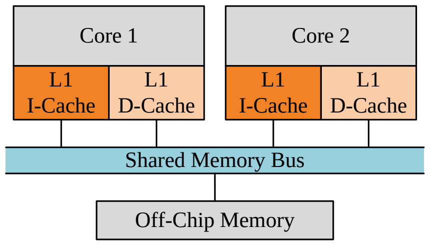


Figure 1. The system architecture

for all cores, as depicted in Figure 1. Some architectural assumptions made in our analysis are as follows:

- *A1*: We assume set-associative caches with least-recently-used (LRU) replacement policy, and the cache hierarchy is non-inclusive.
- *A2*: We do not consider cache and memory access for data, but our framework can be extended to handle data caches.
- *A3*: We assume a perfect in-order pipeline, so the micro-architecture is assumed to be free of *timing anomalies* [21].
- *A4*: We assume it takes constant time to assert the address and to deliver data from main memory to the processor core.
- *A5*: If multiple programs are allocated on one core, they are statically scheduled according to some given order, and the programs do not share code.

In assumption A1, we choose LRU since it's the most widely used [4], [22], but other policy like FIFO can also be handled within our framework [23]. Our study mainly concentrates on instructions caches, but it can be extended to handle data caches using AI based techniques [24]. Assumption A3 is made since pipeline analysis is out of the scope of this paper. The memory controller is assumed to be perfect by providing with constant access time for any memory block. The interplay between shared buses and memory controllers is left for future work. We have assumption A5 since we do not intend to discuss cache related preemption delay (CRPD) in this paper. This assumption could be removed by integrating CRPD analysis techniques [25] in our cache analysis.

IV. Timing Analysis of Caches and Buses

In this section, we first present the main work flow of our analysis framework, depicted by Figure 2. First, we conduct private L1 cache analysis for the tasks independently using abstract interpretation to obtain the cache hit/miss classifications (CHMC). Then a TA to model the precise timing information of the program on when to access the memory bus is constructed. Given a bus configuration, we can also model the bus as a timed automaton. The WCET is obtained by using the UPPAAL model checker to explore the

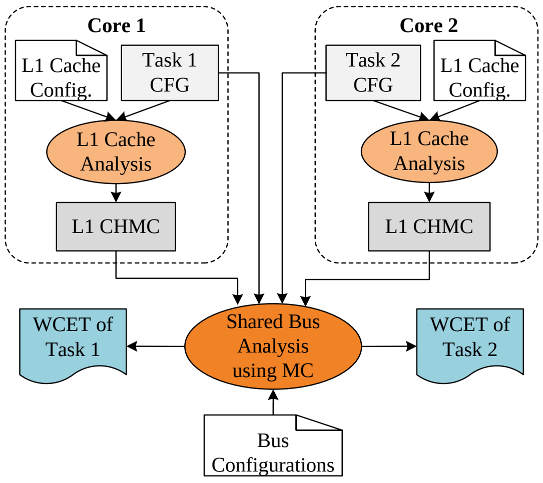


Figure 2. The overall analysis framework

TA models. In the coming sub-sections, we will present in details how the caches and buses are analyzed, respectively.

A. Private Cache Analysis by Abstract Interpretation

The private cache analysis employs abstract interpretation presented in [4], which works on abstract cache states (ACS). Three independent analysis are performed to predict cache hits or misses.

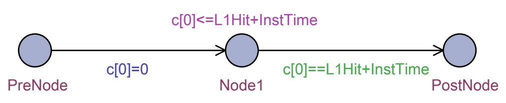
- *Must Analysis*: To determine whether a memory block is definitely in the cache. This analysis maintains in the ACS the upper bound on the ages of a memory block.
- *May Analysis*: To determine whether a memory block is never in the cache. This analysis maintains in the ACS the lower bound on the ages of a memory block.
- *Persistence Analysis*[1]: This analysis predicts whether a memory block is never evicted from the cache once loaded into the cache.

By applying the above analysis, each memory block is classified to exactly one of the following categories:

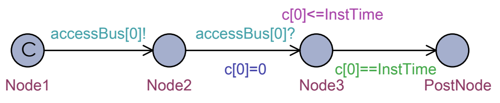
- *Always Hit (AH)*: Reference to this memory block is guaranteed to be hit in the cache.
- *Always Miss (AM)*: Reference to this memory block is guaranteed to be miss in the cache.
- *First Miss (FM)*: the memory block is guaranteed never to be evicted from the cache once it is loaded, which means all but the first cache access are hits.
- *Not Classified (NC)*: The memory block cannot be classified as either AH, AM, or FM.

One can use AI and add the worst case delay by bus to get a bound on WCET, but this will be very pessimistic for

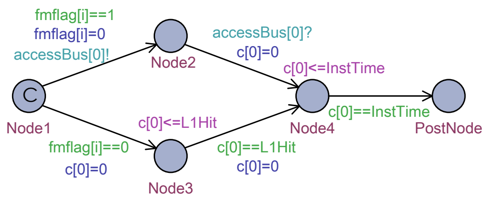
[1]In our research, we found that the original persistence analysis proposed in [4] gives unsafe results. The same problem has been identified by Cullmann from Saarland University as well, and he has proposed a correct solution for the problem (http://rw4.cs.uni-saarland.de/c̃ullmann/persistence_talk_200907.pdf). We adopted Cullmann's approach in our paper for persistence analysis.



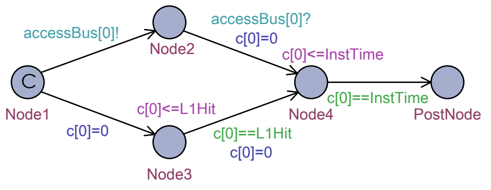
(a) AH instruction



(b) AM instruction



(c) FM instruction



(d) NC instruction

Figure 3. Modeling instructions

multicores with shared resources. Model checking can be used to improve the precision not only for the analysis of shared buses, but also shared caches. In this paper, we focus on shared bus analysis.

We refer interested readers to [4] for the details on how AI works. Here we present an example to show the result of AI analysis. Figure 4 illustrates a program with six basic blocks, numbered from BB0 to BB5. Each basic block may have several instructions annotated with cache hit/miss classifications. For instance, BB3 has two instructions that are classfied as FM and AH, respectively. We will use Figure 4 as a motivating example to show how to transform the program to a timed automaton in the next sub-section.

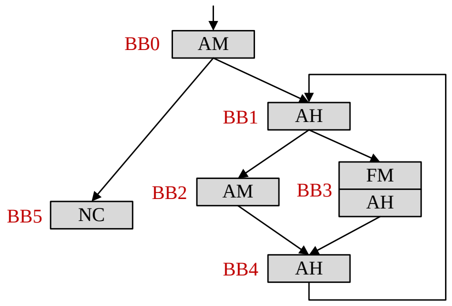


Figure 4. An exemplary program after AI analysis

B. Shared Bus Analysis By Model Checking

In multicores, memory access time may be very unpredictable due to both cache misses and contention on shared buses. A bus access is possible only when an instruction misses in the cache. When a bus request is issued to the shared bus, it is possible that the system is servicing a request issued by some other core, so this request has to wait for the completion of the current service, which results in additional delay. For TDMA buses, wait delay could happen when a bus request is issued in the time slot which is not assigned to it. To precisely estimate the memory access time, one should know the time at which the cache misses occur and how conflicts happen on shared buses. In this paper, we leverage model checking to solve the above problem. We have modeled (1) the automata for the programs that preserves exact timing of bus accesses due to cache misses; (2) the automaton for the shared bus which can help to explore the timing of contention on the bus.

1) From CFG with CHMC to Timed Automata: Programs as shown in Figure 4 is the input to the TA construction. We first build the sub-model for each basic block, then generate the TA for the program by connecting the basic blocks according to the control flow. In building the model for each basic block, instructions with different CHMC should be distinguished. Then we propose a method to group a bunch of instructions, with the objective to reduce the model size, i.e. reduce the number of locations and edges in the model and thus the state space.

Modeling AH instructions. An AH instruction is guaranteed to hit in the cache, so it never access the shared bus. For such instructions, we just model their L1 cache access time and its execution time, as shown in Figure 3(a). Here "`c`" is a set of clock variables in UPPAAL, where "`c[0]`" is used for the program running on Core-0. On the incoming edge of Node1, the assignment "`c[0]=0`" resets the clock. The invariant "`c[0]<=L1Hit+InstTime`" on Node1 and the guard "`c[0]==L1Hit+InstTime`" on its outgoing edge guarantee that the automaton will stay exactly for "`L1Hit+InstTime`" time units once it enters location Node1. Here "`L1Hit`" and "`InstTime`" refer to the delay of an L1 cache hit and the execution time of the instruction, respectively.

Modeling AM instructions. An AM instruction is guaranteed to access the shared bus. Figure 3(b) illustrates the model. Node1 is marked as "committed" meaning the automaton should leave the location immediately after it enters the location. On the transition from Node1 to Node2, a signal is sent to the bus automaton via the `accessBus[0]` channel modeling the issuing of a bus access request. The bus automaton models the time to service this request, which is detailed in later sections. Once the service is completed, a signal is sent back to the program automaton, and then the program automaton can progress. `accessBus[0]?` on the outgoing transition of Node2 specifies that the program automaton will wait in location Node2 until a signal is received from the `accessBus[0]` channel. Instruction execution time is modeled similarly to AH instructions.

Modeling FM instructions. Classification FM means that, the instruction has cache miss the first time it is referenced, and all consequent references are cache hits. So such instructions have two execution scenarios, which is distinguished by a variable named `fmflag` assigned to each FM instruction. As illustrated in Fig. 3(c), the path "Node1→Node2→Node4" models the first time to reference an FM instruction; the path "Node1→Node3→Node4" models all but the first reference to the instruction. Once the first reference is finished, `fmflag` is set to "0" and all consequent executions of this instruction definitely take the lower path.

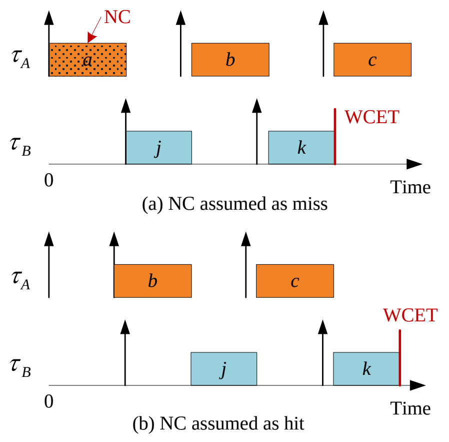


Figure 5. An example showing it is unsafe to assume NC as AM

Modeling NC instructions. An instruction is classified as NC if we cannot guarantee that the instruction is in the cache each time it is referenced. One reason why NC exists is that the cache analysis technique is not precise enough to classify an instruction which is actually AH, AM, or FM. Another reason could be the following scenario. There are two branches A and B in a loop, execution of A will evict some memory blocks of B and vise versa. In this case, the path leading to WCET takes the two branches alternatively. Some instructions on the branches may be either hit or miss in different loop iterations.

NC instructions can be safely treated as AM in WCET calculation in single-cores, but they have to be handled carefully in multicores where bus sharing is possible. It has been proved in [26] that the above assumption is safe for TDMA buses, but it is unsafe for work-conserving FCFS buses. Figure 5 illustrates an execution scenario where treating NC as hit may actually delay the other core's

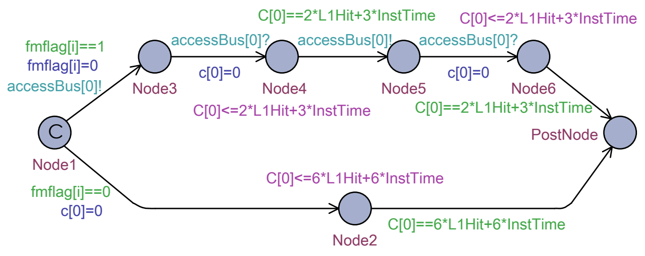


Figure 6. An example of optimization by grouping

execution. Note that the gap between the end of a bus service and the next request is constant, for example, it could be the execution of some AH instructions. So in the analysis of FCFS buses, we have to consider both possibilities of an NC instruction, as illustrated in Figure 3(d).

Optimization by grouping. To reduce the model size, i.e. reduce the number of locations and edges in the model and thus the state space, we present an optimization technique called grouping. In a basic block, there may exist multiple consecutive FM or AH instructions. In this case, the sequence of instructions can be grouped together. For instance, we have a sequence of six instructions with the CHMC pattern $< FM, AH, AH, FM, AH, AH >$. The results of grouping is shown in Figure 6. The upper path models the scenario when the sequence is executed the first time. Node3 and Node5 model the bus access delay of the two FM instructions. Node4 and Node6 model the time delay of the execution time of the FM instruction itself and the two consequent AH instructions. The lower path models all but the first iterations of the sequence, where all instructions are hits in the cache. Only 7 locations are needed to model the six instructions compared to 12 locations without grouping.

Following the above techniques, we can obtain the TA of the example of Figure 4, which is illustrated in Figure 8. Then we will present the modeling of the shared buses.

2) Shared Bus Analysis by Model Checking: In our approach, we adopt model checking for bus analysis. In general, we can use this approach to model a broad range of shared buses, but here we take TDMA and FCFS buses as examples.

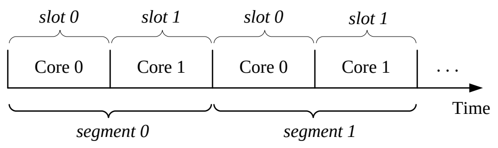


Figure 7. The TDMA bus schedule

The TDMA bus schedule defines how different processing units use the bus in a time sharing manner. A TDMA bus is composed of consecutive *segments*, which is further divided into *slots* with each slot assigned to some processing unit. Figure 7 shows the TDMA bus schedule assumed in

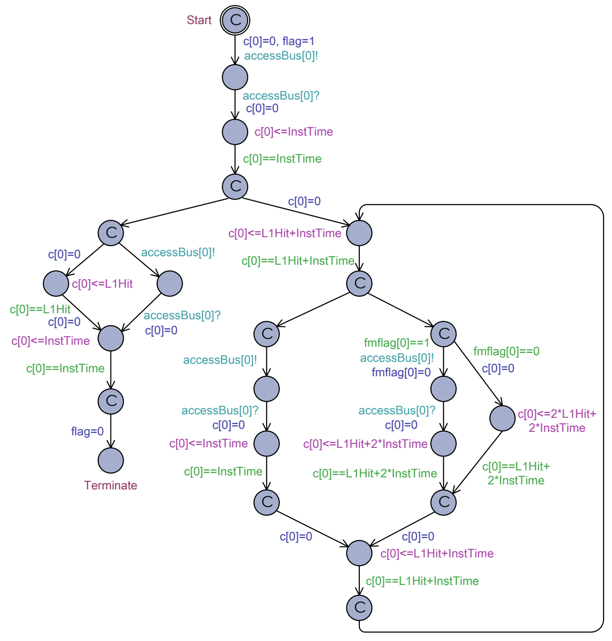


Figure 8. The TA for the motivating example

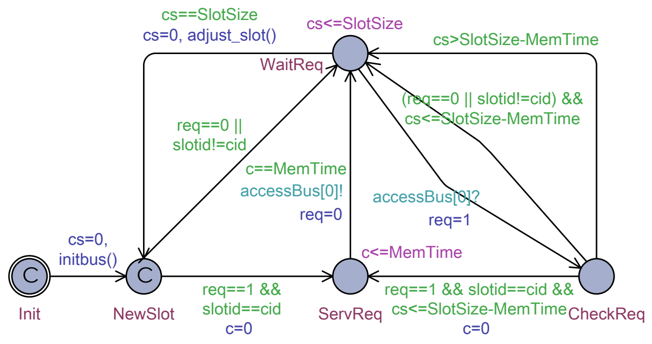


Figure 9. Modeling the TDMA bus

this paper, in which all the segments have the same size, and each segment are evenly divided into N slots (N is the number of cores). So no two cores can shared one slot.

Figure 9 is the TA for the TDMA bus. Note that since each core will be assigned a different slot, interference on the bus is isolated. In this case, the bus delay of one program only depends on itself. Given assumption A3, we can also guarantee that the bus accesses from one program will come sequentially. So the TDMA bus model only needs to incorporate the behaviors related to the program to analyze.

When the system starts, the bus automaton will enter location NewSlot. If there is no pending request or the current slot is not assigned to the program, the automaton goes to location WaitReq where the automaton waits for new requests to come. When a new request is issued, a transition from WaitReq to CheckReq is taken, and the request is recorded by setting the variable `req` to 1. The request can be served if both the following two conditions are satisfied: (1) the current slot is assigned to the core where the request comes from; (2) there is enough remaining time in the slot to service a request. If the conditions hold, the automaton goes to location ServReq, where the time to service a request is modeled using clock `c`. When the service is completed, the automaton goes back to WaitReq. If the condition is not satisfied, the automaton will directly go to WaitReq. Since the request has been recorded, it will be serviced in the next slot assigned to it. We use another clock `cs` to guarantee fixed slot size. When the time of a slot runs up, the automaton will go from WaitReq to NewSlot, and the variable `slotid` is adjusted by calling function `adjust_slot()` to represent a different slot. Note that we can reuse the clock used by the program instead of introducing clock `cs`, since at this time, the program is guaranteed not using its clock (it is waiting for the request completion signal). Reducing the number of clocks is very important to reduce the state space in model checking with time automata.

We also modeled a work-conserving non-preemptive FCFS bus, which is illustrated in Figure 10. If a bus request arrives when another request is being serviced, this request will be buffered in a queue. Note that for each core, a bus request is issued only after the previous one is serviced, so there is at most one request from each core at any time. If there are N cores in the system, it suffices to design a queue with size N. We use an array `queue[]` to model the queue, where value i ranging from 0 to $N-1$ represents a pending request from core-i, and value N is used to represent an empty position in the queue.

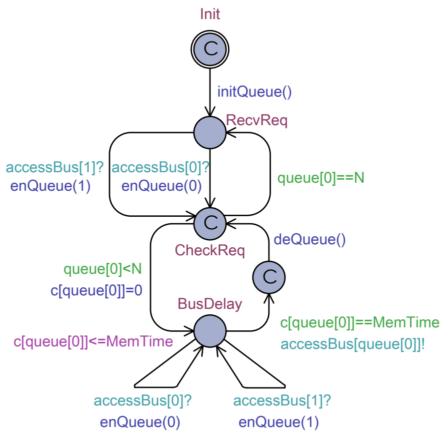


Figure 10. Modeling the FCFS bus

In the FCFS model, each transition from location RecvReq to location CheckReq enables the bus automaton to receive bus requests from one core, and the requests trigger the automaton to progress from RecvReq to CheckReq. Once a request is received, it is put in the queue immediately by executing the `enQueue()` function. Then the automaton

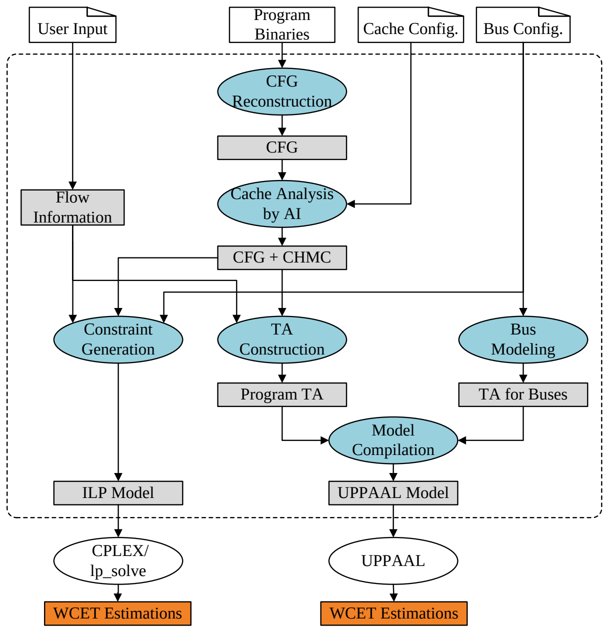


Figure 11. The WCET analysis tool for multicore software

checks whether there are pending requests in the queue: If there are one or more requests in the queue, the first request is serviced, and the automaton goes to location BusDelay to model the time to access the bus; otherwise, the automaton goes back to RecvReq to wait for future requests. These conditions are checked by evaluating `queue[0]<N` and `queue[0]==N` on the outgoing edges of CheckReq.

The BusDelay location mainly models the time to service a bus request. The cyclic transitions on location BusDelay enables receiving new requests during bus service and putting them in the queue. Once a request service is finished, the bus automaton send a signal to the requesting program to notify service completion, and then the request is removed from the queue by executing the `deQueue()` function. After that, the bus automaton goes back to CheckReq to service other pending requests. Since we assume perfect memory controller, each bus service has a constant delay of “`MemAccessTime`”.

3) Putting All Together: Now we have presented how to model the programs and the buses. The model for the programs communicate with the bus model via the `accessBus` channel. Let’s assume program P_A and P_B are assigned to core-0 and core-1. For TDMA bus, if we want to calculate the WCET of P_A, we only put the TA for P_A and the TA for TDMA bus into one UPPAAL model and let the model checker find the bound. But for FCFS bus, we have to put the TA for P_A, P_B and the FCFS bus into one UPPAAL model, since the WCET of P_A depends on the behavior of P_B as well.

V. Implementation and Evaluation

In this section, we first describe the WCET analysis tool developed in this paper; and then give the results and evaluation for the TDMA bus and the FCFS bus, respectively.

A. A WCET Tool for Analysis of Multicore Software

Based on the techniques presented in previous sections, we have developed a WCET tool for analysis of multicore software. The work flow of the tool is illustrated within the dotted box of Figure 11. The major input to the tool is programs allocated to a multicore processor. The programs are compiled to executable binaries before analysis. The first step in to reconstruct the CFGs of the programs. Then cache analysis using abstract interpretation is conducted on the programs, and the result is CFGs with cache hit/miss classifications. After that, the tool automatically constructs the TA of the programs out of the results of AI analysis, which preserves exact bus access behaviors. For a given bus configuration, we also model them as a timed automaton. Then UPPAAL is invoked to explore the TA models, and the WCET of a program is extracted from the clock constraints within the UPPAAL model checker as in the TIMES tool [27]. Our analysis tool allows for modeling a broad range of bus arbitration policies, although we only use TDMA and FCFS buses as exmaples in this paper.

B. Experimental Settings

We take six benchmark programs from the Mälardalen benchmark suite[2]. Table I lists the name, description, code size (in terms of the number of instructions) of each program. Loop bounds are set manually.

Table I
Benchmark Programs

Name	Description	#inst.
bs	Binary search algorithm for an array	78
edn	Finite Impulse Response (FIR) filter calculations	896
fdct	Fast Discrete Cosine Transform	647
insertsort	Insertion sort on a reversed array	106
jfdctint	Discrete Cosine Transformation on a pixel block	691
matmult	Matrix multiplication	287

C. Results for the TDMA bus

Since programs are statically scheduled and there is no code sharing, the WCET of the programs can be calculated independently for the TDMA bus. We conduct experiments for a duo-core system with a private L1 cache for each core. The configuration for the L1 cache is: cache size = 2KB, cache associativity = 4, cache line size = 8 bytes. L1 cache hit latency is 1 cycle, and transferring a cache line on the bus takes 40 cycles. It takes 1 cycle for each instruction to execute. We performed experiments on two different slot sizes for the TDMA bus: 100 and 200 cycles.

[2]http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

Table II
RESULTS FOR A DUO-CORE SYSTEM WITH SLOT SIZE 100

Programs	WCET		Improvement
	AI+MC	AI+Worst-Case	
bs	8,282	14,644	77%
edn	9,219,082	16,565,100	80%
fdct	268,882	479,946	78%
insertsort	21,041	29,702	41%
jfdctint	315,882	563,936	79%
matmult	151,241	174,390	15%
Average			62%

Table III
RESULTS FOR A DUO-CORE SYSTEM WITH SLOT SIZE 200

Programs	WCET		Improvement
	AI+MC	AI+Worst-Case	
bs	8,482	22,444	165%
edn	9,207,282	25,756,000	180%
fdct	267,282	742,646	178%
insertsort	21,282	40,302	89%
jfdctint	314,564	873,336	178%
matmult	150,841	203,090	35%
Average			138%

Table IV
RESULTS FOR A 4-CORE SYSTEM WITH SLOT SIZE 100

Programs	WCET		Improvement
	AI+MC	AI+Worst-Case	
bs	16,082	30,244	88%
edn	18,428,441	34,946,900	90%
fdct	529,682	1,005,350	90%
insertsort	31,641	50,902	61%
jfdctint	624,482	1,182,740	89%
matmult	179,241	231,790	29%
Average			75%

Table V
RESULTS FOR A 4-CORE SYSTEM WITH SLOT SIZE 200

Programs	WCET		Improvement
	AI+MC	AI+Worst-Case	
bs	16,082	53,644	234%
edn	18,404,164	62,519,600	240%
fdct	529,682	1,793,450	239%
insertsort	32,082	82,702	158%
jfdctint	628,164	2,110,940	236%
matmult	179,241	317,890	77%
Average			197%

For slot size 100, the worst-case bus delay happens when a bus request arrives in the slot assigned to it, but finds that there are only 39 cycles left, which is just not enough to service the request. In this case, the request has to wait for the corresponding slot in the next segment. The total delay on the bus including service time is: 39 + 100 + 40 = 179. The worst-case delay for slot size 200 is 279 accordingly. Table II and Table III list the results for the TDMA bus with slot size 100 and 200, respectively. All the WCET values are clock cycles, which applies throughout this paper. We define $improvement$ as $(WCET_{AI+WC}/WCET_{AI+MC} - 1)$, which describes how much our approach can tighten compared to assuming worst-case bus delay.

Experimental results show that the WCET bounds can be tightened by 62% and 138% in average for slot size 100 and 200 using our approach. It is can be seen that WCETs obtained by our approach for different slot sizes do not make much difference for most of the programs. If we look at the behaviors of the programs, we can find that: there is a burst of cache misses (bus accesses) when a program is being loaded into the cache; and then the program spends most of the remaining time within the loops where a large portion of cache accesses are hits. We call the time for loading a program the *burst period*. During burst periods, bus accesses come continuously, so the bus slots are busy servicing requests, leaving little time for the bus to idle. In the long run, each core are utilizing nearly half of the bus service time, and the slot size has little effect on the resulting WCET. But slot size makes great difference in the calculated worst-case bus delay. As we have shown before, the worst-case bus delay for slot size 100 is 179, the figure goes up to 279 for slot size 200, which results in very large WCET estimations. This explains why our approach has bigger improvement for larger slot size. In real systems, it is not common to configure small slot sizes, since it results in frequent slot switching and consequently more overhead. So assuming worst-case bus delay will lead to too pessimistic or even useless estimations in practice. In our approach, the UPPAAL model checker can explore the timed automata for the programs and the buses and precisely capture their behaviors, so tighter estimations are produced.

We have also conducted experiments for a 4-core system with the same cache and slot configurations as that of the duo-core system. The results are listed in Table IV and Table V. In average the WCET bounds by our approach is tighten by 75% and 197%, which is better compared to the duo-core system. The maximal observed improvement 240% comes from *edn* in the slot size 200 experiment. For the 4-core system, if a bus request misses its slot, it generally has to wait more slots. The effect is almost equivalent to configuring a large slot size for the duo-core system, the consequence of which has been explained before. So bigger improvements are observed.

Our analysis method is efficient with respect to analysis time and memory usage. Most experiments for TDMA return results within several seconds. The maximum analysis time is 738 second, which is witnessed in the experiment for *edn*. Maximum observed memory usage is 110MB. As we have discussed before, we can analyze the programs independently when analyzing the TDMA bus. So our method can scale well with the number of cores. This has been demonstrated in the experiments.

Table VI
RESULTS FOR THE FCFS BUS

Programs	WCET (AI+MC)								WCET	Max. Impr.	Avg. Impr.
	S1	S2	S3	S4	S5	S6	S7	S8	AI+Worst-Case		
bs	**3,802**	6,815	3,802	3,802	4,921	3,802	3,802	3,802	6922	**82%**	67%
edn	246,313	240,345	**240,267**	262,376	244,400	240,345	255,790	245,925	276,068	15%	12%
fdct	37,573	41,278	44,486	**37,573**	56,996	44,486	37,573	56,996	63,453	69%	46%
insertsort	14,968	**14,968**	14,968	18,238	14,968	14,968	18,055	14,968	19,208	28%	23%
jfdctint	40,153	40,153	40,153	55,161	60,025	**40,153**	48,758	59,892	67,793	69%	45%
matmult	140,504	141,348	141,360	138,406	139,561	141,469	**138,406**	139,881	145,977	5%	4%
Average improvement for all programs											33%

D. Results for the FCFS bus

To evaluate the results of the FCFS bus, we group the six benchmark programs into two task sets – $\{bs, edn, fdct\}$ and $\{insertsort, jfdctint, matmult\}$, each of which is allocated on a different core. Table VII lists the schedules used in our experiments. The tasks within the same task set execute sequentially according to the given order. We calculate the WCET of each task. The cache configurations are: cache size = 8KB, cache line size = 8 bytes, cache associativity = 4. Instruction execution time, L1 cache hit delay and transferring a cache line on the bus takes 1 cycle, 1 cycle and 40 cycles, respectively.

Table VII
STATIC SCHEDULES EXPERIMENTED

Schedules	**Core-0**	**Core-1**
S1	edn, bs, fdct	matmult, insertsort, jfdctint
S2	bs, fdct, edn	matmult, insertsort, jfdctint
S3	fdct, edn, bs	matmult, insertsort, jfdctint
S4	edn, bs, fdct	insrtosrt, jfdctint, matmult
S5	fdct, bs, edn	jfdctint, matmult, insertsort
S6	fdct, bs, edn	matmult, insertosrt, jfdctint
S7	edn, bs, fdct	jfdctint, insersort, matmult
S8	fdct, edn, bs	jfdctint, matmult, insertsort

Table VI shows the results for experiments on the FCFS bus. We list the WCET for each program in every schedule. For FCFS buses, the worst-case bus delay happens when a request req_i arrives when the bus is servicing a request from the other core which is issued immediately before req_i. The delay is 80 given the above system configuration. The last row in Table VI gives the average estimation improvement for all the programs.

the WCET bounds by our approach is tightened by 33% in average, and the maximum is 82%, which is observed from *bs* in schedule S1 for example. WCET with maximal improvement for each program is shown in bold numbers. Experiment results also show that the WCETs of the programs vary a lot if the tasks are scheduled differently. The reason lies in the overlapping of the burst periods of the programs. Figure 12 shows an example in which shaded areas represent the burst periods. In schedule S2, the burst period of *bs* is completely covered by that of *matmult*. In this case, the conflicts from the other program reaches the maximum, and the resulting WCET is very close to the worst-case. But in S6, when *bs* starts, *matmult* is running concurrently but not in its burst period. In this schedule, *bs* suffers no conflicts at all, and the WCET corresponds the best case. We can see that the overlapping of the burst periods greatly affects the WCET of the programs. Note that the lengths of the blocks in Figure 12 do not represent real execution time, and they are only used to show the overlapping of burst periods. Since our approach can precisely explore the overlapping, it produces tight estimations for all execution scenarios. Assuming worst-case bus delay could be too pessimistic for some execution scenarios.

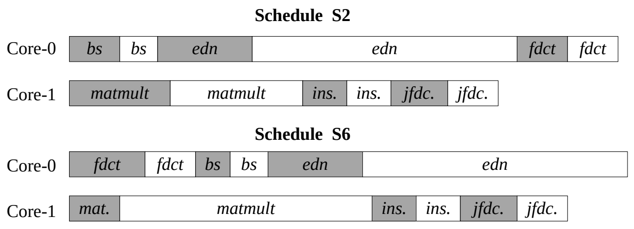


Figure 12. Different overlapping of burst periods in S2 and S6

Note that the improvement for *matmult* is very small. This is because *matmult* spends a lot of time in small loops, so cache hits dominate the execution time, and there is not a big gap between the best-case and the worst-case.

For the analysis performance, the longest experiment runs 3,362 seconds, and the maximum memory usage is 477MB. Although not so efficient as the TDMA bus, analyzing the FCFS bus still scales well and can handle real-life programs.

VI. CONCLUSION

Bounding the execution time of programs on multicores with shared buses is a challenging task. We solved this problem by presenting an approach that combines abstract interpretation with model checking to estimate the WCET of multicore software considering variable memory access time. We analyze the memory access behavior of programs running on dedicated cores with local caches using abstract interpretation techniques to generate timed automata capturing the precise timing information of the programs on when to access the memory bus. Based on the techniques presented in this paper, we have developed a tool for multicore timing analysis, which allows automatic generation of the TA models from binary code and WCET estimation for any given TA model of the shared bus. Experimental results show that our combined approach can significantly improve the analysis precision. The WCET bounds by our approach are tightened by up to 240% and 82% for the TDMA and FCFS bus respectively, compared with the worst-case bounds estimated based on cache misses and maximal bus access delay. Our technique may deal with a broad range of shared buses, that can be modeled using timed automata, other than TDMA and FCFS used as examples in this paper.

References

[1] S. Williams, A. Waterman, and D. Patterson, Roofline: an insightful visual performance model for multicore architectures. Communications of the ACM, vol.52, no.4, 2009.

[2] J. Bengtsson, Y. Wang. Timed Automata: Semantics, Algorithms and Tools. In Lecture Notes on Concurrency and Petri Nets. LNCS 3098, Springer-Verlag, 2004.

[3] H. Theiling. Control Flow Graphs for Real-Time System Analysis: Reconstruction from Binary Executables and Usage in ILP-Based Path Analysis. PhD thesis of Saarland University, 2002.

[4] C. Ferdinand. Cache Behavior Prediction for Real-Time Systems. Ph.D. Thesis of Saarland University, 1997.

[5] X. Li, A. Roychoudhury, and T. Mitra. Modeling Out-of-Order Processors for WCET Analysis. Real-Time Systems 34(3): 195-227, 2006.

[6] S. M. Yau-Tsun Steven Li. Performance analysis of embedded software using implicit path enumeration. 32nd Conference on Design Automation, 1995., pages 456C461, 1995.

[7] J. Yan and W. Zhang. Wcet Analysis for Multi-core Processors with Shared L2 Instruction Caches. In 14th Real-Time and Embedded Technology and Applications Symposium, 2008.

[8] W. Zhang and J. Yan. Accurately Estimating Worst-Case Execution Time for Multi-Core Processors with Shared Directmapped Instruction Caches. In International Workshop on Real-Time Computing Systems and Applications, 2009.

[9] Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury. Timing Analysis of Concurrent Programs Running on Shared Cache Multi-Cores. In RTSS, 2009.

[10] S. Chattopadhyay, A. Roychoudhury, and T. Mitra. Modeling Shared Cache and Bus in Multi-cores for Timing Analysis. Accepted to SCOPES, 2010.

[11] D. Hardy, T. Piquet, and I. Puaut. Using Bypass to Tighten WCET Estimates for Multi-Core Processors with Shared Instruction Caches. In RTSS, 2009.

[12] N. Guan, M. Stigge, W. Yi, and G. Yu. Cache-Aware Scheduling and Analysis for Multicores. In EMSOFT, 2009.

[13] V. Suhendra and T. Mitra. Exploring Locking & Partitioning for Predictable Shared Caches on Multi-Cores. In Proceedings of the 45th annual Design Automation Conference, 2008.

[14] A. Andrei, P. Eles, Z. Peng and J. Rosen. Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. In 21th International Conference on VLSI Design, 2008.

[15] J. Rosén, A. Andrei, P. Eles and Z. Peng. Bus Access Optimization for Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. In 28th IEEE international Real-Time Systems Symposium, 2007.

[16] A. Schranzhofer, J. Chen and L. Thiele. Timing Analysis for TDMA Arbitration in Resource Sharing Systems. In RTAS, 2010.

[17] J. Stachulat, S. Schliecker, M. Ivers and R. Ernst. Analysis of Memory Latencies in Multi-Processor Systems. In 5th International Workshop on Worst-Case Execution Time (WCET) Analysis, 2007.

[18] B. Andersson, A. Easwaran and J. Lee. Finding an Upper Bound on the Increase in Execution Time Due to Contention on the Memory Bus in COTS-based Multicore Systems. ACM SIGBED Review, Vol.7, No.1, 2010.

[19] R. Pellizzoni and M. Caccamo. Impact of Peripheral-Processor Interference on WCET Analysis of Real-Time Embedded Systems. In IEEE Transcations on Computers, Vol.59, No.3, 2010.

[20] R. Bourgade, C. Ballabriga, H. Cassé, C. Rochange and P. Sainrat. Accurate Analysis of Memory Latencies for WCET Estimation. In RTNS, 2008.

[21] R. Kirner, A. Kadlec, and P. Puschner. Precise Worst-Case Execution Time Analysis for Processors with Timing Anomalies. 21st Euromicro Conference on Real-Time Systems, 2009.

[22] D. Hardy, and I. Puaut. WCET Analysis of Multi-level Noninclusive Set-Associative Instruction Caches. In Proceedings of the 2008 Real-Time Systems Symposium, 2008.

[23] D. Grund, and J. Reineke. Abstract Interpretation of FIFO Replacement. In Proceedings of the 16th international Symposium on Static Analysis, 2009.

[24] B. Lesage, D. Hardy, and I. Puaut. WCET Analysis of Multi-Level Set-Associative Data Caches. 9th Int'l Workshop on Worst-Case Execution Time Analisis, 2009.

[25] H. S. Negi, T. Mitra and A. Roychoudhury. Accurate Estimation of Cache-Related Preemption Delay. In proceedings of the 1st International Conference on Hardware/software Codesign and System Synthesis, 2003.

[26] A. Andrei. Energy Efficient and Predictable Design of Real-Time Embedded Systems. PhD thesis, Linkoping University, Sweden, 2007.

[27] E. Fersman, L. Mokrushin, P. Pettersson and Wang Yi. Schedulability analysis of fixed-priority systems using timed automata. In Theoretical Computer Science, Vol.354, no.2, 301-317, 2006.