WUPPAAL: Computation of Worst-Case Execution-Time for Binary Programs with UPPAAL

Franck Cassez[1], Pablo Gonzalez de Aledo[1,3], and Peter Gjøl Jensen[1,⋆]

[1] Macquarie University, Sydney, Australia
[2] Aalborg University, Aalborg, Denmark
[3] University of Cantabria, Santander, Spain

Abstract. We address the problem of computing the worst-case execution-time (WCET) of binary programs using a real-time model-checker. In our previous work, we introduced a fully automated and modular methodology to build a model (network of timed automata) that combined a binary program and the hardware to run the program on. Computing the WCET amounts to finding the longest path time-wise in this model, which can be done using a real-time model checker like Uppaal. In this work, we generalise the previous approach and we define a generic framework to support arbitrary binary language and hardware.
We have implemented our new approach in an extended version of Uppaal, called Wuppaal. Experimental results using some standard benchmarks suite for WCET computation (from Mälardalen University) show that our technique is practical and promising.

Keywords: binary program, control flow graph, worst-case execution-time

1 Introduction

Embedded real-time systems (ERTS) are composed of a set of periodic tasks (software) to run on a given architecture (hardware). The tasks are usually released at periodic time intervals. For safety-critical ERTS, each task must be completed by a deadline (relative to the release time). Checking whether a set of periodic tasks can be scheduled on a processor such that they always complete before their deadline is a *schedulability analysis.*

Tests for schedulability are based on the tasks' parameters, among them an upper bound for the *execution time* of each task. Over-estimating the execution time of a task may be safe but can also result in a set of tasks being declared non schedulable. This may lead to a choice of over-powered and over-expensive hardware.

With the ever increasing connectivity of many devices, ERTS are also subject to malicious attacks. Some of them can make use of time measurements to

[⋆] Part of this work was done while this author visited Macquarie University.

establish communication channels (*timing covert channel*): private information can be communicated or leaked to attackers by controlling/observing the time intervals between events (e.g., how long a computation takes).

It follows that tight bounds for the execution time of the tasks are instrumental to designing safe (schedulable), efficient and secure ERTS. Each task in an ERTS executes a program. The execution time of the program may depend on the input. The worst-case execution-time (WCET) of the program is the supremum of the execution times of the program over all the input. Computing the WCET for binary programs is a non-trivial task for at least two reasons:

- the set of input data may be very big and simulating the program over a subset of the input data only provides a lower bound of the worst-case execution-time;
- the hardware that runs the program is complex (pipelined architecture, caches) and it is effectively a *timed concurrent system* (e.g., the pipeline runs in parallel with the caches and they both have timing specifications.)

The WCET problem. Given a binary program P, some input data d and the hardware H, the *execution time* of P for the input d on H, denoted $\mathsf{Xtime}(P, d, H)$, is measured as the number of processor cycles between the beginning and end of P's computation for d (we assume P always terminates.) The *worst-case execution time (WCET)* of program P on hardware H, denoted $\mathsf{WCET}(P, H)$, is the supremum of the $\mathsf{Xtime}(P, d, H)$ for d ranging over the input data domain $\mathcal{D}$:

$$\mathsf{WCET}(P, H) = \sup_{d \in \mathcal{D}} \mathsf{Xtime}(P, d, H). \tag{1}$$

The WCET problem asks the following:

“Given P and H, compute $\mathsf{WCET}(P, H)$”.

In general, the WCET problem is undecidable because otherwise we could solve the halting problem. However, for programs that always terminate and have a bounded number of paths, it is computable. Indeed the possible runs of the program can be represented by a finite tree (and there is a finite number states for the program and the hardware). This does not mean that the problem is tractable though: the (values of the) input data (e.g., an fixed-size array to be sorted) are usually unknown and the number of program paths to be explored may grow exponentially in the size of the program.

As mentioned before, programs run on increasingly complex architectures featuring multi-stage *pipelines* and fast memory components like *caches*: they both influence the WCET in a complicated manner. It is then a challenging problem to determine a precise WCET even for relatively small programs running on complex single-core architectures.

Computing a precise WCET for a given program is very hard and the WCET problem is usually re-stated as:

“Given P and H, compute a *tight upper bound* of $\mathsf{WCET}(P, H)$”.

Tightness can be measured (see [9]) by comparing actual WCET to the ones computed using a particular method. In the sequel we use $\mathsf{WCET}(P, H)$ to denote an upper bound of the WCET for a given program.

Standard methods and tools for computing WCET. The survey article [23] provides an exhaustive presentation of WCET computation techniques and tools. A first set of methods based on simulations ([18, 5, 19]) are not suitable for safety-critical ERTS as they only provide lower bounds for the WCET.

A second set of methods rely on the construction of a Control Flow Graph (CFG) for the binary program to analyse, and the determination of *loop bounds*. The CFG is then annotated with some timing information about the cache misses/hits (some may/must analysis using abstract interpretation based techniques) and pipeline stalls to build a finite model of the system. A final paths analysis is carried out on this model e.g., using Integer Linear Programming (ILP). There are many implementations of this technique, the most prominent one is probably aiT [1, 13] which combines static analysis tools and ILP for computing WCET.

Real-time model-checking based methods for computing WCET. Considering that (*i*) modern architectures are composed of *concurrent* components (the units of the different stages of the pipeline, the caches) and (*ii*) the *synchronisation* of these components depends on *timing constraints* (time to execute in one stage of the pipeline, time to fetch data from the cache), formal models like *timed automata* [2] and state-of-the-art *real-time model-checkers* like Uppaal [16, 3] appear well-suited to address the WCET problem.

The use of network of timed automata (NTA) and the model-checker Uppaal for computing WCET on pipelined processors with caches was first reported in [12, 11] where the METAMOC method is described. METAMOC consists in: 1) computing the CFG of a program, 2) composing this CFG with a (network of timed automata) model of the processor and the caches. Computing the WCET is then reduced to computing the longest path (time-wise) in a NTA.

The previous framework is very elegant yet has some shortcomings: (1) METAMOC relies on a *value analysis* phase to compute the CFG but this may not terminate, (2) some programs cannot be analysed (if they contain register-indirect jumps), (3) manual annotations (loop bounds) is required on the binary program, and (4) the *unrolling* of loops is not safe for some cache replacement policies (FIFO). In our previous work [7, 9] we have reported some results on the computation of WCET using NTA that overcome the limitations of METAMOC: (1) we introduced an automatic method to compute a CFG and a reduced abstract program equivalent WCET-wise to the original program; (2) we designed detailed hardware formal models and (3) we evaluated the accuracy of our technique (comparison of measured execution times and the results of our analysis).

The technique we introduced in [7, 9] still has some drawbacks:

- the Uppaal model (NTA) contains the CFG of the program and the machinery that is needed to simulate some instructions (written as functions in Uppaal); some instructions (e.g., setting the overflow flag) are partially modelled because of the restricted expressiveness of the C-like operators supported by UPPAAL;

- the UPPAAL model (NTA) also contains components to explicitly model the caches as large arrays (of cache lines) which contributes a big part of the state of the system;
- as a result, we rely on UPPAAL to perform a lot of discrete computations which is not effective; moreover, the discrete state of the UPPAAL model contains a large amount of information (e.g., the full state of the caches) which also impacts the efficiency of the UPPAAL analysis engine.

Our contribution. Based on our previous work [7, 9], we propose three new contributions: (1) a generic framework for computing WCET which is language agnostic; (2) a new implementation of our framework based on an extended version UPPAAL and (3) a tool chain that combines our extended UPPAAL and an off-the-shelf binary program simulator (based on *gdb* [22]).

Outline of the paper. In Section 2 we recall how the WCET can be computed via model-checking. The material in this section is based on [7, 9]. In Section 3, we introduce our new generic technique to compute the WCET of arbitrary programs. Examples are provided for an monoprocessor pipelined ARM architecture. Section 4 provides details of the implementation of our technique, a tool chain architecture and some experimental results.

2 Computation of WCET via real-time model-checking

In this section we introduce the basic concepts of program runs together with an abstract model of the hardware in order to compute the execution time of a sequence of program instructions.

Hardware. The hardware usually consists of a finite set $\mathcal{R}$ of registers, a multi-stage execution pipeline and caches (e.g., instruction and data caches). It typically supports a finite set of instructions, $\mathcal{I}$, e.g., `mov r1,r2` is an instruction that copies the contents of register r_2 into register r_1. The main memory component is a table of words of a given width 32-bit or 64-bit words. $\mathcal{M}$ is the (finite) set of main memory cells and we denote $\mathcal{D}$ the memory domain (e.g., 32-bit or 64-bit words). A memory state is thus a map from $\mathcal{M}$ to $\mathcal{D}$. The caches and the pipeline are essential components of the hardware performance-wise but they are not necessary to define the semantics of the instructions. We omit them for now and will account for them later in this section. A *state* of the hardware is fully determined by the contents of the registers, the contents of the memory and the contents of the pipelines and caches. The hardware has a designated register, the *program counter* that points to the next instruction to process. An example of such an architecture, the ARM920T, is given in Fig. 1. The orange blocks are the blocks we need to model to compute the execution time of program runs.

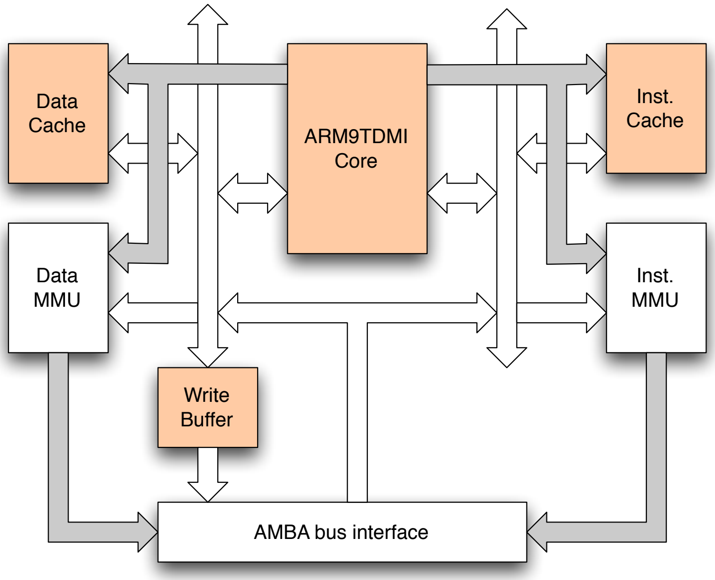


Fig. 1: Simplified ARM920T architecture

Program runs. A binary program is a map $P : \mathbb{P} \rightarrow \mathcal{I}$, with $\mathbb{P} \subseteq \mathcal{M}$, that associates with some memory locations $\ell \in \mathbb{P}$ an instruction. $P(\ell)$ is the instruction to be processed when the program counter of the hardware is at ℓ.

Given a program P, we let $\mathcal{L}_H(P) \subseteq \mathbb{P}^*$ be the set of *valid* executions of P on H. Actually we only require $\mathcal{L}_H(P)$ to over-approximate the set of feasible executions of the program P. To define this set we need to take into account the semantics of each instruction in $\mathcal{I}$, and the values of the registers of H and the memory state: this state is given by a valuation $\nu : \mathcal{R} \cup \mathcal{M} \rightarrow \mathcal{D}$. There are usually many different possible initial states of the hardware (e.g., a sorting program that sorts an array of k arbitrary elements, there are $\mathcal{D}^k$ initial possible input data).

Listing 1.1: Prog1

```
int c_entry(int a, int b){
        int c=1,i;
        for (i = 0; i < 10; i++) {
                if(a < b){
                        c *= 10;
                } else {
                        c += 10;
                }
        }
        return c;
}
```

An example of a binary program compiled for the ARM920T is provided in Fig. 2.a. This program can be obtained by compiling the C program Prog1 (Listing 1.1). The Control Flow Graph (CFG) is given in Fig. 2.b. The semantics of the program does not depend on the pipeline architecture nor on the caches: these components only impact the execution time of the program runs. However, to ensure that the WCET of each program is well-defined, we may assume that $\mathcal{L}_H(P)$ is finite. Otherwise it contains arbitrary long sequences (the alphabet $\mathbb{P}$ is finite) and the set of execution times is unbounded and the WCET is $+\infty$.

The set $\mathcal{L}_H(P)$ of program runs is finite but may contain more than one trace even if the program is deterministic. For instance in Prog1 (Listing 1.1), the values of `a, b` are arbitrary at the beginning of the program because they are parameters of the function `c_entry`. This makes the test at line 4 a non-deterministic choice in our program over-approximation because the values of a and b are arbitrary (there are input parameters of the `c_entry` function). We can over-approximate the set of runs of this program by assuming that each time the test at line 4 is performed, the outcome is either true or false and both cases should be taken into account to compute the WCET. Notice that this is an over-approximation if $a < b$ evaluates to TRUE (resp. FALSE) the first time it must evaluate to TRUE (resp. FALSE) in the following iterations. Using this strategy we generate a super set of the set feasible runs of Prog1.

Execution time of a run. The execution time of a run $\sigma \in \mathbb{P}^*$ typically depends on the following factors:

- the time it takes for the instructions in σ to flow into the pipeline stages. This is usually non-trivial as the stages run in parallel. Moreover, the flow of instructions in the successive stages of the pipeline is governed by precedence rules: the execution of an instruction may require the availability of the result of another instruction which may temporarily block an instruction in a pipeline stage: this is known as a pipeline *stall*.
- the time it takes to fetch instructions and data from the caches and main memory.
  These memory transactions are usually performed in different pipeline stages and can be concurrent (e.g., an instruction in the *fetch* stage can be fetched from the instruction cache while another instruction in the *memory* stage performs some transactions with the data cache.)

In order to determine how long it takes for a run $\sigma \in \mathbb{P}^*$ to execute on the hardware H, it is sufficient to know:

- the processing time of each instruction in the different pipeline stages,
- the registers read from/written to by each instruction (to determine pipeline stalls),
- the status of the memory transactions for the instructions in σ: cache *hits* and *misses*.

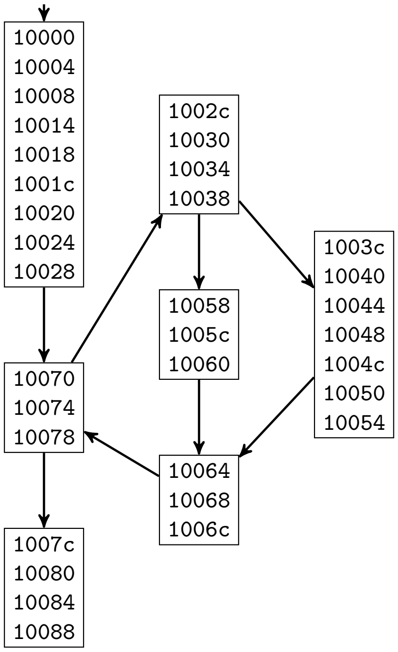
Given a run $\rho \in \mathcal{L}_H(P)$, we can build an *annotated run* $\tilde{\rho}$ that contains the information required to fully determine the execution time of ρ on H. This extended run may capture the processing time of the instruction in each pipeline stage, the registers read from/written and the cache hits and misses. We let $\mathcal{L}^a_H(P)$ be the set of annotated runs associated with $\mathcal{L}_H(P)$.

For example, the following run $\rho = 10000.10004.10008.10014.10018$ in $\mathcal{L}_H(P)$ can be annotated with the time it takes to process each corresponding instruction in Prog1 (Fig. 2.b), and whether fetching the instruction (from the instruction cache) will result in cache Hit or a cache Miss. Hence $\mathcal{L}^a_H(P)$ can be defined

```
10000 <_Reset>:
10000: e1a00000 nop
10004: e59fd004 ldr sp, [pc, #4]
10008: eb000001 bl 10014 <c_entry>
1000c: eafffffe b 1000c <_Reset+0xc>
10010: 00011090 .word 0x00011090

10014 <c_entry>:
10014: e24dd010 sub sp, sp, #16
10018: e3a03001 mov r3, #1
1001c: e58d300c str r3, [sp, #12]
10020: e3a03000 mov r3, #0
10024: e58d3008 str r3, [sp, #8]
10028: ea000010 b 10070 <c_entry+0x5c>
1002c: e59d2004 ldr r2, [sp, #4]
10030: e59d3000 ldr r3, [sp]
10034: e1520003 cmp r2, r3
10038: aa000006 bge 10058 <c_entry+0x44>
1003c: e59d200c ldr r2, [sp, #12]
10040: e1a03002 mov r3, r2
10044: e1a03103 lsl r3, r3, #2
10048: e0833002 add r3, r3, r2
1004c: e1a03083 lsl r3, r3, #1
10050: e58d300c str r3, [sp, #12]
10054: ea000002 b 10064 <c_entry+0x50>
10058: e59d300c ldr r3, [sp, #12]
1005c: e283300a add r3, r3, #10
10060: e58d300c str r3, [sp, #12]
10064: e59d3008 ldr r3, [sp, #8]
10068: e2833001 add r3, r3, #1
1006c: e58d3008 str r3, [sp, #8]
10070: e59d3008 ldr r3, [sp, #8]
10074: e3530009 cmp r3, #9
10078: daffffeb ble 1002c <c_entry+0x18>
1007c: e59d300c ldr r3, [sp, #12]
10080: e1a00003 mov r0, r3
10084: e28dd010 add sp, sp, #16
10088: e12fff1e bx lr
```

(a) ARM binary for Prog1



(b) CFG of the binary program

Fig. 2: ARM binary and corresponding CFG for Prog1

as sequences of pairs $(k, b) \in \mathbb{N} \times \mathbb{B}$ with the following meaning: k is the time it takes to process the instruction at p in the execution stage (E stage) of the pipeline; if b is true, fetching the instruction from the instruction cache results in a Hit otherise it is a Miss. This tranformation will give an annotated run $\tilde{\rho} = (2, \text{TRUE}).(1, \text{FALSE}).(2, \text{TRUE}).(2, \text{FALSE}).(1, \text{FALSE})$.

As mentioned earlier, it is noticeable that the hardware model needed to compute the execution time of a run is much simpler than the actual concrete hardware model: there is no need to model the actual processing unit (e.g., registers, memory) nor to perform actual computations (e.g., execute instructions).

Formal hardware model. As a sequence $\tilde{\rho} \in \mathcal{L}^a_H(P)$ contains enough information to compute the execution time of a program run $\rho \in \mathcal{L}_H(P)$ we can define an abstract model of the hardware as a timed automaton *transducer*, $Aut(H)$, that maps each $\tilde{\rho} \in \mathcal{L}^a_H(P)$ to a positive natural number $Aut(H)(\rho)$, which is the execution time of ρ on H. Hence the WCET of a program P on the hardware H is defined by:

$$\mathsf{WCET}(P, H) = \max_{\sigma \in \mathcal{L}^a_H(P)} Aut(H)(\sigma). \tag{2}$$

As $\mathcal{L}^a_H(P)$ over-approximates the set of program runs, we ensure that the value of the WCET we compute (equation (2)) is an upper bound of the actual WCET (this assumes that the hardware model $Aut(H)$ correctly models the timing behaviour of the hardware).

Modular computation of the WCET of a program. In practice to compute $\mathsf{WCET}(P, H)$ we need to provide a generator for $\mathcal{L}^a_H(P)$ and the model of the hardware $Aut(H)$. $\mathcal{L}^a_H(P)$ can be generated by a finite state automaton $Aut(P)$ (see [7, 9]). In general $\mathcal{L}^a_H(P)$ is a finite set of runs and can be defined by a finite computation tree.

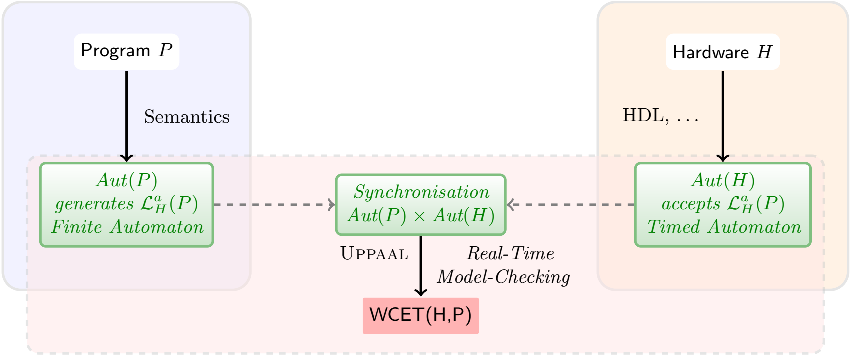

Program P
Semantics
$Aut(P)$ generates $\mathcal{L}^a_H(P)$ Finite Automaton
Hardware H
HDL, ...
$Aut(H)$ accepts $\mathcal{L}^a_H(P)$ Timed Automaton
Synchronisation $Aut(P) \times Aut(H)$
Uppaal
Real-Time Model-Checking
WCET(H,P)

Fig. 3: Modular Computation of WCET

In [7, 9] the modular computation of the WCET depicted in Fig. 3 is fully implemented in Uppaal as follows:

– a UPPAAL automaton, $Aut(P)$, that generates $\mathcal{L}^a_H(P)$ is computed based on the control flow graph of a program (for an ARM architecture.)
– the hardware model is provided for a given architecture (ARM920T). It comprises of a model of the pipeline and a model for the caches (complete model with the current state of the caches.) Notice that our method is robust against the so-called *timing anomalies* [10].
– the WCET can be computed either using a binary search or using UPPAAL *sup* operator.

This implementation has several drawbacks:

– the automaton $Aut(P)$ that generates $\mathcal{L}^a_H(P)$ is implemented using a limited C-like language. This is sometimes cumbersome and the semantics of some instructions had to be partially modelled (e.g., some bitwise operations on registers). The result is that the UPPAAL model of the program which is a finite automaton, is hard to encode using UPPAAL restricted set of C supported operations. This set was sufficient to model a large set of instructions of the ARM920T processor but may be too limited to model the semantics of more complex processors.
– the FIFO caches (instruction and data) are modelled precisely using an array to model the lines in the caches. The hardware model $Aut(H)$ contains the full state of the caches. This makes the discrete part of the state of the system $Aut(P) \times Aut(H)$ very large and impacts the efficiency of the model-checking algorithm.

In the next section we describe how to overcome the previous limitations by having $\mathcal{L}^a_H(P)$ generated by a C-library outside UPPAAL.

3 WUPPAAL

Program computation tree. In this section we assume that $\mathcal{L}^a_H(P)$ is available and represented as a finite tree $\mathsf{Tree}^a_H(P)$. This is based on the assumption that the number of iterations in the loops do not depend on an (arbitrary) input parameter. This is a usual assumption[4] in the WCET methods [23] as otherwise the WCET may be unbounded. Fig. 4 shows a subtree of $\mathsf{Tree}^a_H(Prog1)$. We use a *sliced* version of the binary program when we build the tree. This sliced version is equivalent WCET-wise [9, 6] to the actual program. The components $\mathcal{M}^i$ in Fig. 4 provides the values of the variables that are in the slice (some registers and other memory cells).
The following operations can be performed on $\mathsf{Tree}^a_H(P)$:

– *get_init*() returns the root of the tree $\mathsf{Tree}^a_H(P)$.
– *get_next*(n) returns the list of children of the node n (empty if n is a leaf).
– *hit_ins*(n) is a boolean that indicates whether the instruction to be executed at n will result in a hit or a miss in the instruction cache.

[4] An exact test for this assumption does not exist as this problem is undecidable.

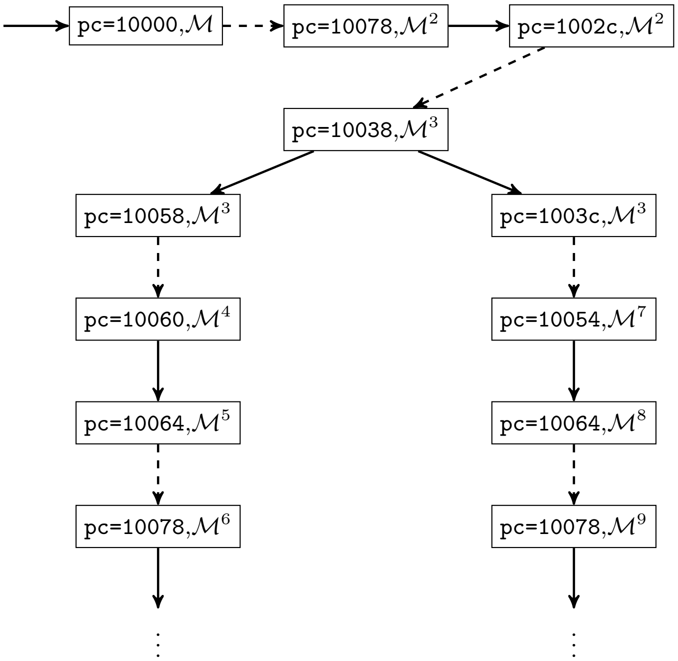


Fig. 4: subtree of $\mathsf{Tree}^a_H(Prog1)$ where we let $\mathcal{M}$ be the memory tracked, and $r3 = 10$ in $\mathcal{M}^2$. Dashed arrows indicate sequences of deterministic instructions omitted for brevity.

– *get_exec*(n) returns the execution (in cycles) in the E stage of the pipeline for the instruction at n.

We refer to these operations as the *tree-API* in the sequel. The implementations of the Tree-API operations live outside Uppaal in the library `libgdb2uppaal` (see Section 4 for the Wuppaal architecture). The Uppaal template in Fig. 5 implements a full search on $\mathsf{Tree}^a_H(P)$ given the *get_init*$()$ and *get_next*(n) functions; we assume each node of the tree has at most 2 children for the sake of simplicity. The Uppaal version of *get_init*$()$ is `get_init(succ)` and fills in the vector `succ` with the pair $(\mathit{get\_init}(), \bot)$ ($\bot$ denotes the absence of node). Similarly *get_next*(n) is implemented by the function `get_next(n,succ)` and fills in the vector of integers `succ` with the children of n where `succ[0]` (resp. `succ[1]`) is the first (resp. second) child of n; the $\bot$ value is represented by a negative integer. The non-deterministic guarded choices in the template Program Automaton (Fig. 5) push the children nodes to be processed to the first stage of the pipeline (see hardware model below). Each path through the template Program Automaton from the initial location (double circle) to the `END` location represents an annotated trace of $\mathcal{L}^a_H(P)$. When we model-check a safety property on this model, Uppaal generates all the traces in $\mathcal{L}^a_H(P)$.

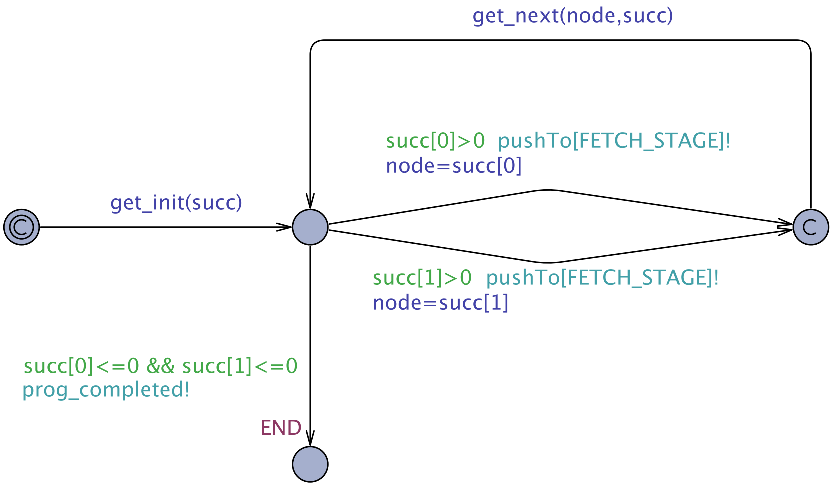


Fig. 5: Program Automaton to enumerate $\mathcal{L}^a_H(P)$.

Hardware specification. The hardware consists of a multi-stage execution pipeline and the caches (e.g., instruction and data caches). As a case-study we model an ARM920T 5-stage *execution pipeline*, the instruction cache and main memory components. The pipeline can execute concurrently the different stages (Fetch, Decode, Execute, Memory, Writeback) needed to fully process an instruction. An instruction is fetched (from the instruction cache) in stage F, decoding and operand register accesses occur in D, execution in E and if there are load/store instructions the memory accesses happen in M. The results are written back to registers in W. The (normal) flow of instructions in the pipeline is shown in Fig. 6. This optimal flow may be slowed down when pipeline *stalls* occur: if the instruction $i+1$ needs a register written to by instruction i there will be a one cycle stall at cycle $j+3$ for instruction $i+1$; when the W stage is finished for instruction i, the E stage can begin for instruction $i+1$.

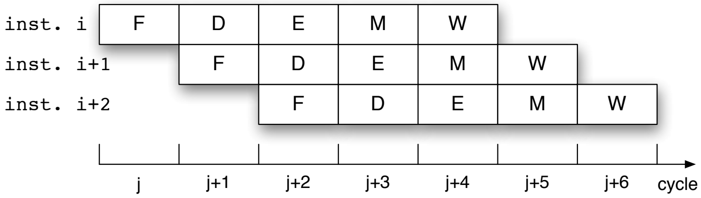
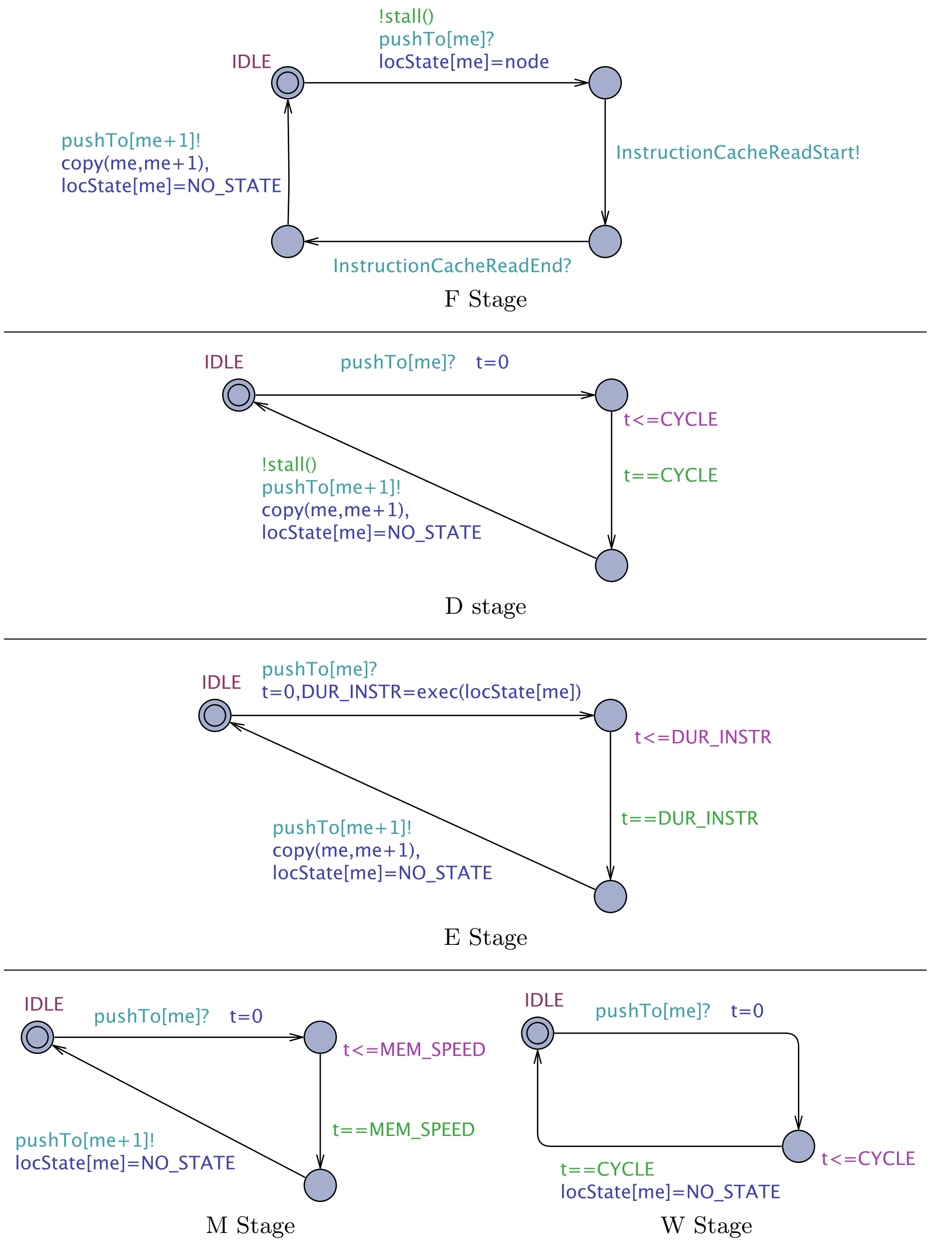


Fig. 6: Pipeline of the ARM920T

Hardware abstract model. A formal model of the hardware for the ARM920T can be specified by a network of timed automata [9]. We provide here simpler models of the hardware because we factor out the actual state of the caches: to compute the execution time of a sequence of instructions we only need to know whether a transaction with a cache is a hit or a miss. This information is provided by each node in $\mathsf{Tree}^a_H(P)$ ($\mathcal{L}^a_H(P)$) for a given program P. It can be computed by monitoring the addresses that are used on a given trace and using a odel of the caches (e.g., number of lines, ways and FIFO replacement policy). In [8] we also proposed an abstraction/refinement scheme to model the caches. For instance the 5-stage pipeline of the ARM920T can be specified by a network of 5 timed automata (see Fig. 7) each of them modelling a single stage of the execution pipeline.

Each stage automaton has a unique identifier `me` (an integer). The values of this identifier for the templates (F, D, E, M, W) are respectively (0,1,2,3,4). This encodes the fact that the stages F, D, E, M, W are ordered: each node of $\mathsf{Tree}^a_H(P)$ flows from one stage k to the next $k+1$ when the `pushTo[k]` channels synchronise. For instance, the F-Stage template automaton is idle until the Program Automaton (Fig. 5) pushes a node via the `pushTo[0]?` transition. It updates the local *state* of this stage 0 (`locState[0]=node`) where `node` is a (meta) variable used to retrieve the value sent by the Program Automaton that issues the `pushTo[0]!` command. The F stage template automaton then synchronises with the instruction cache (see Fig. 7) to simulate the time it takes to fetch the instruction from the instruction cache.

The memory stage (M stage) assumes a constant time to read data from the data cache: each transaction takes `MEM_SPEED` cycles. We can easily model



F Stage

D stage

E Stage

M Stage

W Stage

Fig. 7: Timed Automata for F, D, E, M and W Stages (pipeline ARM920T).

the data cache but for the sake of simplicity we use a simple version here. The other stages (D, E, M, W) are based on the same logic: they are idle until the previous stage pushes some information to them. The `copy(me,me+1)` commands transfers the information from stage `me` to stage `me+1`. When going back to the `IDLE` (initial) location, the local information of the templates are reset to the default value `NO_STATE` which indicates that the pipeline state is empty.

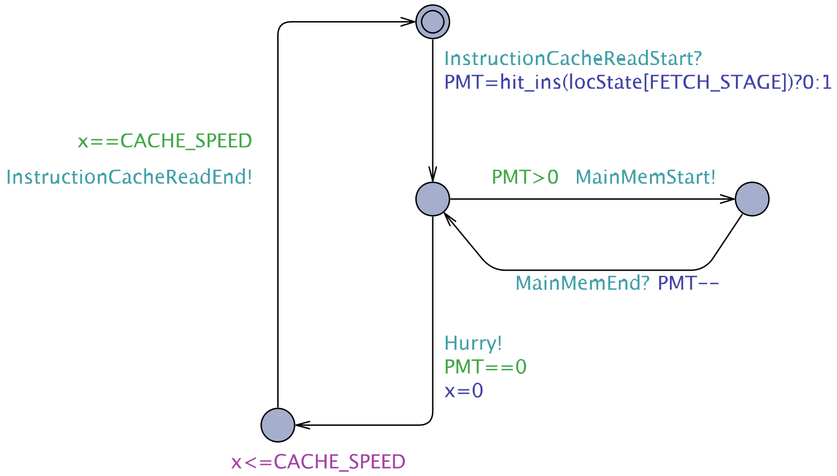


Fig. 8: Instruction cache template.

The instruction cache is specified by the template timed automaton in Fig. 8. The `PMT` variable holds the number of Pending Memory Transactions. This number is determined by the `hit_ins` function that can be retrieved from the annotated node in the tree.

Finally the main memory template simply simulates how long it takes to perform a transaction (read or write) with the main memory.

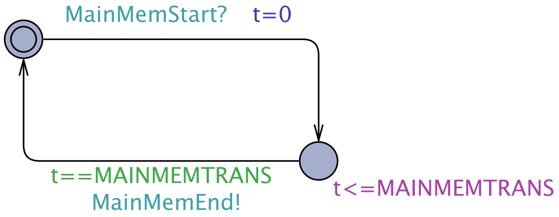


Fig. 9: Main memory template.

4 Implementation and experimental results

Tool chain. Let us dwell on the tool chain we have constructed to demonstrate our methodology described in Section 3. The toolchain, visualized in Fig. 10, is composed of five components:

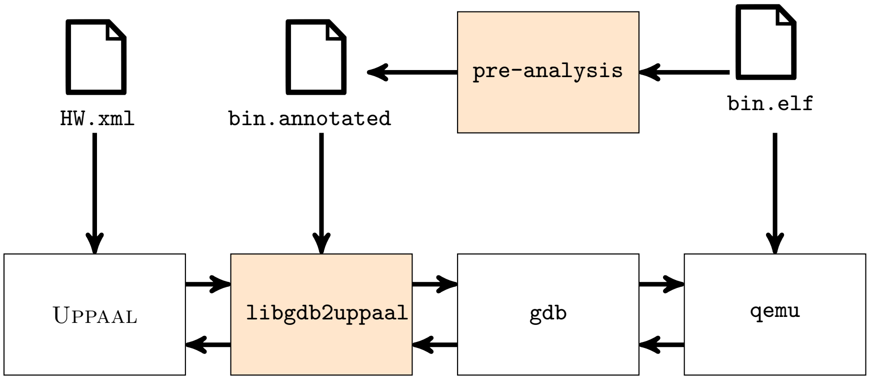


Fig. 10: The tool chain of WUppaal. Orange blocks are the modules we implemented. Other blocks are existing modules.

- a `pre-analysis` module for constructing an annotated program that can be used to generate the program traces $\mathcal{L}_H(P)$; this step is developped in Scala and uses some powerful Grammar and Language Processing packages Kiama [20] and Sbt-Rats! [21].
- `qemu` [4] to emulate the chosen hardware and enables us to compute the next state after executing a program instriuction. As an example of usage, we set up the hardware to a given initial state (program counter and valkues of registers and stack), and with `qemu` we can compute the effect of an instruction. What is communicated back (using `gdb`) is the next program counter and the next state of the registers and stack.
- `gdb` [22] for inspecting `qemu`,
- `libgdb2uppaal` to implement the tree-API given at the beginning of Section 3.
- a Timed Automaton model of the hardware `HW.xml` (an example is provided on Fig. 7, page 13 for the pipelines and Fig. 8, page 14 and Fig. 9, page 14 for the main memory and instruction cache.)
- Uppaal for computing the worst-case execution time given a sequence of nodes using the Program Automaton template Fig. 5, page 11, The Uppaal model uses an integer counter to identify the current state of the program. The `libgdb2uppaal` maintains a table that maps integers to actual program states (program counter, values of the registers and the stack). The `get_next` function in the Tree-API returns all the possible successors of a state as

integers and updates the table that maps integers to program state (when a new state is encountered). The Program Automaton (Fig. 5) will explore all the successor states.

Computing the WCET for a given binary program `bin.elf` using our framework is a two-stage process. In the first stage we compute an annotated program (e.g., a CFG and the set of variables needed to generate the annotated language $\mathcal{L}^a_H(P)$) by using `pre-analysis`.) In the second stage we use Uppaal to drive a search through the state space, interfacing (by proxy of `gdb` and `libgdb2uppaal`) with the emulator of the hardware as described in Section 3. In the current model, we ignore the data cache but this is not a restriction as the caches can be added to the program state and modeled in the `libgdb2uppaal` library.

Support for other languages and hardware. The approach we propose is general enough to accommodate other languages and hardware. For instance, assume we want to use an x86 processor and the corresponding assembly language. What needs to be provided is a new `pre-analysis` module for this assembly language to construct the annotated program. The pre-analysis we have developed for the ARM assembly language is easy to re-use to build support for other languages.

We also need to provide an abstract model for the x86 hardware as a network of timed automata. The widgets we have proposed in Section 2 for the ARM920T pipeline can be adapted to build new formal models for an x86 platform (and of course new pipeline stages can be added if the architecture requires it).

Finally we need `qemu` (or an equivalent program) to support the emulation of the hardware. The general architecture we introduced in Fig. 10 can be re-used as well as the the modular method depicted in Fig. 3 to compute the WCET for programs running on the x86.

Results. We have experimented our technique using some of the standard benchmarks [17] from Mälardalen University, for computing WCET. As we can see in Table 1, we are achieving a reasonable computation time (less than 5 seconds for all experiments), demonstrating the feasibility of our approach. We can also see that for all of the test-cases, the constructed trees are fairly small in size. In this paper we do not provide a thorough comparison with the actual measured execution times because we use simple models for the caches. The models used in [9] may be used in the future. The results in [9] demonstrated that our approach provides very accurate WCET and the new implementation should give similar results when precise models of the caches are used.

5 Conclusion

We have presented a method, based on timed automata and real-time model-checking with Uppaal, to compute the WCET of binary programs. The method we designed is generic and can accommodate arbitrary hardware. The proposed tool chain allows us to achieve a modular approach to WCET-computation,

Program	**Loc**	$\lvert \mathsf{Tree}^a_H(P) \rvert$	**Time**	**WCET**
duff	145	1750	4.51	61215
fibcall	48	553	2.91	19320
insertsort	84	7	2.09	210
janne_complex	67	360	3.21	12565
lcdnum	100	250	2.52	8715

Table 1: The experimental results, time is given in seconds and includes startup overhead from initializing `gdb` and `qemu`. The *loc* measure is the number of lines of assembly. Note that more experiments will be added by the final submission.

reducing the overhead needed to support new binaries, and new architectures. To support different binaries we only have to provide `pre-analysis` with a different input. To support different processors, it is sufficient to provide a new hardware-model (`HW.xml`) and emulator (`qemu`).

Moreover, our technique does not rely on the computation of *loop bounds* or the assumption that the hardware is free of *timing anomalies*: this is one of the strengths of the model-checking method. Another strength is that it generates a witness program trace that produces the WCET. Other interesting features of this approach includes its generality: we do not need to assume that the initial state of the caches is known. The only requirement is that the annotated language $\mathcal{L}^a_H(P)$ over-approximates the program behaviours.

Our technique is also general enough to be paired with *program refinement* techniques. As mentioned in Section 3 for Prog1, some traces in $\mathcal{L}_H(P)$ may not be feasible: if the first choice for the test $a < b$ is TRUE (resp. FALSE), the following test of the same condition must be TRUE (resp. FALSE). In that case we compute a refinement $R_1 \subseteq \mathcal{L}^a_H(P)$ of the annotated program to rule the spurious traces and analyse the refinement R_1. This can de done using the *trace abstraction approach* of [14, 15]. This enables us to define an *iterative* method to compute better and better over-approximations of the WCET and ensure that one witness trace exists.

Notice that this refinement also applies to the hardware model: we can start with a very simple model of the caches where every transaction is either a Hit or a Miss. Once a WCET is computed with UPPAAL, we can check whether the witness trace is feasible in the program *and* in the caches. If the cache behaviour that is in the witness is spurious (infeasible) we can refine it as well. We have implemented a cache refinement technique in [8]. This enables us to get some control on the accuracy of the computation via model-checking.

On another note, we can use our technique as a simulation based technique: the `bin.annotated` component in the tool chain Fig. 10 can be replaced by a generator of traces. In this case we can only compute a lower bound for the WCET but we get access to the *statistical model-checking* engine of UPPAAL. This opens a new avenue to compute some probabilitic distributions of the WCET.

In addition, outsourcing the semantics of a binary program to a trusted emulation tool (qemu) eliminates errors that occurs when semantically translating binary programs into timed automata. As such a translation necessitates a very high level of detail, it can easily result in a state-space explosion – even for simple architectures and programs. With our construction, knowledge of the hardware and static-analysis and abstraction refinement methods can be used to reduce the size of explored state space.

References

1. AbsInt Angewandte Informatik: aiT Worst-Case Execution Time Analyzers., http://www.absint.com/ait/
2. Alur, R., Dill, D.: A theory of timed automata. TCS 126(2), 183–235 (1994)
3. Behrmann, G., David, A., Larsen, K.G., Håkansson, J., Pettersson, P., Yi, W., Hendriks, M.: Uppaal 4.0. In: QEST. pp. 125–126. IEEE Computer Society (2006)
4. Bellard, F.: Qemu, a fast and portable dynamic translator. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference. pp. 41–41. ATEC '05, USENIX Association, Berkeley, CA, USA (2005), http://dl.acm.org/citation.cfm?id=1247360.1247401
5. Bernat, G., Colin, A., Petters, S.M.: pWCET a Toolset for automatic Worst-Case Execution Time Analysis of Real-Time Embedded Programs. In: Proceedings of the 3rd Int. Workshop on WCET Analysis, Workshop of the Euromicro Conference on Real-Time Systems. Porto, Portugal (2003)
6. Cassez, F.: Timed Games for Computing Worst-Case Execution-Times. Research report, National ICT Australia (Jun 2010), 31 pages. Available from http://arxiv.org/abs/1006.1951
7. Cassez, F.: Timed Games for Computing WCET for Pipelined Processors with Caches. In: ACSD'11. pp. 195–204. IEEE Comp. Soc. (Jun 2011)
8. Cassez, F., de Aledo Marugán, P.G.: Timed automata for modelling caches and pipelines. In: van Glabbeek, R.J., Groote, J.F., Höfner, P. (eds.) Proceedings Workshop on Models for Formal Analysis of Real Systems, MARS 2015, Suva, Fiji, November 23, 2015. EPTCS, vol. 196, pp. 37–45 (2015)
9. Cassez, F., Béchennec, J.: Timing analysis of binary programs with UPPAAL. In: 13th International Conference on Application of Concurrency to System Design, ACSD 2013. pp. 41–50. IEEE Computer Society (Jul 2013)
10. Cassez, F., Hansen, R.R., Olesen, M.C.: What is a timing anomaly? In: 12th International Workshop on Worst-Case Execution Time Analysis, WCET 2012, July 10, 2012, Pisa, Italy. OASICS, vol. 23, pp. 1–12. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2012)
11. Dalsgaard, A.E., Olesen, M.C., Toft, M., Hansen, R.R., Larsen, K.G.: Metamoc: Modular execution time analysis using model checking. In: Lisper, B. (ed.) WCET. OASICS, vol. 15, pp. 113–123. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany (2010)
12. Dalsgaard, A.E., Olesen, M.C., Toft, M.: Modular Execution Time Analysis using Model Checking. Master's thesis, Dpt. of Computer Science, Aalborg University, Denmark (2009)
13. Ferdinand, C., Heckmann, R., Wilhelm, R.: Analyzing the worst-case execution time by abstract interpretation of executable code. In: Broy, M., Krüger, I.H., Meisinger, M. (eds.) ASWSD. LNCS, vol. 4147, pp. 1–14. Springer (2004)

14. Heizmann, M., Hoenicke, J., Podelski, A.: Refinement of trace abstraction. In: Palsberg, J., Su, Z. (eds.) SAS. LNCS, vol. 5673, pp. 69–85. Springer (2009)
15. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Sharygina, N., Veith, H. (eds.) CAV. LNCS, vol. 8044, pp. 36–52. Springer (2013)
16. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a Nutshell. Journal of Software Tools for Technology Transfer (STTT) 1(1-2), 134–152 (1997)
17. Mälardalen WCET Research Group: WCET Project – Benchmarks., http://www.mrtc.mdh.se/projects/wcet/benchmarks.html
18. Rapita Systems Ltd.: Rapita Systems for timing analysis of real-time embedded systems., http://www.rapitasystems.com/
19. Rieder, B., Puschner, P., Wenzel, I.: Using Model Checking to Derive Loop Bounds of General Loops within ANSI-C Applications for Measurement Based WCET Analysis. In: 6th Int. Workshop on Intelligent Solutions in Embedded Systems (WISES'08). Regensburg, Germany (2008)
20. Sloane, A.M.: Lightweight Language Processing in Kiama. In: Generative and Transformational Techniques in Software Engineering III, pp. 408–425. No. 6491 in Lecture Notes in Computer Science, Springer Berlin Heidelberg (Jan 2011), http://link.springer.com/chapter/10.1007/978-3-642-18023-1_12
21. Sloane, A., Cassez, F., Buckley, S.: The Sbt-rats parser generator plugin for scala (tool paper). In: Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala. pp. 110–113. SCALA 2016, ACM, New York, NY, USA (2016), http://doi.acm.org/10.1145/2998392.3001580
22. Stallman, R., Pesch, R., Shebs, S., et al.: Debugging with gdb. Free Software Foundation 51, 02110–1301 (2002)
23. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D.B., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P.P., Staschulat, J., Stenström, P.: The Worst-Case Execution-Time Problem - Overview of Methods and Survey of Tools. ACM Trans. Embedded Comput. Syst. 7(3) (2008)