Timed Automata for Modelling Caches and Pipelines

Franck Cassez
Macquarie University
Sydney, Australia
franck.cassez@mq.edu.au

Pablo González de Aledo Marugán [*]
University of Cantabria
Santander, Spain
pabloga@teisa.unican.es

In this paper, we focus on modelling the timing aspects of binary programs running on architectures featuring caches and pipelines. The objective is to obtain a timed automaton model to compute tight bounds for the worst-case execution time (WCET) of the programs using model-checking tehcniques.

1 Introduction

We focus on modelling the timing aspects of binary programs running on architectures featuring caches and pipelines. The objective is to obtain a model to compute tight bounds for the worst-case execution time (WCET) of binary programs. The reader is referred to [11] for an exhaustive presentation of WCET computation techniques and tools. The main approach for computing the WCET is based on abstract interpretation and integer linear programming.

Modelling pipelined processors with caches was first reported in [7, 6] (METAMOC). In METAMOC the model-checker UPPAAL is used to compute the WCET. METAMOC has still some disadvantages, the main one being the requirement of manual loop bound annotations. In [3, 4] we introduced a technique to compute the WCET of binary programs in a fully automated manner without the need for loop bound annotations. The binary program is modeled as an (untimed) finite automaton and the timing aspects of the hardware components like the caches and the concurrent units (stages) in the pipeline (e.g., fetch, decode, execute stages) are modeled with a timed automaton or product of simple timed automata. The executions of the program on the hardware correspond to the possible traces in the synchronised product of the program automaton and the NTA modelling the hardware components. We can then reduce the computation of the WCET to a real-time model-checking problem and use state-of-the-art tools like UPPAAL [2] to compute a tight bound for the WCET.

In our previous work [3, 4], we have designed and implemented some techniques (based on program slicing) to minimise the program automaton of a binary program and demonstrated that the model-checking approach based on timed automata provides very accurate bounds for the WCET of the binary programs from the Mälardalen University benchmarks [10]. The automata modelling the different stages of the pipeline are also very compact and captures the essential timing features of the pipeline. However, the other hardware components, the caches, are modelled by very detailed generic timed automata that record the full state of the caches and model the exact behaviour and content of the caches.

Model checking is known to suffer from the state explosion problem. For timed automata where the *clocks* are part of the state space it is even more important to try and reduce the state space of the model to analyse while preserving enough details to faithfully model the property to be checked. In this paper, we focus on the following problem: how to automatically compute small models for the behaviours of the caches. We show that reducing the size of the models of the hardware components provides substantial benefits in the model-checking approach to compute the WCET.

[*]Author gratefully acknowledges the funding from projects TEC2011-28666-C04-02, TEC2014-58036-C4-3-R and grant BES-2012-055572, awarded by the Spanish Ministry of Economy and Competitivity.

2 Programs and hardware

For the sake of clarity we restrict to a simple setting: the programs can only read/write into a fixed number of registers (not in main memory). A special register is the *program counter*, `PC`, that points to the next instruction to be executed. Our programs can contain non-deterministic choices. As we are interested in computing a *worst-case execution time* (WCET), this value should be well defined. We impose that a program do not contain executions of (finite) but arbitrarily large lengths. In other words, the computation tree of a program P must be a bounded depth. Although this assumptions are not realistic for general programs and architectures, previous work has already addressed the case [1], and in this work we focus on how to compute small cache models rather than design these models by hand.

A program *run* (or execution) is completely defined by a finite sequence of (positive) integers given by the successive values of the program counter (there is no input data). The execution time of a run is defined by the time it takes for the *hardware* to execute the sequence of instructions in the run. The hardware is composed of an *instruction cache* and the *main memory* to store the program, and a *one-stage pipeline* (CPU) to execute the instructions.

To execute a run, say $\sigma_1 = 1.2.3.1$, the code of instruction 1 (e.g., an addition between two registers) has to be fed to the CPU (pipeline) to be executed. The same applies for instruction 2 and so on. Initially, the code of each instruction is stored in the main memory and the cache is empty. To execute the instruction at `PC`$= i$ the following steps occur:

1. the CPU requests the code of the instruction i.
2. If the code of i is in the cache, this is a cache *Hit*, and the code of i is transferred from the cache to the CPU, otherwise, a cache *Miss* occurs. The code of i is fetched from main memory, stored in the cache and then transferred to the CPU.
3. the CPU executes the code of i and is ready to process the next instruction.

The cache is a fast memory component but has limited capacity (size). Assume our instruction cache has capacity 3, and is initially empty. Executing $\sigma_1 = 1.2.3.1$ will result in 3 cache misses and a cache hit (instruction 1). If the cache size is 2, and we are about to execute 3, the cache is full. We need to remove one instruction stored in the cache, either 1 or 2. A standard *replacement* policy is the FIFO policy where the least recently accessed item in the cache is removed. In our case, this would mean that 1 is removed and replaced by 3, and 2 becomes the least recently accessed item.

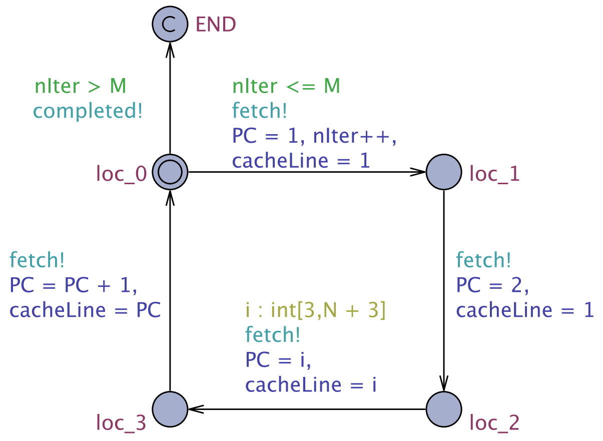
In real caches, each line of the cache may contain a *set* of instructions[1]. In this case, checking whether instruction i is in the cache reduces to checking whether $set(i)$ is the cache. Finally, executing the code of an instruction i takes some CPU cycles $dur(i)$.

In the next section, we show how to model the timing aspects of the hardware and how to compute the WCET of a given program.

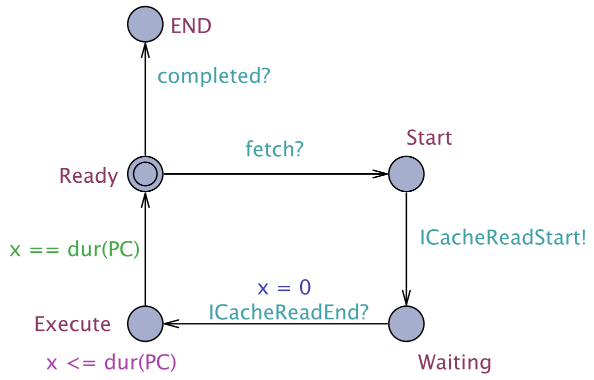
3 Timed automata model

Program and pipeline models. Fig. 1 provides an overview of our model specified as a NTA (in UPPAAL) and this will serve as a running example to illustrate our modelling techniques. The program automaton (Fig. 1(a)) models a program that performs M iterations of a loop, the body of which consists of N switch statements. The UPPAAL model of Fig. 1(a) encodes the switches as non-deterministic

[1]When transferring from the main memory to the cache, it is usually faster to transfer a sequence of consecutive instructions rather than a single one.



(a) Program automaton



(b) CPU (two-stage pipeline, Fetch/Execute)

Figure 1: Program and pipeline

choices from location `loc_2`. An integer i is non-deterministically picked within the interval $[3..3+N]$ and the program counter is set to this value. If $N = 0$ there is no switch and the program has a single run. If $N = 1$ there are two sets of runs (modelling an `if then else` statement): in `loc_2`, if $i = 3$ if picked, the next two instructions are $3, 4$ otherwise $i = 4$ and the next instructions are $4, 5$. By incrementing N we can model a control loop with N different inner branches. The variable `cacheLine` corresponds to set(`PC`).

We model the execution of an instruction (described in Section 2) as follows:

1. the program is eager to be executed and wants to send a `fetch!` event as soon as the receiver (CPU) is ready to receive (`fetch?`) it (handshake synchronisation). This is modelled in UPPAAL by an *urgent channel*, `fetch`.
2. when the `fetch!` occurs, both `PC` and `cacheLine` are set to hold the values of the next instruction to be executed and the corresponding cache set.
3. the CPU requests the code of the instruction by sending a `ICacheReadStart!` to the cache. This is an urgent channel as well (handshake synchronisation must happen as soon as both the sender and the receiver are ready).
4. When the instruction is in the CPU, the cache issues the `ICacheReadEnd!` event.
5. the CPU receives this event `ICacheReadEnd?` and is ready to execute the code of the instruction. Executing the instruction takes dur(`PC`). We encode the duration by using a *clock* x. The clock x is reset when entering the `Execute` location and the invariant $x \leq dur$(`PC`) and the guard $x ==$ dur(`PC`) (from `Execute` to `Ready`) ensures that the automaton remains dur(`PC`) cycles in `Execute` before moving to `Ready`.

Cache model. The model for the instruction cache is given by the timed automaton `FullCache` Fig. 2. When a cache read request is received (`ICacheReadStart?`) location `Check` is reached. The location `Check` is *committed* which in UPPAAL implies that time cannot elapse and we must leave this location instantaneously[2].

[2]This is a loose definition of the actual semantics of committed in UPPAAL but valid for our model.

If the instruction `PC` is in the cache, `find(cacheLine)` returns a positive value, otherwise a negative value. The timing behaviour of the cache depends on whether the instruction `PC` is in the cache or not: if `PC` is in the cache, a *Hit* occurs and the delay is set to `HitTime` (e.g., 2 cycles). Otherwise, a *Miss* occurs and the delay is set to `MissTime` (e.g., 20 cycles). In both cases the cache is updated by executing the `access(cacheLine)` function. The UPPAAL specification of the functions `find(·)` and `access(·)` are given in Appendix A.

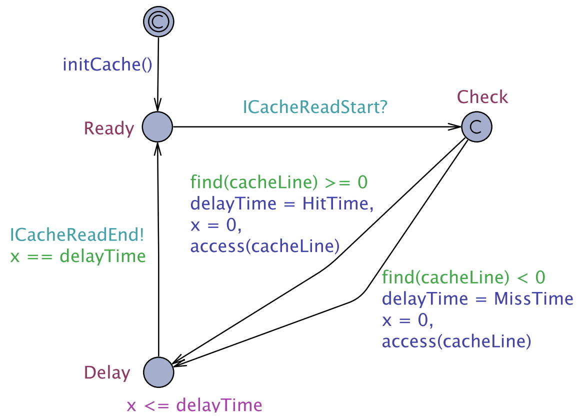


Figure 2: Instruction Cache automaton `FullCache`

4 Computing the WCET

Real-time model checking. To compute the WCET of our program, we synchronise the previous components, program, CPU (pipeline) and cache to build an NTA. Assume we have a clock `GBL_CLK` which is initially set to 0 and never reset. The location `END` of the program (Fig. 1(a)) is committed and has no outgoing edges; this implies that time cannot elapse and no other transition can be taken, so it creates a deadlock. This ensures that when we reach the end of the program, the value of `GBL_CLK` is frozen. Without blocking the system that way, `GBL_CLK` could grow arbitrarily large in the `END` location of the program. In this model, the WCET of our program running on the simple hardware, is the maximal value that `GBL_CLK` can hold at the end of the execution of the program, in location `END`. This can be easily computed in UPPAAL with the *sup* operator.

In our previous [3, 4, 1] work we designed a very accurate model of the ARM920 hardware and compared the results obtained with our method (based on timed automata), with actual execution times on a real ARM board. On the set of benchmarks from Mälardalen University we managed to compute very tight WCETs.

Advantages of timed automata modelling. Using timed automata to model (binary) programs running on complex hardware has several advantages:

- NTA have precise semantics; our models are formal behavioural models for the hardware.
- we define the model of a program running on some hardware as the composition (synchronisation) of very simple components. All these components can be validated separately.
- we can model features that cannot be modelled with other approaches based on abstract interpretation and Integer Linear Programming (ILP); for example, if the execution time of an instruction is within an interval (depends on the data to be processed), we can easily model the duration by an interval in the CPU model. We can also model changes of the speed of the CPU during program executions.
- our models and methodology ensures that we obtain an upper bound of the actual WCET provided that the models of the hardware are certified to be good abstractions (time-wise) of the actual board.

- our methodology is resilient to *timing anomalies* [5]. This is a challenging problem faced by other methods based on abstract interpretation.

On the other hand, model-checking is very sensitive to the number of states of the system to analyse: this is the well known state explosion problem that can hinder the analysis of large systems. This is even more true when model-checking timed automata: the state space is composed of a discrete part and clock constraints.

Reducing the state space. The model checking approach we have introduced is mostly state based: the program and the hardware are modelled as timed automata and we can generate all the possible states of the system and obtain the maximal value of a clock in a particular location.

A key concept in our approach [3] is to define the WCET problem as a language based problem:

- the program P is viewed as a generator of sequences of instructions; the sequences are in a language $\mathscr{L}(P)$ generated by an automaton $Aut(P)$.
- the hardware H is a transducer: given a sequence $w \in \mathscr{L}(P)$, it computes an integer value, the execution time of w, number of cycles.

The information needed to compute the WCET is captured by the sequences of instructions (the `PC`). We do not need to simulate the actual effect of each instruction. In other words, if we can generate $\mathscr{L}(P)$ and we have a timed automaton model of the hardware, we can compute the WCET by examining all the words in $\mathscr{L}(P)$ and "running" them in the model of the hardware (viewed as a transducer). If we generate $\mathscr{L}(P)$ with the actual program by keeping track of the full state of the program (registers, stack), we may end up generating the same sequence of instructions many times.

In [3, 4, 1] we have shown how to compute (using program slicing) a reduced program P' and an associated program automaton $Aut(P')$, such that $\mathscr{L}(P') = \mathscr{L}(P)$ but $Aut(P')$ is *minimal*: it generates every word in $\mathscr{L}(P)$ only once. We have demonstrated that this is essential for being able to compute the WCET of real programs. In practice, using $Aut(P')$ avoids feeding the hardware transducer twice with the same sequence of instructions hence reducing the amount of work of the real-time model-checker. Still, in our approach the caches are modelled explicitly and this is not optimal as demonstrated in the next paragraph.

Equivalent states in cache models. The execution time of a sequence of instructions depends on the sequence of hits and misses and the time it takes for the CPU to execute each instruction. Assume that the time to execute each instruction is 0 and only the cache hits and misses impact the execution time.

With a model that explicitly represents the cache, we can precisely track the hits and misses. For instance for the sequence $\sigma_1 = 1.2.3.1$ and a cache of size 3, the successive states of the cache are $\varnothing$, 1, 1.2, 1.2.3 and we obtain the following sequence of pairs (instruction, Hit/Miss) : $(1,M).(2,M).(3.M).(1,H)$. This sequence fully determines the execution time. It turns out that the sequence of instructions 1.4.5.1 would produce exactly the same execution time (if 4 and 5 have the same execution time as 2 and 3).

Now let P_1 be the program that comprises of two sequences of instructions $\pi_1 = 1.2.3.w$ and $\pi_2 = 1.4.5.w$, with $w = k.w'$ a finite sequence of instructions that does not contains $\{1,2,3,4,5\}$. Assume 1 takes 1 cycle to execute, 2 and 3 both take 2 cycles and 4 and 5 both take 1 cycle to execute. A cache hit costs 1 and a cache miss 10. If we want to compute the WCET of P_1 we have to analyse both π_1 and π_2. As w is the same in π_1 and π_2, the `PC` of the program before w is k and is the same after 1.2.3 and 1.4.5. But the cache is in a different state after 1.2.3 and 1.4.5. After 1.2.3 the current time given by `GBL_CLK` is 35 and after 1.4.5, `GBL_CLK` is 33. To compute the WCET in our model-checking based approach with

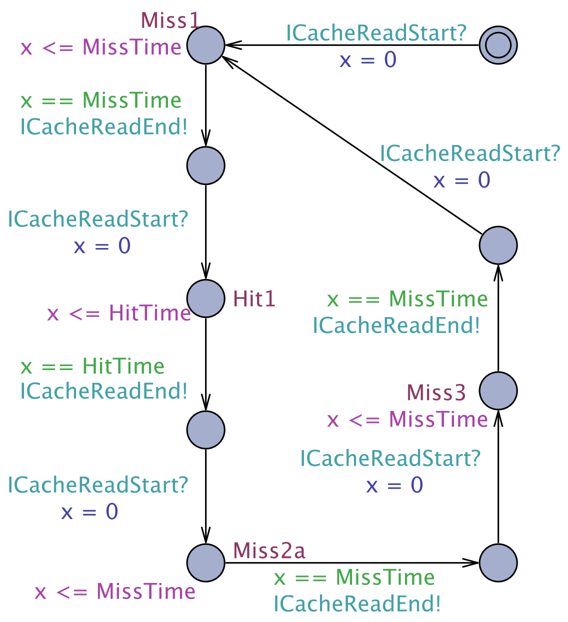


Figure 3: A small cache model

N	States Explored		WCET
	Explicit Model	Small Model	
1	549	147	396
2	1055	196	396
3	1626	245	396
4	2267	294	396
5	2953	343	396
6	3699	392	396
7	4505	441	396
8	5371	490	396
9	6297	539	396
10	7283	588	396

Table 1: States explored for computing the WCET

explicit cache, we enumerate the (symbolic) state space of the synchronised product of the program and the cache. In the state space we can reach two different symbolic states $s_1 = (k, 1.2.3, \texttt{GBL\_CLK} = 35)$ and $s_2 = (k, 1.4.5, \texttt{GBL\_CLK} = 33)$. They have to be explored (e.g., in a Depth-First Search) to compute the WCET. From s_1 and s_2 the sequences of pairs (i, α) of (instructions, Hit/Miss) will be the same when executing w. As we know that GBL_CLK is the largest after 1.2.3, is never reset, we do not need to explore the state s_2 and the full path π_2 because it will surely yield a smaller execution time. The states s_1 and s_2 can be considered as equivalent *given* the program we analyse: they will generate the same sequences of Hits and Misses and the same execution time.

Computing equivalent cache states can be achieved using abstraction refinement techniques as described in Section 5. The following example shows that using an abstract cache can lead to drastic reductions in the state space of the system to analyse while presering the WCET.

An example. Consider the program of Fig. 1(a). We fix the maximum number of iterations to $M = 5$. We can compute the WCET of the program with an explicit cache of size 2. A model in which equivalent cache states are collapsed is given by Fig. 3. With a cache of size 2 all the runs in the program generates sequences of Hit and Miss of the form $(M.H.M.M)^*$. Hence the automaton of Fig. 3 is a good cache model for this program for any number of switches N.

Table 1 gives the number of states explored[3] to compute the WCET with the explicit cache and the small cache of Fig. 3. As can be seen, many states of the cache are equivalent and this is captured by the small model whereas the explicit model generates all the configurations.

The explicit state cache model can be used with any program to analyse but may inflate the state space to be explored for computing the WCET. Another disadvantage of the explicit state cache model is that the initial state has to be fixed. In the next section we describe how to compute small cache models *for a given program.*

[3]We used UPPAAL and the command `verifyta -t0 -u simple-prog.xml`.

5 Computing abstract cache models

As demonstrated in the previous section, using an explicit cache model can be detrimental to the model-checking approach as many equivalent (time-wise) cache states may be explored. The objective of this section is to indicate how we can compute small cache models that are good abstractions *for a given program*. To do this, we use an abstraction refinement technique introduced in [8, 9].

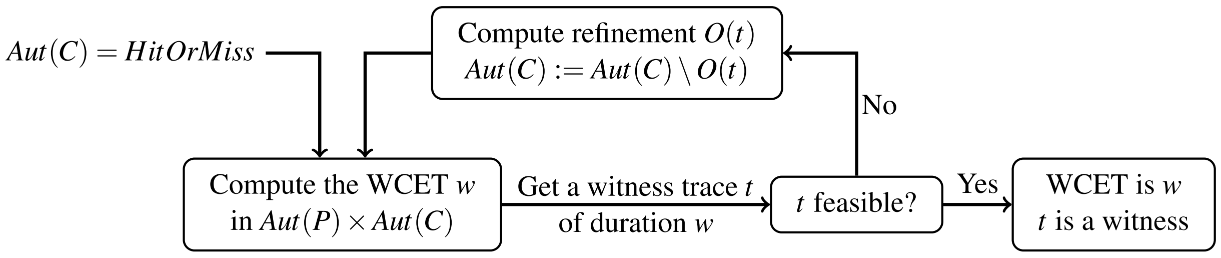

$Aut(C) = HitOrMiss$
Compute refinement $O(t)$
$Aut(C) := Aut(C) \setminus O(t)$
No
Compute the WCET w in $Aut(P) \times Aut(C)$
Get a witness trace t of duration w
t feasible?
Yes
WCET is w
t is a witness

Figure 4: Trace Abstraction Refinement Algorithm for computing WCET

Computation of WCET via trace refinement. The *trace abstraction refinement technique* was originally developed to analyse imperative programs. We adapt here it to compute the WCET of a program. Our method works as follows to compute the WCET of program P:

1. start with the most abstract model of the cache: every cache access can be either a hit or a miss;
2. compute the WCET, w, of P with the current model of the cache.
3. get a witness trace t (sequence of pairs (instructions, Hit/Miss)) that yields this execution time w.
4. check if t is *feasible* in a concrete model of the cache. t contains the sequence of instructions and using the specification of the cache we can compute whether the corresponding sequence of hit and miss is feasible or not. Notice that we do not need to specify an initial state for the cache but rather check that there is an initial state of the cache such that t is feasible.
5. if t is feasible, the WCET is w and t is a witness trace.
6. otherwise t is infeasible in any concrete model of the cache, and we have to refine the abstract cache. The refined abstract cache model should not allow t. We iterate this process and re-start at step 2.

Step 6 above is the instrumental step of the trace abstraction refinement method: from one infeasible trace, we can compute a *set* of traces that are infeasible for the same reason. From t we can obtain a regular language of traces that are infeasible and represent it by a finite automaton $O(t)$. This crucial step is based on the computation of *interpolants*. We have defined a logic for caches and the corresponding notion of interpolants that enables us to compute $O(t)$ for each infeasible trace t.

Fig. 4 gives an algorithm to compute the WCET of a program P by iteratively refining the initial abstract model of the cache. The initial abstract model for the cache is the timed automaton *HitOrMiss* which allows the cache to generate a Hit or a Miss for each memory access. When an infeasible trace is discovered, we refine the cache model by removing a set of infeasible traces given by $O(t)$. Notice that the cache model we iteratively compute does not make any assumption on the initial state of the cache. The traces captured by the automata $O(t)$ are not feasible for any initial state of the cache.

6 Conclusion

We have presented a method to compute the WCET based on timed automata and real-time model-checking with UPPAAL. The method we propose computes the model of the instruction cache as needed by refining the initial abstract model of the cache. This yields a reduction in the search space, which results in smaller models to be analysed and is key step towards the scalability of our timed automata based method. Another interesting feature of this approach is that we do not need to assume that the initial state of the cache is known. Indeed, checking whether a trace is feasible can be encoded as a SAT problem. The refined models of the cache we compute generate the cache behaviours that are possible from *some* initial state of the cache. We are currently implementing the refinement loop and the computation of the refined models of the cache.

References

[1] Jean-Luc Béchennec & Franck Cassez (2011): *Computation of WCET using Program Slicing and Real-Time Model-Checking*. *CoRR* abs/1105.1633. Available at http://arxiv.org/abs/1105.1633.

[2] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, John Håkansson, Paul Pettersson, Wang Yi & Martijn Hendriks (2006): *UPPAAL 4.0*. In: *QEST*, IEEE Computer Society, pp. 125–126. Available at http://doi.ieeecomputersociety.org/10.1109/QEST.2006.59.

[3] Franck Cassez (2011): *Timed Games for Computing WCET for Pipelined Processors with Caches*. In *11th International Conference on Application of Concurrency to System Design, ACSD 2011*, IEEE Computer Society, pp. 195–204, Available at http://dx.doi.org/10.1109/ACSD.2011.15.

[4] Franck Cassez & Jean-Luc Béchennec (2013): *Timing Analysis of Binary Programs with UPPAAL*. In: *13th International Conference on Application of Concurrency to System Design, ACSD 2013*, IEEE Computer Society, pp. 41–50, Available at http://dx.doi.org/10.1109/ACSD.2013.7.

[5] Franck Cassez, René Rydhof Hansen & Mads Chr. Olesen (2012): *What is a Timing Anomaly?* In: *12th International Workshop on Worst-Case Execution Time Analysis, WCET 2012, July 10, 2012, Pisa, Italy*, *OASICS* 23, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 1–12. Available at http://dx.doi.org/10.4230/OASIcs.WCET.2012.1.

[6] Andreas E. Dalsgaard, Mads Chr. Olesen, Martin Toft, René Rydhof Hansen & Kim Guldstrand Larsen (2010): *METAMOC: Modular Execution Time Analysis using Model Checking*. In Björn Lisper, editor: *WCET*, *OASICS* 15, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, pp. 113–123. Available at http://dx.doi.org/10.4230/OASIcs.WCET.2010.113.

[7] Andreas Engelbredt Dalsgaard, Mads Christian Olesen & Martin Toft (2009): *Modular Execution Time Analysis using Model Checking*. Master's thesis, Dpt. of Computer Science, Aalborg University, Denmark.

[8] Matthias Heizmann, Jochen Hoenicke & Andreas Podelski (2009): *Refinement of Trace Abstraction*. In Jens Palsberg & Zhendong Su, editors: *SAS*, *Lecture Notes in Computer Science* 5673, Springer, pp. 69–85. Available at http://dx.doi.org/10.1007/978-3-642-03237-0_7.

[9] Matthias Heizmann, Jochen Hoenicke & Andreas Podelski (2013): *Software Model Checking for People Who Love Automata*. In Natasha Sharygina & Helmut Veith, editors: *CAV*, *Lecture Notes in Computer Science* 8044, Springer, pp. 36–52. Available at http://dx.doi.org/10.1007/978-3-642-39799-8_2.

[10] Mälardalen WCET Research Group: *WCET Project – Benchmarks*. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

[11] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat & Per Stenström (2008): *The Worst-Case Execution-Time Problem - Overview of Methods and Survey of Tools*. *ACM Trans. Embedded Comput. Syst.* 7(3). Available at http://doi.acm.org/10.1145/1347375.1347389.

A Instruction cache UPPAAL code (declarations)

```
int MissTime = 20;              // number of cycles taken by a Miss
int HitTime = 2;                // number of cycles taken by a Hit

const int cache_size = 2;       // cache size is 2
const int bot = - 1;            // dummy place holder element

int delayTime;                  // how long it takes to fetch the data

int cache_content[cache_size]; // array to hold the cache content

clock x;

// initialise array to bot
void initCache() {
  int i;
  for (i = 0; i <= cache_size; i++) {
    cache_content[i] = bot;  // initialised to bot means not in
  }
}

// check if element is the cache
int find(int element){
  int i;
  // scan cache
  for ( i = 0; i <= cache_size; i++) {
    if(cache_content[i] == element) return i;
  }
  return -1;  //  not found
}

void insert(int k,int element){
  int i;
  //  maintain the cache in FIFO order
  for ( i = k; i >= 1; i--) {
     cache_content[i] = cache_content[i-1];
  }
  cache_content[0] = element;
}

void access(int thePC){
  int i = find(thePC);  // is it in?
  if(i <= 0) {
    // not found, cache Miss, insert
    insert(cache_size - 1, thePC);
  } else {
    // found, cache Hit, update FIFO cache
    cache_content[i] = bot;
    insert(i, thePC);
  }
}
```