Timed Games for Computing WCET for Pipelined Processors with Caches

Franck Cassez*
CNRS/IRCCyN, Nantes, France

***Abstract*—We introduce a framework for computing upper bounds of WCET for hardware with caches and pipelines. The methodology we propose consists of 3 steps: 1) given a program to analyse, compute an equivalent (WCET-wise) abstract program; 2) build a timed game by composing this abstract program with a network of timed automata modelling the architecture; and 3) compute the WCET as the optimal time to reach a winning state in this game. We demonstrate the applicability of our framework on standard benchmarks for an ARM9 processor with instruction and data caches, and compute the WCET with UPPAAL-TiGA. We also show that this framework can easily accommodate dynamic changes in the speed of the processor during program execution.**

I. Introduction

Embedded real-time systems are composed of a set of tasks (software) that run on a given architecture (hardware). These systems are subject to strict timing constraints that must be enforced by a scheduler. Designing an effective scheduler is possible only if some bounds are known about the execution times of each task. Performance wise, determining tight bounds is crucial as using rough over-estimates might either result in a set of tasks being wrongly declared non schedulable, or a lot of computation time might be wasted in idling cycles and loss of computing power.

The WCET Problem. The *execution-time*, $\mathsf{time}(p, d, H)$, of a program p, with input data d on the hardware H, is measured as the number of cycles of the fastest component of the hardware i.e., the processor. Data take their values in a finite domain $\mathcal{D}$. The program is given in binary code or equivalently in the assembly language of the target processor[1]. The *worst-case execution-time* of program p on hardware H is defined by:

$$\text{WCET}(p, H) = \sup_{d \in \mathcal{D}} \mathsf{time}(p, d, H). \tag{1}$$

The WCET problem asks the following: Given p and H, compute $\text{WCET}(p, H)$.

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 obviously computable. Indeed the possible runs of the program can be represented by a finite tree. Notice that this does not mean that the problem is tractable though.

If the input data are known or the program execution time is independent from the input data, the tree contains a single path and it is usually feasible to compute the WCET. Likewise, if we can determine some input data that produces the WCET (but this might be as difficult as computing the WCET itself), we can compute the WCET on a single-path program.

If is not often the case that the input data are known or that we can determine an input that produces the WCET. Rather the (values of the) input data are unknown, and the number of paths to be explored might be extremely large: for instance, for a Bubble Sort program with 100 data to be sorted, the tree representing all the runs of the (assembly) program on all the possible input data has more than 2^{50} nodes. Although symbolic methods (e.g., using BDDs) can be applied to analyse some programs with a huge number of states, they will fail to compute the exact WCET on Bubble Sort by exploring all the possible paths.

Another difficulty of the WCET problem stems from the more and more complex architectures embedded real-time systems are running on. They feature a multi-stage *pipeline* and a fast memory component like a *cache*, and they both influence in a complicated manner the WCET. It is then a challenging problem to determine a precise WCET even for relatively small programs running on complex architectures.

Methods and Tools for the WCET Problem. The reader is referred to [1] for an exhaustive presentation of the WCET computation techniques and tools. There are two main classes of methods for computing WCET:

- Testing-based methods. These methods are based on experiments i.e., running the program on some data, using a simulator of the hardware or the real platform. The execution time of an experiment is measured and, on a large set of experiments, a maximal and minimal bounds can be obtained. The maximal bound computed this way is *unsafe* as not all the possible paths have been explored. These methods might not be suitable for safety critical embedded systems but they are versatile and rather easy to implement.
  RapiTime [2] (based on pWCET [3]) and Mtime [4] are measurement tools that implement this technique.
- Verification-based methods. These methods often rely on the computation of an *abstract* graph, the control flow graph (CFG), and an abstract model of the hard-

*Author supported by a Marie Curie International Outgoing Fellowship within the 7th European Community Framework Programme.

[1]When we refer to the "source" code, we assume the program p was generated by a compiler, and refer to the high-level program (e.g., in C) that was compiled into p.

ware. Together with a static analysis tool they can be combined to compute WCET. The CFG should produce a superset of the set of all feasible paths. Thus the largest execution time on the abstract program is an upper bound of the WCET. Such methods produce *safe* WCET, but are difficult to implement. Moreover, the abstract program can be extremely large and beyond the scope of any analysis. In this case, a solution is to take an even more abstract program which results in drifting further away from the exact WCET.

Although difficult to implement, there are quite a lot of tools implementing this scheme: Bound-T [5], OTAWA [6], TuBound [7], Chronos [8], SWEET [9] and aiT [10], [11] are static analysis-based tools for computing WCET.

The verification-based tools mentioned above rely on the construction of a control flow graph, and the determination of loop bounds. This can be achieved using user annotations (in the source code) or sometimes inferred automatically. The CFG is also annotated with some timing information about the cache misses/hits and pipeline stalls, and paths analysis is carried out on this model e.g., by Integer Linear Programming (ILP). The algorithms implemented in the tools use both the program and the hardware specification to compute the CFG fed to the ILP solver. The architecture of the tools themselves is thus monolithic: it is not easy to adapt an algorithm for a new processor. This is witnessed by *WCET'08 Challenge Report* [12] that highlights the difficulties encountered by the participants to adapt their tools for the new hardware in a reasonable amount of time.

WCET and Model-Checking. Surprisingly enough, only a few tools use model-checking techniques to compute 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 *synchronization* of these components depends on *timing constraints* (time to execute in one stage of the pipeline, time to fetch a data from the cache), formal models like *timed automata* [13] and state-of-the-art *real-time model-checkers* like UPPAAL [14], [15] appear well-suited to address the WCET problem.

It has previously been claimed [16] that *model-checking* was not adequate to compute WCET, but this statement has since been revised. In [17], A. Metzner showed that model-checkers could well be used to compute safe WCET on the CFG for programs running on pipelined processors with an instruction cache.

In [18], B. Huber and M. Schoeberl consider Java programs and compare ILP-based techniques with model-checking techniques using the model-checker UPPAAL. Model-checking techniques seem slower but easily amenable to changes (in the hardware model). The recommendation is to use ILP tools for large programs and model-checking tools for code fragments.

More recently, the TASM toolset [19] (M. Ouimet & K. Lundqvist) has been used to compute WCET with UPPAAL: the TASM machine is a high level machine not featuring pipelining nor caches.

Related Work. The use of timed automata (TA) and the model-checker UPPAAL for computing WCET on pipelined processors with caches is reported in [20], [21] (METAMOC, A. E. Dalsgard *et al.*). The METAMOC method consists of: 1) computing a flow graph (FG) from a binary program, 2) composing this FG with a (network of timed automata) model of the processor and the caches. Computing the WCET is then reduced to computing the longest path in the network of TA.

The previous framework is very elegant yet has some shortcomings: (1) METAMOC relies on a value analysis phase that may not terminate, (2) some programs cannot be analysed (if they contain register-indirect jumps), (3) some manual annotations are still required on the binary program, e.g., loop bounds and (4) the *unrolling* of loops is not safe for some cache replacement policies (FIFO).

Our Contribution. In this paper we use *timed game automata* (TGA) and UPPAAL-TiGA [23] (UPPAAL for timed games) to compute WCET. We model the WCET problem as a two-player timed game: Player 1 is the program, and Player 2 is in charge of deciding the outcome of the *comparison* instructions (e.g., `cmp, tst` that set the conditional branching conditions) that depend on the input data. As the choice of the input data is not controllable by Player 1, we obtain a two-player game. The problem we solve on this game is an *optimal time reachability problem*:

> "What is the optimal time for Player 1 to reach the end of the program ?"

What is similar to METAMOC is the use of network of timed automata to model the cache[2] and pipeline stages. To obtain a model (automaton) of the program we use a radically different approach: (1) we build a graph without any need for annotations and (2) we propose a new and very compact encoding of the program and pipeline stages' states whereas METAMOC uses a values analysis phase and requires loop bounds annotations to obtain an (unfolded) graph of the program. First this enables us to compute the WCET for 14 programs[3] (see Table I) of the Mälardalen University benchmark programs with concrete data and instructions caches, including some programs that cannot be analyzed by METAMOC (although METAMOC reports WCET results for 21 programs of the same benchmark). Second, compared to METAMOC that requires a computer with 32GB RAM and 2.5Ghz Quad Core, we can compute the results on a laptop computer, 2Ghz Dual Core, with 2GB RAM, within a few seconds (only two programs require more than one

[2]Note that a similar model is reportedly due to A. P. Ravn in [18].

[3]The remaining programs contain assembly instructions that are currently not supported by our compiler from ARM assembly language to UPPAAL.

Program	loc†	N‡	UPPAAL-TiGA time/space¶	METAMOC time£	WCET	Abs§	Low Power
Single-Path Programs							
fac	26	0	0.35s/6.91MB	1.35s	1883	4/34	26/1.3%
fib	74	0	0.25s/5.68MB	1.52s	571	4/22	26/4.5%
janne-complex*	65	0	0.54s/7.76MB	1.92s	792	0/23	**176/22%**
matmult*	162	0	119.2s/936.75MB	Out of Mem.	614827	**31/107**	800/0.001%
jfdcint	374	0	7.13s/55.99MB	16min46s	49017	**394/454**	108/0.22%
expint(50,1)	81	0	6.08s/59.16MB	12s	65042	0/124	70/1.7%
expint(50,21)	81	0	3.65s/43.21MB	12s	41015	0/124	71/1.7%
fdct	238	0	2.83s/26.79MB	41m22	26099	0/286	90/0.3%
edn*	284	0	22.28s/230.98MB	NA	62968	0/460	26/0.04%
recursion*	41	0	2.68s/28.82MB	NA	10335	0/38	32/0.3%
Multiple-Paths Programs							
bs	174	5	0.52s/6.52MB	0.52s	366	0/22	**30/8.2%**
cnt*	115	100	100.25s/377.02MB	4s	6483	0/82	40/0.06%
insertsort*	91	675	9.36s/81.27MB	2.44s	27061	0/53	400/1.4%
ns*	497	625	12.38s/110.92MB	7s	43239	0/41	32/0.0007%

†lines of code in the C source file
‡N = Max number of Player 2 moves along a path
§Abstracted instructions/Instructions
*Program selected for the WCET Challenge 2006 [22]
¶On Intel Dual Core 2Ghz 2GB RAM (FIFO Caches)
£On Intel Quad Core 2.5Ghz 32GB RAM (LRU Caches)

Table I
RESULTS (C PROGRAMS COMPILED WITH `GCC -O2`)

hundred seconds). Third, using timed games instead of timed automata is also notably different: (i) the on-the-fly algorithm [24] implemented in UPPAAL-TiGA [23] is different from the one running in UPPAAL, and it can also compute the *optimal time* (in the presence of adversary) to reach a designated state; thus we do not need to do a binary search or use a tailored version of UPPAAL to compute the results; and (ii) solving a game allows us to prove that the program always *terminates* which validates the model of hardware (pipeline and caches) we use in the sense that they produce no deadlocks.

Finally, we also show that taking into account processor speed variations is easy in a framework using timed games. This can be important as it is possible to adjust the speed of the processor depending on the program to be run. For some programs, the saved power can be up to 22% (see Table I and Section V for detailed comments).

Outline of the Paper. In Section II, we briefly introduce the ARM9 architecture and the assumptions we make on the assembly programs to be analysed. Section III describes how to encode an assembly program with non-deterministic choices into a game. In Section IV we give the timed automata models of the architecture we use to compute the WCET. Section V gives an overview of the tool chain we propose and the components (compiler) we have designed together with some comments on the case studies presented in Table I. [25] is an extended version of this paper.

II. ASSEMBLY PROGRAMS AND ARM9 ARCHITECTURE

Runs of a Program. A *program* p is a list of instructions $p = i_1, i_2, \cdots, i_r$ and i_1 is the initial instruction. The control usually goes from instruction i_j to i_{j+1} except for branching instructions that give the next instruction i_b to be performed. Each instruction performs some basic operations (arithmetic, logic, memory load or store, branching) and has a duration which gives the amount of time it takes in each stage of the pipeline of the processor[4]. We assume this duration is independent from the content of the operands of the instructions[5]. In the sequel we use the variable ι to denote an instruction of p.

The hardware H on which p runs has a pool of *registers* (different from the main memory and the caches). We let $\mathcal{R} = \{r_0, \cdots, r_k\}$ be the set of registers of H. For example on the ARM9 processor [26] there are 16 registers. A designated register **pc** (register 15 on the ARM9) contains the *program counter* and points to the next instruction to be performed.

We let $\mathcal{M} = \{m_1, m_2, \cdots, m_n\}$ be the set of memory cells' addresses used by the program (we assume the program can access $\mathcal{M}$). The content of the memory cells and registers is in a finite domain $\mathcal{D}$ (e.g., 32 bit integers). A *state* of a computation of p is a mapping $v : \mathcal{M} \cup \mathcal{R} \to \mathcal{D}$. We let $\mathcal{V}$ be the set of states and $\mathbb{B} = \{\text{TRUE}, \text{FALSE}\}$.

As we do not want to simulate p for each single possible value of each input data, we assume that the initial values of these data can be arbitrarily chosen and thus can have an arbitrary and unknown value. To formalise this, we use an extended domain for the values of the registers and memory cells: $\mathcal{D} \cup \{\perp\}$ where $\perp$ is a special *unknown* value. $\mathcal{V}_\perp$ denotes the set of states on the extended domain. At the beginning of the computation, every register (except **pc**) and every memory cell is set to $\perp$. The initial state

[4]A particular case is a processor with one stage and 1 time unit for each instruction.

[5]This is not always the case as for instance the duration of the instruction `mull` (multiplication on long integers) on the ARM9 depends on how large one of the operand is. However, we can always take the longest duration to obtain a safe upper bound of the WCET.

of the computation of p is thus v_0 with $v_0(x) = \perp$ for $x \in (\mathcal{R} \setminus \{\mathbf{pc}\}) \cup \mathcal{M}$ and $v(\mathbf{pc}) = i_1$ where i_1 is the address of the first instruction of program p.

An example of an assembly program is given in Listing 1. This program performs a binary search on an array of 14 elements. The address of an instruction appears on the left hand side of a line and we refer to an instruction using this address. Instruction at 24 loads register r3 with the content of memory cell at address $v(\mathsf{r4}) + (v(\mathsf{r2}) * 8)$. As we do not know the values stored in $\mathcal{M}$, the value of r3 is unknown after this instruction has been performed. r0 contains the value we are looking for in the search: it is set to 9 by the instruction at 0. The result of the comparison (`cmp r3,r0`) between the registers r0 and r3 performed by the instruction at 2c is *undetermined* as the value of r3 is unknown: thus setting the initial value of r0 to 9 does not restrict the set of paths defined using our extended domain values. The outcome of the comparison is used later in *conditional*[6] instructions (e.g., `ldreq r5, [r1, #4]` and `subgt ip,r2,#1`) and branching instructions `beq 44`. Two *status bits*[7] are needed to encode the result of the comparison of the instruction at 2c: whether r3 is "lower or equal" than r0 and whether r3 is "equal" to r0. This is indicated by the two predicates[8] eq and le between / . . . /.

```
00000000 <main>:
 0:  e3a00009 mov  r0, #9 ; 0x9
 4:  eaffffff b    8 <binary_search>

00000008 <binary_search>:
 8:  e92d4030 stmdb sp!, {r4, r5, lr}
 c:  e59f4040 ldr  r4, [pc, #64] ;
10:  e3a0e000 mov  lr, #0 ; 0x0
14:  e3a0c00e mov  ip, #14 ; 0xe
18:  e3e05000 mvn  r5, #0 ; 0x0
1c:  e08e300c add  r3, lr, ip
20:  e1a020c3 mov  r2, r3, asr #1
24:  e7943182 ldr  r3, [r4, r2, lsl #3]
28:  e0841182 add  r1, r4, r2, lsl #3
2c:  e1530000 cmp  r3, r0        / eq le /
30:  05915004 ldreq r5, [r1, #4]
34:  024ec001 subeq ip, lr, #1 ; 0x1
38:  0a000001 beq  44 <binary_search+0x3c>
3c:  c242c001 subgt ip, r2, #1 ; 0x1
40:  d282e001 addle lr, r2, #1 ; 0x1
44:  e15e000c cmp  lr, ip         / le /
48:  c1a00005 movgt r0, r5
4c:  dafffff2 ble  1c <binary_search+0x14>
50:  e8bd8030 ldmia sp!, {r4, r5, pc}
54:  00000158 .word 00000158
```

Listing 1. Binary Search Program

We assume that for each program p, when a memory cell is referenced, its actual address is known (although the content of it might be unknown). This address only depends on the path (sequence of instructions) that has been taken in the program from the initial state. This means that we rule out programs that can access unknown memory addresses that depend on the values of the input data. This is not an important restriction and this requirement was fulfilled by all the programs we have encountered so far.

A *run* of program p is a sequence of instructions. What is important to notice is that some instructions may have more than one outcome (e.g., comparisons) because the input data are considered to have an unknown value. Thus the set of runs of p on all possible input data is represented by a *tree*, $\mathsf{tree}(p)$. We make the assumption that all runs of p are *bounded*, and thus $\mathsf{tree}(p)$ has bounded depth: this is a common assumption when computing WCET and amounts to saying that all loops are bounded[9].

Execution Time of a Run. If each instruction was performed one after the other, the execution-time of a run would be the sum of the execution times of each instruction. On pipelined architecture, the execution of an instruction is split into two or more stages (fetch, decode, execute, memory and so on). On pipelined architectures with caches, the execution-time solely depends on:

1) the sub-sequences of instructions: pipeline *stalls* can occur, for instance because one instruction (e.g., in the execute stage) reads a register written to by the instruction in the next stage (e.g., memory stage).
2) the time to read/write a memory cell: instructions that perform memory transfers (load and store) might take different durations if a *cache* is used, depending on whether the memory cell is already in the cache or not.

The *duration* of a run ρ on architecture H, possibly featuring pipelining and caching, is denoted $\mathsf{time}(\rho, H)$. As each path $\rho \in \mathsf{tree}(p)$ corresponds to a run of p on some input data $d \in \mathcal{D}$ and conversely, and as $\mathsf{tree}(p)$ is finite, we can rewrite Equation **??** as follows:

$$\mathrm{WCET}(p, H) = \max_{\rho \in \mathsf{tree}(p)} \mathsf{time}(\rho, H).$$

This function might be rather complex (because of pipeline stalls, cache misses) but is yet well-defined.

III. From Programs to Games

In this section we describe how to encode an assembly program into a game.

Given a program p, we define a two-player game to model the runs of the program. Player 1 executes the instructions of p. The role of Player 2 is to set the values of the *status bits* that record the outcome of a comparison: when an instruction that modifies them is encountered and some operands have *unknown* values, the result is undetermined; the outcome is thus picked up non-deterministically.

On the ARM9 processor, there are 4 status bits. A simple encoding would be to have 4 boolean variables to model the

[6] A conditional instruction is executed only if the condition is true.

[7] On the ARM9 processor, bits V and Z.

[8] The needed predicates are computed by our program ARM2UPP, see section V.

[9] Notice that we do not require to annotate the program with loop bounds.

value of each bit. As we let Player 2 choose the outcome, this corresponds to choosing four values for Player 2: N (negative), Z (zero), V (overflow) and C (carry). This could create $2^4 = 16$ different next states in a computation and thus as many new potential branches in the game. Most of the time, it is not necessary to know the actual values of the 4 status bits. For instance the result of a comparison instruction cmp r0, r1 with, say r1 unknown, could be used later on only to check whether r0 = r1. In this case the value of the Z-status bit is required but the values of the other status bits are irrelevant.

To reduce the number of branches (choices of Player 2) in the game, we determine, for each instruction ι of p that sets a status bit, the next instructions that depend on the outcome of ι. This can be automatically computed on the program p. For each instruction ι that sets a status bits, we let *flags*(ι) be the set of predicates used after ι. For instance in the example code of Listing 1, page 4, the result of the instruction cmp r3, r0 line 2c is used at 30, 34, 38, 3c and 40, and the predicates needed are le and eq (gt is the negation of le). In the worst case we still need 4 variables to encode the outcome of an instruction ι that sets the status bits, but we reduce the choices of Player 2 to the predicates in *flags*(ι). In the previous examples, instead of having 16 branches from the current state, there will be only 4.

To model program p in UPPAAL, and simulate it, we need:

- an array, `val`, of 16 variables for the registers of the ARM9 processor;
- at most 4 Boolean variables for the encoding of the status bits (we use `cmple`, `cmplt`, `cmpls`, `cmpeq` instead of the actual status bits N, Z, V and C, but this is equivalent);
- a *stack*[10] of size K (the size of the stack is determined in a previous stage using a C program to simulate p).

Although the model-checker UPPAAL is very efficient, we have to be careful when encoding p: some information can be encoded using variables, but they will be part of the *state* of the network of TA we build, and will be encoded in the BDD representation of each state. Some information is not dynamic but rather static (e.g., the *type* of an instruction ι, or the registers read/written by an instruction). They are only used to make a decision when moving from one state to another (i.e., as *guard*) and thus can be encoded using UPPAAL *functions*. This saves space as functions are not part of the encoding of a state. This encoding is a major difference compared to the one used in METAMOC. Given a program p, we define the functions:

- *SetStatusB* : $p \to \mathbb{B}$ which, given an instruction $\iota \in p$, returns TRUE if ι sets some status bits (comparison instructions cmp, tst and instructions with the "s" flag like subs, adds etc) and FALSE otherwise;
- *cmpU* : $p \times \mathcal{V}_\perp \to \mathbb{B}$ returns TRUE if the result of ι in state v is unknown and FALSE otherwise.

As a shorthand we write $\mathit{NDcmp}(\iota, v) = \mathit{SetStatusB}(\iota) \wedge \mathit{cmpU}(\iota, v)$ and this indicates whether instruction ι, when executed from state v, should be played by Player 2 (some status bits have to be set but an operand is unknown).

In addition to this, we define another function *update* : $\mathcal{V}_\perp \to \mathcal{V}_\perp$ which updates the values of the registers and the status bits if required: this function encodes the semantics of each instruction on the extended domain.

All these functions are computed by our compiler (ARM2UPP, Fig. 6, Section V), and the result for the binary search program is given in Listings 2 and 3, Appendix B.

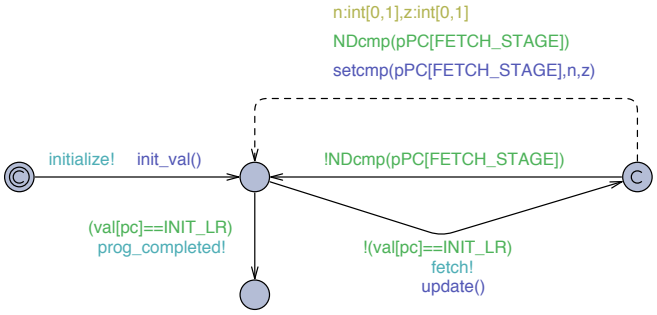

Figure 1. Generic Automaton *Prog* to Simulate a Program

The generic automaton to simulate a program p is given in Fig. 1. We assume that program p is called by a *caller* and a particular value `INIT_LR` gives the return point when p terminates. The automaton *Prog* (Fig. 1) performs some initialisation (`init_val()`) and then computes the next state until the end of the program is reached: this is when the value of the **pc** register is equal to the return point `INIT_LR` (guard `val[pc]=INIT_LR`). To simulate each instruction, the automaton *Prog* performs the following steps:

1) feed the current instruction ι to the first stage of the pipeline when it is empty (to do so it has to synchronise with the first stage of the pipeline on the `fetch!` channel) and compute the next state (`update()` function). This also sets the next value of register **pc**. The result of `update()` is that the number of the current instruction is stored into the variable `pPC[FETCH_STAGE]` which is part of the first stage (#0) of pipeline.
2) if the instruction ι in `pPC[FETCH_STAGE]` is an undetermined comparison, the function `NDcmp(pPC[FETCH_STAGE])` evaluates to TRUE and the upper dashed transition is taken: Player 2 chooses two values n and z and the predicates that must be set (`cmple`, `cmplt`, etc) are set by `setcmp` (Listing 2, page 12). If ι does not set any flag or the outcome is determined by the current state (the operands are all known), the middle plain transition is taken (Player 2 does not have to play).

[10]We simulate the program stack as well, as it is used extensively to pass parameters and store return addresses.

Abstractions. All instructions have side effects but some of them can be ignored as they have no influence on the WCET. Indeed the *effect* of an instruction does not always have to be taken into account: for instance, in the Fibonacci program, we are not interested in the actual result of the computation, but only in the time it takes to compute it. Some instructions can thus be *abstracted away* by only advancing the **pc** register without modifying the other registers.

In our framework, we can check (see Section V) whether an instruction can be safely abstracted. As witnessed by Table I, for some programs (`matmult` and `jfdcint`), a large part of the instructions can be abstracted away (the abstraction only abstracts the *effect* of the instructions on the registers but they are still fed into the pipeline with their timing constraints). This results in smaller ranges for registers and a more compact representation of the *state* of the system in UPPAAL, and consequently a faster analysis. Actually the two programs `matmult` and `jfdcint` generate a huge range of registers values without abstraction which may prevent their analysis or badly impact the computation time as witnessed on Table I by the results of METAMOC on these programs. Using abstraction, we were able to compute the WCET using a very small amount of time and memory.

IV. Model of the Hardware

In this section we give a UPPAAL model for the architecture of the ARM920T processor and the caches.

A. Model of the Pipeline

Each stage of the pipeline contains an instruction and some other information and this is stored in arrays: `pPC[k]` gives the index of the instruction in stage k; `Todo[k]` is a Boolean value and indicates whether the instruction `pPC[k]` is scheduled (some instructions are conditional and are skipped); `dataAdr[k]` contains the address[11] of the memory cell referenced by `pPC[k]` (-1 if none). There are 5 stages in the pipeline of the ARM9:

- stage 1: *fetch* stage. It fetches the next instruction (pointed to by **pc**) from the cache and this instruction becomes the current instruction at stage 1;
- stage 2: *decode* stage. It decodes the instruction.
- stage 3: *execute* stage. It carries out the computation (addition, comparisons, etc) of the instruction.
- stage 4: *memory* stage. It carries out the memory transfers (from registers to cache/main memory or cache/main memory to registers) of the instruction.
- stage 5: *writeback* stage. It writes the values of registers that are ("writeback") operands of the instruction.

[11]For multiple loads and stores, this should be a range of addresses; this information is used only for determining whether a stall should occur in the pipeline. For multiple loads and stores, we force a stall in a pipeline until the end of the multiple loads/stores instruction. This is a safe encoding as the ARM9 does not exhibit timing anomalies.

An instruction ι enters the pipeline at stage 1. It is transferred from stage i to $i+1$ as soon as possible. When it exits stage 5, it is completed. The execution of a program is completed when its last instruction is completed.

Pipeline Stalls. The goal of *pipelining* is to split the execution of an instruction into different simple steps. The idea being that each step can be carried out concurrently for different instructions: while stage 1 fetches the next instruction ι_k, stage 2 decodes instruction ι_{k-1}, etc. It may happen that the simple steps of some sequences of instructions cannot be carried out concurrently. A *pipeline stall* is a situation when one stage i of the pipeline cannot perform its computation because it has to wait for another stage $j > i$ to complete its computation.

To handle pipeline stalls we only transfer an instruction from the decode stage to the (next) execute stage if no write/read dependencies exist between the registers (and memory cells) at stage 2 and the registers at stage > 2. In the UPPAAL model (Fig. 2) of the decode stage, the transition `copy(me,me+1)` which feeds the next stage $me+1$ of the pipeline with the information from stage me, is *guarded* by the function `!stall()` ("no stall"). The registers read from/written to by an instruction are encoded using a function `reg` (see Appendix B, Listing 4.)

Branch Prediction. When a conditional branch instruction enters the pipeline, the next instruction to flow in is determined by the truth value of the condition. This value might not yet be available when the branch instruction is in the first stage of the pipeline. If the condition is determined by the value of a variable which is not in the cache, it might take a few cycles before the result becomes available. In this case, we should *stall* until the outcome of the comparison is computed. This might however be inefficient.

Some heuristics, *branch prediction*, can be applied to guess the most plausible next instruction after a conditional branching. After the *prediction*, the chosen instruction flows in the pipeline. If the guess was right the result is a shortest execution-time for this part of the program. If the guess was wrong, the computations of the mistakenly taken branch have to be undone, and the pipeline flushed which results in a longer execution-time. We do not discuss here the choice of a good heuristics, but there are some options that give good results on *average*. In our model we implement the ARM920T pipeline where there is no branch prediction.

UPPAAL Pipeline Model. The timed automata models we introduce are close to the ones proposed in [20]. However there are some differences as we do not have the same model for the program.

The timed automata for each stage (ARM9, 5 stages) are depicted on Fig. 2–5. The stage modelled by each automaton can be inferred by the synchronisation channel from the initial state (e.g., `decode?`). The first stage of the pipeline is of particular importance as it models the case of a wrong

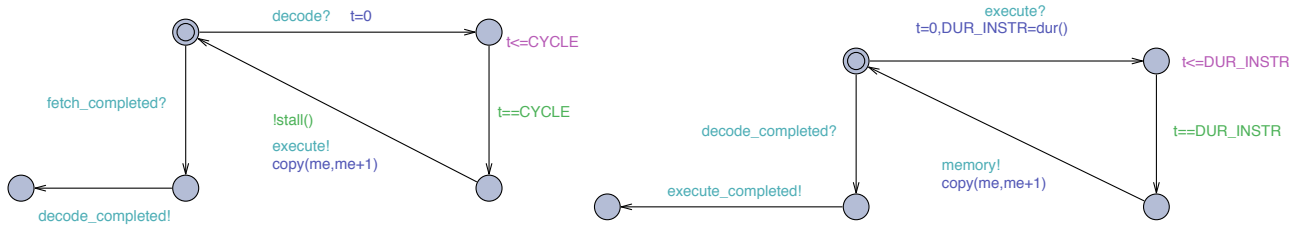

Figure 2. Timed Automata for the Decode and Execute Stages.

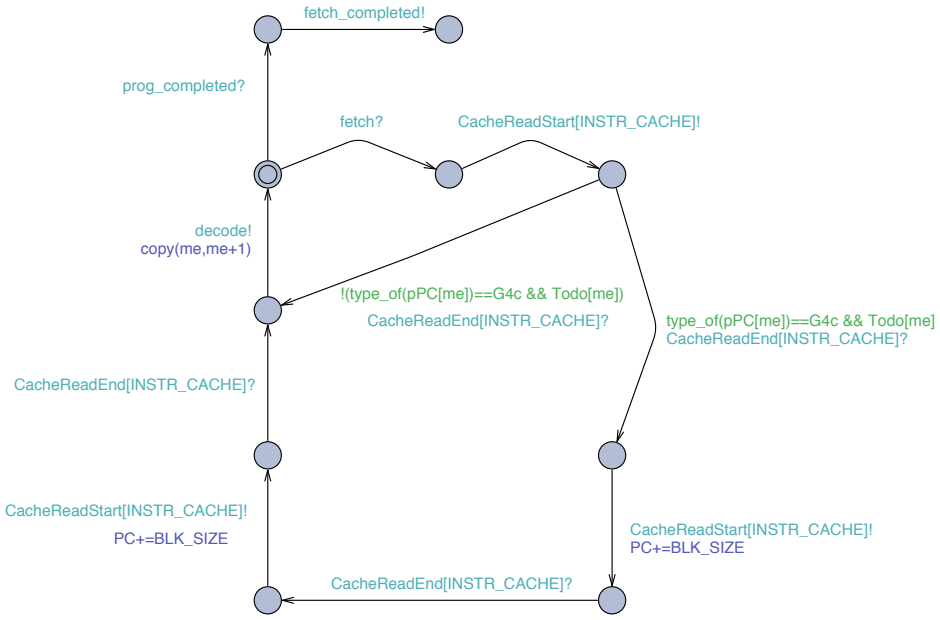

Figure 3. Timed Automata Model of the ARM9 Pipeline

guess in an branch prediction. The automaton of Fig. 3 models the fetch stage:

1) it accepts a `fetch?` synchronisation when it is idle;
2) after accepting an instruction (`fetch?` synchronises with `fetch!` in the automaton *Prog* of Fig. 1), it actually fetches the instruction from main memory via the *instruction cache* (on channel `CacheReadStart[INSTR_CACHE]!`, where `INSTR_CACHE` is the ID or the instruction cache);
3) when the instruction has been read from the cache there are two options:
   a) the instruction to be processed is a *conditional branch* (e.g., `type_of(pPC[me])==G4c`) and the variable `Todo[me]` indicates whether the condition was evaluated to TRUE or FALSE. In case it is a conditional branch and the condition was TRUE, we simulate two "instruction read from the cache" steps: indeed our branch prediction algorithm is "never branch" and thus if it happened that we had to branch, we should simulate a pipeline `flush`. As we do not execute the instructions in the pipeline (but rather when we feed the first stage of the pipeline), this can be modelled by reading the next two instructions (the "never branch" prediction) without executing them, and then resuming the simulation from the target address of the branch instruction.
   b) the instruction to be processed is not a conditional branching or the condition was evaluated to FALSE; in this case the prediction was right and nothing has to be undone.

   After an instruction has been fetched in the `fetch` stage, it is fed to the next stage. This is modelled by the `decode!` synchronisation and the `copy(me,me+1)` transition. `copy(me,me+1)` copies the information in the variables `pPC[me]`, `Todo[me]` and `dataAdr[me]` to the next stage `me+1`.

B. Model of the Caches

We use two caches (instruction and data) with FIFO replacement policy and assume write allocate on a write/miss. The automaton modeling the behaviour of a cache (together with the model of the main memory automaton) is given in Fig. 7, appendix B. The caches are j-way caches where the L-line cache is partitioned into L/j sets. In the example of

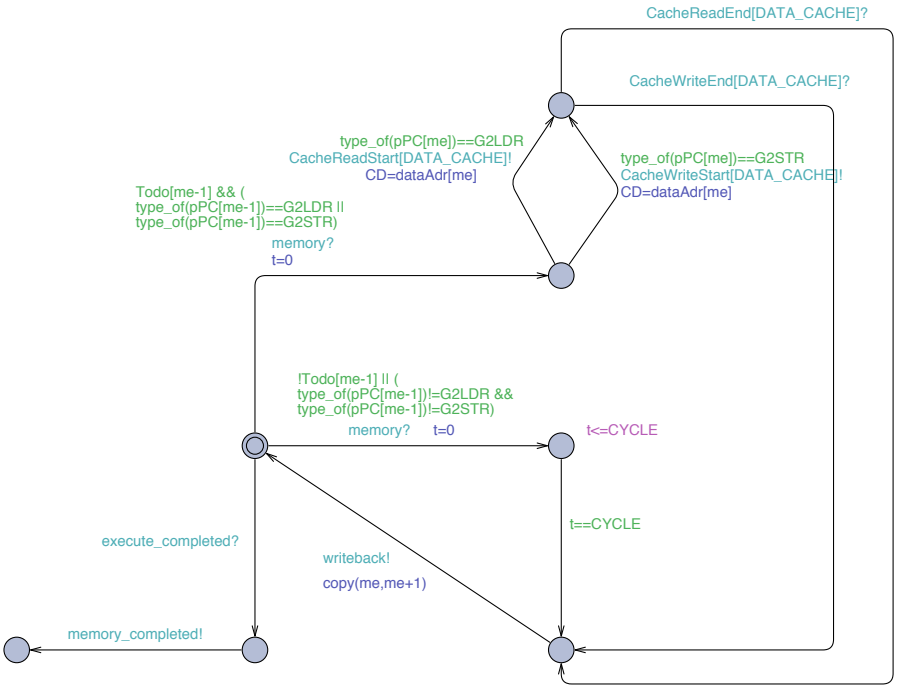

Figure 5. Timed Automata for the Memory Stage

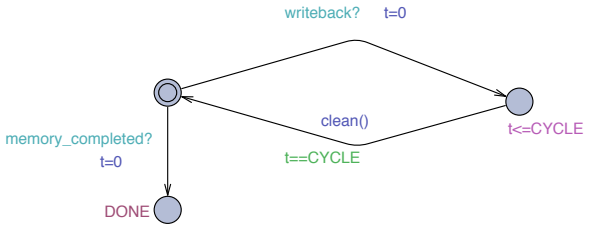

Figure 4. Timed Automata for the Writeback Stage

Table I, we use caches with 16 lines and 4 sets. The length of a line is set to be 4 words. The caches are initially empty but the initial set can be set to any value (could be fully filled). Notice that the two caches share the same main memory and thus synchronization with this memory is necessarily serialized (The main memory automaton of Fig. 7 can only accept one request at a time).

C. Timing Anomalies

Timing anomalies [1] can occur because of the complex architecture of the hardware H. The term refers to counter-intuitive observations in the sense that larger *local* execution-times may not result in larger *global* execution-times on some given hardware. *Pre-fetching* instructions can lead to such observations on some processors. This can also be observed on complex pipeline architectures (e.g., *out-of-order* execution of instructions).

The ARM9 processor does not exhibit timing anomalies. [21] reports that using a FIFO cache on the ARM9 can cause a *domino effect*. This is not related to actual hardware timing anomalies but to timing anomalies related to abstractions of programs, specifically when unrolling loops. We do not use loop unrolling but perform a fair simulation of a binary program and thus the domino effect does not affect our method which is another difference with METAMOC.

V. Tool Chain and Case Studies

The model we check to compute the WCET consists of the automata $Prog$ 1, the automata the different stages of the pipeline 2–5 and the automata modelong the instruction and data caches (the reader is refered to http://www.irccyn.fr/franck/wcet for details of the models).

Tool Chain. The tool chain to compute WCET is depicted on Fig. 6. The component we have developed are `ARM2UPP` to generate a game model g of a program and `PATCH_UPP` to replace some constants in g (e.g., maximum stack size, range of registers) once they are computed:

- `ARM2UPP` takes as input a program in assembly code (`file.arm`) and generates four files:
  - `file.{xml,q}` that contain respectively the UPPAAL network automata (and functions like `update()`) modelling the execution of the program on the architecture of the ARM9 and the UPPAAL queries to compute the WCET;
  - `file-reach.cpp` is a C++ program that simulates the program in `file.arm`. It does so

using the extended domain for variables using the unknown $\perp$. When a comparison is unknown both branches are explored. This ensures that our simulation explores at all the actual possible paths of the program (and maybe some infeasible ones). `file-reach.cpp` is compiled into `file-reach.exe`. When this program terminates the file `file.inf` contains some useful information (like maximal stack size, registers ranges, etc) that can be used further on.

  - `file-equiv.cpp` is a C++ program that checks whether an abstraction mapping (which is given by a function) is valid or not, i.e., the abstracted program is equivalent WCET-wise to the non-abstracted one. The abstracted instructions are given (by the user as a list). This program is compiled into an executable `file-equiv.exe` that returns either TRUE if equivalence holds and FALSE otherwise; equivalence by that checking the synchnonized product of the non-abstracted program and the abstracted one generate exactly the same sequence of instructions into the pipeline and the same memory references.

- `PATCH_UPP` modifies some constants in the network of TA of `file.xml` to incorporate the information from `file.inf` (like stack size, registers range) and can also include the function giving the abstracted instructions.

Notice that we assume the input is a C program but the entry point can be the compiled binary ELF program.

UPPAAL-TiGA Queries. In order to compute the WCET of a program, we can check whether the program always terminates within k time units. This can be computed using a binary search with UPPAAL. The drawback of this check is that some deadlocks may occur in the system, yielding a biased value of the WCET.

An alternative way of computing the WCET is to check a *control* property: "Can Player 1 enforce termination of the program and if yes, what is the best duration he can guarantee?" This optimal time reachability control objective can be checked in one query (see [24]) with UPPAAL-TiGA [23], provided we know an upper bound of the WCET and this can be roughly over-estimated on the program.

Program termination in the UPPAAL model happens when the location `DONE` is reached in the `writeBackStage` automaton (last stage of the pipeline). Technically, the optimal time is computed using the `control_t*` modality of UPPAAL-TiGA as follows:

```
control_t*(#n,0):A[true U writeBackStage.DONE]
```

where $\#n$ is an upper bound of the expected WCET.

Case Studies & Results. We have applied the framework described in Fig. 6 to a number of benchmark programs from Mälardalen University. We could not analyse the full set of programs because of the current limitations of our tools: floating point operations are not supported yet; a few operators (e.g., `ror`) of the ARM9 assembly language are not supported yet. To evaluate the relevance of our method, we compare our results concerning the time/space needed to compute WCET to the ones obtained with the METAMOC method [21]. We have chosen to implement FIFO caches because we want to compare our computed results to the actual execution times of the programs running on our testbed which is an ARM920T. This means that the actual WCET computed with METAMOC and ours are not comparable as we use FIFO caches vs LRU caches in METAMOC.

There are 21 programs that can be analysed by METAMOC using a concrete instruction and data caches. Using our encoding and tool chain, we could analyse 14 programs (the remaining contain unsupported operations) with concrete instruction and data caches. Moreover, the time/space needed to compute the results is very small compared to the resources used in METAMOC. Table I, page 3, gives the values of WCET for each program, and the time for UPPAAL-TiGA[12] to compute the result. The time needed to compute the intermediary files is negligible. The UPPAAL files are available from http://www.irccyn.fr/franck/wcet.

It is to be noticed that *abstraction* seems to be an important point to be able to handle programs with a large number of states/paths: indeed for `matmult` and `jfdcint` we can compute the WCET in less than 2mins whereas METAMOC runs out of memory or needs around 16mins (on a faster machine than ours).

Energy/Power Consumption Optimisation. Dealing with power consumption is also a strength of the timed automata based approach (METAMOC shares this feature). It is very easy to model *dynamic* changes in the speed of the processor: it suffices to set the value of the processor speed in the models: the processor speed can be switched during the execution of the program using a timed automaton to set the current speed. As an example, we have computed the WCET when only one switch is allowed during the program execution. The processor starts at $1/4$th of its fastest speed and at some fixed point in time switches to full speed. The last column of Table I gives the percentage of time the processor can run at the slower clock rate without any impact on the WCET: this is due to the initial transient phase of the execution of a program where instructions are loaded into the cache. For some small programs the result is impressive (22% for `janne-complex`).

VI. Conclusion & Future Work

In this paper we have presented a framework based on timed games and the model checker UPPAAL-TiGA

[12] We used version 0.12 for MacOS (http://www.cs.aau.dk/~adavid/tiga/).

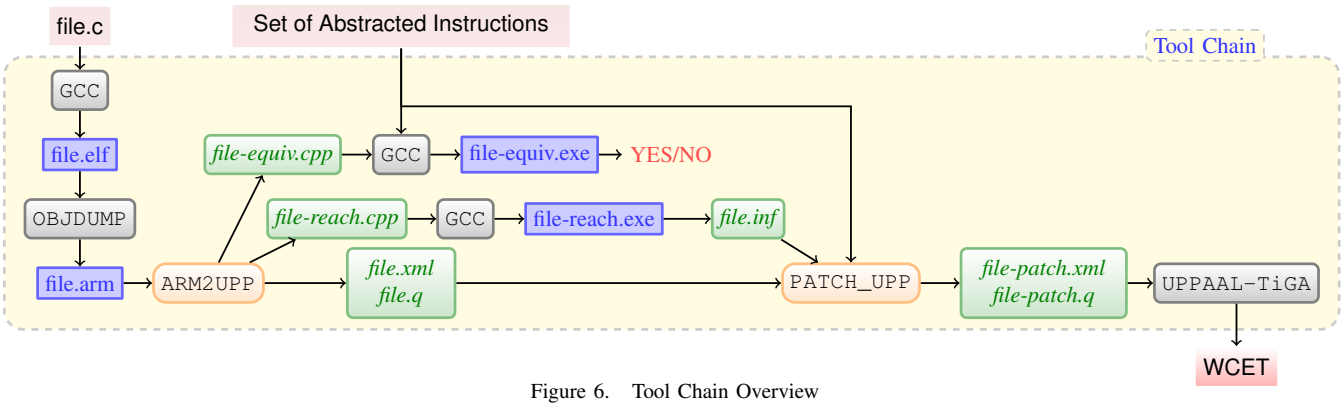

Figure 6. Tool Chain Overview

to compute WCET for programs running on architectures featuring pipelining and caching.

The advantages of our approach are many-fold (METAMOC [21] shares 1–3,6):

1) it is very easy to implement as it consists of two separate and independent phases: 1) computation of a model of the program to be analysed; this only requires a (formal) semantics of the assembly language of the target processor[13]; 2) computation of the WCET with UPPAAL-TiGA and the models for the caches, pipelines which specify the timing features.
2) the design of the models for pipeline stages and caches can be stressed by simulating some simple sample programs; this enables us to get more confidence in the model of the hardware as this is not hidden in the analysis algorithm; this is especially important for concurrent architectures like pipelined processors that can be hard to describe;
3) UPPAAL or UPPAAL-TiGA can be used to simulate the program on the architecture. It is thus a quick way of obtaining a simulator for a given hardware;
4) (manual) annotations are not required. On the 14 programs of Table I, the analysis is fully automatic; notice though that if termination is ensured by an input data dependent check in the program we cannot discover it.
5) we solve an *optimal time reachability problem* on the program p of the form: "what is the optimal time to enforce *termination* of program p ?". This at once 1) proves that p running on our hardware model terminates on every input data, and 2) computes the WCET.
6) using timed automata, it is easy to add *power* related constraints in the model e.g., processor speed switches;
7) we can check that an abstract program (with the effect of some instructions ignored) is *equivalent* to a concrete one and reduce the size of the encoding of a *state* of a computation.

The results we have obtained support the claim that model checking is adequate for computing WCET. Also to be noticed is that UPPAAL-TiGA could be tuned to handle WCET computation even more efficiently: *priorities* between processes can reduce unnecessary interleavings and there are not yet implemented in UPPAAL-TiGA (though they are in UPPAAL); a lot of time is spent *checking* whether a new state has already been encountered: this will never be the case in the programs we check (otherwise there would be an infinite loop). Disabling this check would also reduce the time to compute the results. We advocate the combination of different techniques like model-checking and absraction to solve the WCET problem. Indeed *Abstract Interpretation* (AI) combined with *Integer Linear Programming* (ILP) have given very good results [11] but this method is yet to prove that: (1) it can be *easily* adapted to different processors or multi-core processors and (2) it can take into account *power* related features (like change of speed of the processor).

Our ongoing work focuses on the following aspects:

1) compare the WCET computed with our method to WCET obtained on the actual hardware using a testbed whihc is an ARM920T;
2) extend the set of supported instructions and provide models for other architectures (like ARM11);
3) refine the models of the architecture to reflect more accurately the actual hardware;
4) compute an abstract program automatically (currently given as an input file). This has been recently implemented using the technique of *slicing* [27] and thus an abstract program is computed automatically;
5) prune the execution tree of the program. The goal of this step is to reduce the number of paths of the program still preserving the paths giving the WCET. Indeed using an over-approximation (having more paths) of a program may result in over-estimated WCET or oversized models that cannot be model checked. Another solution is to use an under-approximation (less paths to explored) that preserves the paths yielding to the WCET. Techniques can be developed to try and prune the tree (an example of such techniques was used to compute optimal *job shop scheduling* [28]).

[13] In contrast, the verification-based tools would need a description of the hardware to compute the CFG.

REFERENCES

[1] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström, "The Worst-Case Execution-Time Problem - Overview of Methods and Survey of Tools," *ACM Trans. Embedded Comput. Syst.*, vol. 7, no. 3, 2008.

[2] Rapita Systems Ltd., "Rapita Systems for timing analysis of real-time embedded systems." http://www.rapitasystems.com/.

[3] G. Bernat, A. Colin, and S. M. Petters, "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.

[4] B. Rieder, P. Puschner, and I. Wenzel, "Using Model Checking to Derive Loop Bounds of General Loops within ANSI-C Applications for Measurement Based WCET Ananlysis," in *Proc. of the 6th Int. Workshop on Intelligent Solutions in Embedded Systems (WISES'08)*, Regensburg, Germany, 2008.

[5] Tidorum Ltd., "Bound-T time and stack analyser." http://www.tidorum.fi/bound-t/.

[6] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, "Otawa: An open toolbox for adaptive wcet analysis," in *Software Technologies for Embedded and Ubiquitous Systems (SEUS) - 8th IFIP WG 10.2 International Workshop, SEUS 2010, Waidhofen/Ybbs, Austria, October 13-15, 2010. Proceedings*, ser. Lecture Notes in Computer Science, S. L. Min, R. G. P. IV, P. P. Puschner, and T. Ungerer, Eds., vol. 6399. Springer, 2010, pp. 35–46.

[7] A. Prantl, M. Schordan, and J. Knoop, "TuBound - A Conceptually New Tool for WCET Analysis." in *Proceedings of the 8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis (WCET'08)*, Prague, Czech Republic, Jul. 2008.

[8] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury, "Chronos: A Timing Analyzer for Embedded Software," *Science of Compuer Programming*, vol. 69, no. 1–3, 2007, special Issue on Experimental Software and Toolkit.

[9] J. Engblom, A. Ermedahl, M. Nolin, J. Gustafsson, and H. Hansson, "Worst-case execution-time analysis for embedded real-time systems," *International Journal on Software Tools for Technology Transfer*, vol. 4, no. 4, pp. 437–455, October 2003.

[10] AbsInt Angewandte Informatik, "aiT Worst-Case Execution Time Analyzers." http://www.absint.com/ait/.

[11] C. Ferdinand, R. Heckmann, and R. Wilhelm, "Analyzing the worst-case execution time by abstract interpretation of executable code," in *ASWSD*, ser. Lecture Notes in Computer Science, M. Broy, I. H. Krüger, and M. Meisinger, Eds., vol. 4147. Springer, 2004, pp. 1–14.

[12] N. Holsti, J. Gustafsson, G. Bernat, C. Ballabriga, A. Bonenfant, R. Bourgade, H. Cassé, D. Cordes, A. Kadlec, R. Kirner, J. Knoop, P. Lokuciejewski, N. Merriam, M. D. Michiel, A. Prantl, B. Rieder, C. Rochange, P. Sainrat, and M. Schordan, "Wcet 2008 - report from the tool challenge 2008," in *Proceedings of the 8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis (WCET'08)*, Prague, Czech Republic, Jul. 2008.

[13] R. Alur and D. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.

[14] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a Nutshell," *Journal of Software Tools for Technology Transfer (STTT)*, vol. 1, no. 1-2, pp. 134–152, 1997.

[15] G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks, "Uppaal 4.0," in *QEST*. IEEE Computer Society, 2006, pp. 125–126.

[16] R. Wilhelm, "Why ai + ilp is good for wcet, but mc is not, nor ilp alone," in *VMCAI*, ser. Lecture Notes in Computer Science, B. Steffen and G. Levi, Eds., vol. 2937. Springer, 2004, pp. 309–322.

[17] A. Metzner, "Why model checking can improve wcet analysis," in *CAV*, ser. Lecture Notes in Computer Science, R. Alur and D. Peled, Eds., vol. 3114. Springer, 2004, pp. 334–347.

[18] B. Huber and M. Schoeberl, "Comparison of Implicit Path Enumeration and Model Checking Based WCET Analysis," in *Proceedings of the 9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis (WCET'09)*, Dublin, Ireland, Jul. 2009.

[19] M. Ouimet and K. Lundqvist, "The TASM Toolset: Specification, Simulation and Formal Verification of Real-Time Systems," in *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, ser. Lecture Notes in Computer Science, vol. 4590. Springer, 2007, pp. 126–130, (Tool Paper).

[20] A. E. Dalsgaard, M. C. Olesen, and M. Toft, "Modular execution time analysis using model checking," Master's thesis, Department of Computer Science, Aalborg University, Denmark, 2009.

[21] A. E. Dalsgaard, M. C. Olesen, M. Toft, R. R. Hansen, and K. G. Larsen, "Metamoc: Modular execution time analysis using model checking," in *WCET*, ser. OASICS, B. Lisper, Ed., vol. 15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2010, pp. 113–123.

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

[23] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. G. Larsen, and D. Lime, "Uppaal-tiga: Time for playing games!" in *CAV*, ser. Lecture Notes in Computer Science, W. Damm and H. Hermanns, Eds., vol. 4590. Springer, 2007, pp. 121–125, (Tool Paper).

[24] F. Cassez, A. David, E. Fleury, K. G. Larsen, and D. Lime, "Efficient on-the-fly algorithms for the analysis of timed games," in *Proc. of the 16th Int. Conf. on Concurrency Theory (CONCUR'05)*, ser. Lecture Notes in Computer Science, vol. 3653. Springer, Aug. 2005, pp. 66–80.

[25] F. Cassez, "Timed Games for Computing Worst-Case Execution-Times," National ICT Australia, Research Report, Jun. 2010, 31 pages. Available from http://arxiv.org/abs/1006.1951.

[26] ARM Ltd., "ARM9 – ARM9 Processor Family," available from http://www.arm.com/products/CPUs/families/ARM9Family.html.

[27] M. Weiser, "Program slicing," *IEEE Trans. Software Eng.*, vol. 10, no. 4, pp. 352–357, 1984.

[28] Y. Abdeddaïm, E. Asarin, and O. Maler, "Scheduling with timed automata," *Theor. Comput. Sci.*, vol. 354, no. 2, pp. 272–300, 2006.

APPENDIX

A. Caches

A *cache* is a fast memory device. It is characterised by its size K (usually in Kbytes), the length of a cache *line* (B in Bytes) and the number of cache lines $L = \frac{K}{B}$.

Fig. 7 depicts an automaton to implement a cache read-/write. The variable `PMT` give the number of memory transfers needed for the data to be in the cache: as we use *dirty* bits, it may be the case that two transfers are needed. Fig. 8 gives the TA for the main memory component.

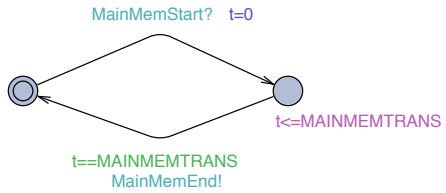

Figure 8. Timed Automata Models for the Caches and main memory

B. Functions Used to Simulate a Program

The following code is generated by `ARM2UPP` in the tool chain (Fig. 6).

C Code for `SetStatusB`, `cmpU` and `NDcmp`

```
1:  /* function to determine whether status bits should ne set */
2:  bool SetStatusB(int i) { // i is the PC of instruction;
        function that tells whether status bits should be set
3:  // comparisons for function fib
4:  // comparisons for function main
5:  /* cmp/cmn for function main starting 0 ending 4 */
6:
7:  // end def of cmp/cmn for function main
8:  // comparisons for function binary_search
9:  /* cmp/cmn for function binary_search starting 8 ending 80 */
10: if (i==44) { // setting status bits for instruction cmp at 44
        [0x2c]
11:     return true ;
12: }
13: if (i==68) { // setting status bits for instruction cmp at 68
        [0x44]
14:     return true ;
15: }
16:
17: // end def of cmp/cmn for function binary_search
18: return false ;
19: }
20:
21: /*  setcmp for instructions used in the program */
22: void setcmp(int  i,bool n1,bool n2) {
23: /* res_comp for function binary_search starting 8 ending 80 */
24:     if (i==44) { // instruction  cmp r3, r0 at 44 [0x2c]
25:         cmpeq=n1;
26:         cmple=n2;
27:     }
28:     if (i==68) { // instruction  cmp lr, ip at 68 [0x44]
29:         cmple=n1;
30:     }
31:
32: } // end setcmp of instruction
33:
34: bool NDcmp(int i) {
35:           return SetStatusB(i) && cmpU(i) ;
36: }
```

Listing 2. C Code for `SetStatusB`, `cmpU` and `NDcmp`

C Code for `update` function

```
1:  void update() { // update function
2:  int nextpc,nextfp,tmp;
3:  /*
4:   updates for function main starting 0 ending 4
```

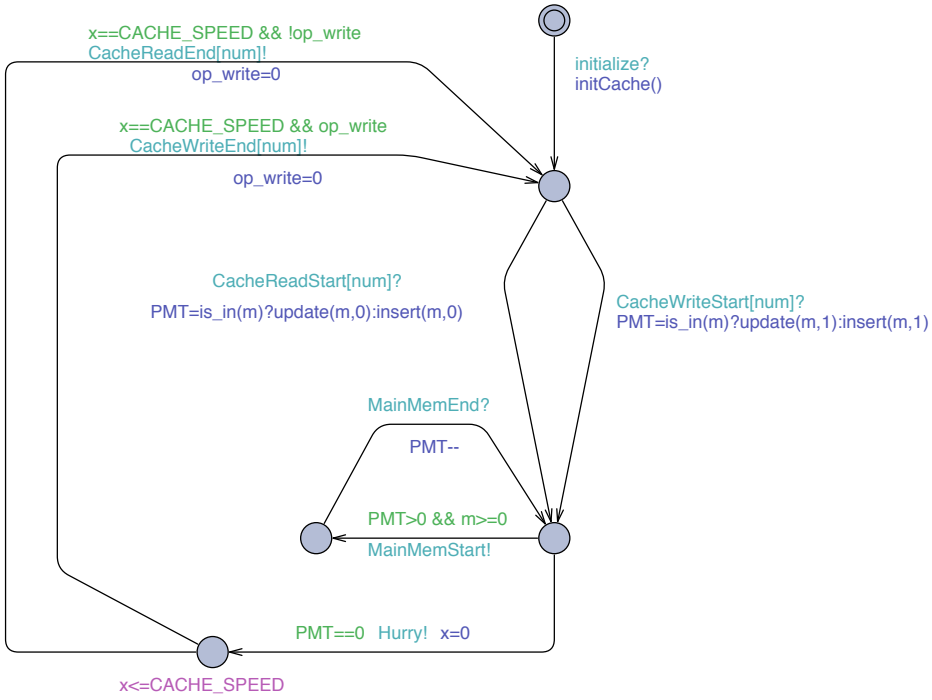

Figure 7. Timed Automata Models for the Caches and main memory

```
*/
if (val[pc]==0) { // Instruction mov r0, #9 at 0x0
    nextpc=val[pc]+4;
    if (!is_abstracted(val[pc])) {
        val[r0]=(9);
        SET(0,-1,1); // instruction scheduled is 0, no memory access and scheduled
    }
    else SET(0,-1,0) ; // instruction not scheduled, no mem access
} // end mov at 0x0
if (val[pc]==4) { // Instruction b 8, (unconditional) at 0x4
    nextpc=8; // to 0x8
    SET(4,-1,1) ; // instruction scheduled, no mem access, branching
} // end b at 0x4

/*
 end of updates for function main
*/

/*
 updates for function binary_search starting 8 ending 80
*/
if (val[pc]==8) { // Instruction stmdb sp!,{r4,r5,lr,} at 0x8
    nextpc=val[pc]+4;
    // push should first decrease val[pc] and then store in stack(val[pc])
    push(val[lr]);
    push(val[r5]);
    push(val[r4]);
    SET(8,-1,1); // instruction scheduled is 8, no memory access
} // end stmdb at 0x8
if (val[pc]==12) { // Instruction ldr r4,[pc,#64] at 0xc
    nextpc=val[pc]+4;
    val[r4]=content(84);
    SET(12,content(84),1); // instruction scheduled is 12, memory access to content(84)
} // end ldr at 0xc
if (val[pc]==16) { // Instruction mov lr, #0 at 0x10
    nextpc=val[pc]+4;
    if (!is_abstracted(val[pc])) {
        val[lr]=(0);
        SET(16,-1,1); // instruction scheduled is 16, no memory access and scheduled
    }
    else SET(16,-1,0) ; // instruction not scheduled, no mem access
} // end mov at 0x10
if (val[pc]==20) { // Instruction mov ip, #14 at 0x14
    nextpc=val[pc]+4;
    if (!is_abstracted(val[pc])) {
        val[ip]=(14);
        SET(20,-1,1); // instruction scheduled is 20, no memory access and scheduled
    }
    else SET(20,-1,0) ; // instruction not scheduled, no mem access
} // end mov at 0x14
if (val[pc]==24) { // Instruction mvn r5, #0 at 0x18
    nextpc=val[pc]+4;
    if (!is_abstracted(val[pc])) {
        val[r5]=-(0+1);
        SET(24,-1,1); // instruction scheduled is 24, no memory access and scheduled
    }
    else SET(24,-1,0) ; // instruction not scheduled, no mem access
} // end mvn at 0x18
if (val[pc]==28) { // Instruction add r3, lr, ip at 0x1c
    nextpc=val[pc]+4;
    if (!is_abstracted(val[pc])) {
        if (val[lr]==UNKNOWN||val[ip]==UNKNOWN) {
            val[r3]=UNKNOWN;
        }
        else {
            val[r3]=(val[lr]+val[ip]);
        }
        SET(28,-1,1); // instruction scheduled is 28, no memory access and scheduled
    }
    else SET(28,-1,0) ; // instruction not scheduled, no mem access
} // end add at 0x1c
if (val[pc]==32) { // Instruction mov r2, r3 asr #1 at 0x20
    nextpc=val[pc]+4;
    if (!is_abstracted(val[pc])) {
        if (val[r3]==UNKNOWN) {
            val[r2]=UNKNOWN;
        }
        else {
            val[r2]=((val[r3] >>  1));
        }
        SET(32,-1,1); // instruction scheduled is 32, no memory access and scheduled
```

```
    }
    else SET(32,-1,0) ; // instruction not scheduled, no mem access
} // end mov at 0x20
if (val[pc]==36) { // Instruction ldr r3,[r4,r2 lsl #3] at 0x24
    nextpc=val[pc]+4;
    val[r3]=content(val[r4]+(val[r2] <<  3));
    SET(36,val[r4]+(val[r2] <<  3),1); // instruction scheduled is 36, memory access to val[r4]+(val[r2] <<  3)
} // end ldr at 0x24
if (val[pc]==40) { // Instruction add r1, r4, r2 lsl #3 at 0x28
    nextpc=val[pc]+4;
    if (!is_abstracted(val[pc])) {
        if (val[r4]==UNKNOWN||val[r2]==UNKNOWN) {
            val[r1]=UNKNOWN;
        }
        else {
            val[r1]=(val[r4]+(val[r2] <<  3));
        }
        SET(40,-1,1); // instruction scheduled is 40, no memory access and scheduled
    }
    else SET(40,-1,0) ; // instruction not scheduled, no mem access
} // end add at 0x28
if (val[pc]==44) { // Instruction cmp r3, r0 at 0x2c
    nextpc=val[pc]+4;
    if (!is_abstracted(val[pc])) {
        // Should set the Z and N and C bits
        if ((val[r3]-val[r0])==0) cmpeq=1 ; else cmpeq=0;
        if ((val[r3]-val[r0])<=0) cmple=1 ; else cmple=0;
        SET(44,-1,1); // instruction scheduled is 44, no memory access and scheduled
    }
    else SET(44,-1,0) ; // instruction not scheduled, no mem access
} // end cmp at 0x2c
if (val[pc]==48) { // Instruction ldreq r5,[r1,#4] at 0x30
    nextpc=val[pc]+4;
    if (eq()) {
        val[r5]=content(val[r1]+4);
        SET(48,val[r1]+4,1); // instruction scheduled is 48, memory access to val[r1]+4
    }
    else SET(48,-1,0) ; // instruction not scheduled, no mem access
} // end ldreq at 0x30
if (val[pc]==52) { // Instruction subeq ip, lr, #1 at 0x34
    nextpc=val[pc]+4;
    if (!is_abstracted(val[pc]) && eq()) {
        if (val[lr]==UNKNOWN) {
            val[ip]=UNKNOWN;
        }
        else {
            val[ip]=(val[lr]-1);
        }
        SET(52,-1,1); // instruction scheduled is 52, no memory access and scheduled
    }
    else SET(52,-1,0) ; // instruction not scheduled, no mem access
} // end subeq at 0x34
if (val[pc]==56 && (!eq())) { // Instruction beq 44,  at 0x38
    nextpc=val[pc]+4;
    SET(56,-1,0) ; // instruction scheduled, no mem access, no branching
} // end beq at 0x38 [cond false]
if (val[pc]==56 && eq()) { // Instruction beq 44,  at 0x38
    nextpc=68; // to 0x44
    SET(56,-1,1) ; // instruction scheduled, no mem access, branching
} // end beq at 0x38 [cond true]
if (val[pc]==60) { // Instruction subgt ip, r2, #1 at 0x3c
    nextpc=val[pc]+4;
    if (!is_abstracted(val[pc]) && gt()) {
        if (val[r2]==UNKNOWN) {
            val[ip]=UNKNOWN;
        }
        else {
            val[ip]=(val[r2]-1);
        }
        SET(60,-1,1); // instruction scheduled is 60, no memory access and scheduled
    }
    else SET(60,-1,0) ; // instruction not scheduled, no mem access
} // end subgt at 0x3c
if (val[pc]==64) { // Instruction addle lr, r2, #1 at 0x40
    nextpc=val[pc]+4;
    if (!is_abstracted(val[pc]) && le()) {
        if (val[r2]==UNKNOWN) {
            val[lr]=UNKNOWN;
        }
        else {
            val[lr]=(val[r2]+1);
        }
        SET(64,-1,1); // instruction scheduled is 64, no memory access and scheduled
    }
    else SET(64,-1,0) ; // instruction not scheduled, no mem access
} // end addle at 0x40
if (val[pc]==68) { // Instruction cmp lr, ip at 0x44
    nextpc=val[pc]+4;
    if (!is_abstracted(val[pc])) {
        // Should set the Z and N and C bits
        if ((val[lr]-val[ip])<=0) cmple=1 ; else cmple=0;
        SET(68,-1,1); // instruction scheduled is 68, no memory access and scheduled
    }
    else SET(68,-1,0) ; // instruction not scheduled, no mem access
} // end cmp at 0x44
if (val[pc]==72) { // Instruction movgt r0, r5 at 0x48
    nextpc=val[pc]+4;
    if (!is_abstracted(val[pc]) && gt()) {
        if (val[r5]==UNKNOWN) {
            val[r0]=UNKNOWN;
        }
        else {
            val[r0]=(val[r5]);
        }
        SET(72,-1,1); // instruction scheduled is 72, no memory access and scheduled
    }
    else SET(72,-1,0) ; // instruction not scheduled, no mem access
} // end movgt at 0x48
if (val[pc]==76 && (!le())) { // Instruction ble 1c,  at 0x4c
    nextpc=val[pc]+4;
    SET(76,-1,0) ; // instruction scheduled, no mem access, no branching
} // end ble at 0x4c [cond false]
if (val[pc]==76 && le()) { // Instruction ble 1c,  at 0x4c
    nextpc=28; // to 0x1c
    SET(76,-1,1) ; // instruction scheduled, no mem access, branching
} // end ble at 0x4c [cond true]
if (val[pc]==80) { // Instruction ldmia sp!,{r4,r5,pc,} at 0x50
    nextpc=val[pc]+4;
    val[r4]=pop();
    val[r5]=pop();
    nextpc=pop();
    SET(80,-1,1); // instruction scheduled is 80, no memory access
} // end ldmia at 0x50


/*
 end of updates for function binary_search
*/

val[pc]=nextpc;
} // end update
```

Listing 3. C Code for `update` function

C Code for `reg`; Use `reg(in,1)` (resp. `reg(in,0)`) for written to (resp. read from) registers

```
/*  registers  used in the program */
int[0,65536] reg(int instruction, bool w) {
int[0,65536] res=0;
    /* Registers for function main starting 0 ending 4 */
    if (instruction==0) { // registers used in instruction mov r0, #9 at 0x0
        if (w) res=(1 << r0);
        else  res=(0);
    } // end mov at 0x0
    if (instruction==4) { // registers used in instruction b 8, at 0x4
        if (w) res=0;
        else  res=0;
    } // end b at 0x4

    /* Registers for function binary_search starting 8 ending 80 */
    if (instruction==8) { // registers used in instruction stmdb sp!,{r4,r5,lr,} at 0x8
        if (w) res=(1 << sp);
        else  res=( 0  | (1 << r4) | (1 << r5) | (1 << lr))
            ;
```

```
    } // end stmdb at 0x8
    if (instruction==12) { // registers used in instruction ldr
          r4,[pc,#64] at 0xc
            if (w) res=r4;
            else  res=((1 << pc));
    } // end ldr at 0xc
    if (instruction==16) { // registers used in instruction mov
          lr, #0 at 0x10
            if (w) res=(1 << lr);
            else  res=(0);
    } // end mov at 0x10
    if (instruction==20) { // registers used in instruction mov
          ip, #14 at 0x14
            if (w) res=(1 << ip);
            else  res=(0);
    } // end mov at 0x14
    if (instruction==24) { // registers used in instruction mvn
          r5, #0 at 0x18
            if (w) res=(1 << r5);
            else  res=(0);
    } // end mvn at 0x18
    if (instruction==28) { // registers used in instruction add
          r3, lr, ip at 0x1c
            if (w) res=(1 << r3);
            else  res=((1 << lr) | (1 << ip));
    } // end add at 0x1c
    if (instruction==32) { // registers used in instruction mov
          r2, r3 asr #1 at 0x20
            if (w) res=(1 << r2);
            else  res=(0 | (1 << r3));
    } // end mov at 0x20
    if (instruction==36) { // registers used in instruction ldr
          r3,[r4,r2 lsl #3] at 0x24
            if (w) res=r3;
            else  res=((1 << r4) | (1 <<r2));
    } // end ldr at 0x24
    if (instruction==40) { // registers used in instruction add
          r1, r4, r2 lsl #3 at 0x28
            if (w) res=(1 << r1);
            else  res=((1 << r4) | (1 << r2));
    } // end add at 0x28
    if (instruction==44) { // registers used in instruction cmp
          r3, r0 at 0x2c
            if (w) res=0;
            else  res=((1 << r3) | (1 << r0));
    } // end cmp at 0x2c
    if (instruction==48) { // registers used in instruction
         ldreq r5,[r1,#4] at 0x30
            if (w) res=r5;
            else  res=((1 << r1));
    } // end ldreq at 0x30
    if (instruction==52) { // registers used in instruction
        subeq ip, lr, #1 at 0x34
            if (w) res=(1 << ip);
            else  res=((1 << lr));
    } // end subeq at 0x34
    if (instruction==56) { // registers used in instruction beq
          44,  at 0x38
            if (w) res=0;
            else  res=0;
    } // end beq at 0x38
    if (instruction==60) { // registers used in instruction
         subgt ip, r2, #1 at 0x3c
            if (w) res=(1 << ip);
            else  res=((1 << r2));
    } // end subgt at 0x3c
    if (instruction==64) { // registers used in instruction
        addle lr, r2, #1 at 0x40
            if (w) res=(1 << lr);
            else  res=((1 << r2));
    } // end addle at 0x40
    if (instruction==68) { // registers used in instruction cmp
          lr, ip at 0x44
            if (w) res=0;
            else  res=((1 << lr) | (1 << ip));
    } // end cmp at 0x44
    if (instruction==72) { // registers used in instruction
         movgt r0, r5 at 0x48
            if (w) res=(1 << r0);
            else  res=(0 | (1 << r5));
    } // end movgt at 0x48
    if (instruction==76) { // registers used in instruction ble
          1c,  at 0x4c
            if (w) res=0;
            else  res=0;
    } // end ble at 0x4c
    if (instruction==80) { // registers used in instruction
         ldmia sp!,{r4,r5,pc,} at 0x50
            if (w) res=((1 << sp) | (1 << r4) | (1 << r5) | (1
                 << pc));
            else  res=( 1 << sp);
    } // end ldmia at 0x50

return res;
}

bool stall() {
bool resmem=false,resreg=false;
bool tododec=true,todomem=true,todowb=true;

          // check for registers dependencies
          // one of my read reg is written to by MEMORY_STAGE
               or WRITE_BACK_STAGE
          if (pPC[DECODE_STAGE]==-1) tododec=false;
          if (pPC[MEMORY_STAGE]==-1) todomem=false;
          if (pPC[WRITE_BACK_STAGE]==-1) todowb=false;

          resreg=(tododec & (reg(pPC[DECODE_STAGE],0))) & ((reg
               (pPC[MEMORY_STAGE],1) & todomem )| (reg(pPC[
               WRITE_BACK_STAGE],1) & todowb)) ;

          if (type_of(pPC[DECODE_STAGE])==G2LDR && tododec &&
               type_of(pPC[MEMORY_STAGE])==G2STR && todomem ) {
                    resmem=(dataAdr[DECODE_STAGE] == dataAdr[
                         MEMORY_STAGE]);
          }

          return resreg || resmem ;
}
```

Listing 4. C Code for `reg`; Use `reg(in,1)` (resp. `reg(in,0)`) for written to (resp. read from) registers