Timing Analysis of Binary Programs with UPPAAL

Franck Cassez
NICTA and University of New South Wales, Sydney, Australia*
Jean-Luc Béchennec
LUNAM University, CNRS, IRCCyN, Nantes, France

***Abstract*—We address the problem of computing accurate Worst-Case Execution Time (WCET). We propose a fully automatic and modular methodology based on *program slicing* and *real-time model-checking*. We have implemented our methodology and applied it to standard benchmarks. To further validate the approach, we also compare our results to the real execution times of the programs measured on a real board.**

I. Introduction

Embedded real-time systems are composed of a set of tasks (software) that run on a given architecture (hardware) that are subject to timing constraints, the most prominent one being schedulability. Checking schedulability requires upper bounds for the execution times of each task. Performance wise, determining tight bounds is crucial since rough over-estimates might either result in a set of tasks being wrongly declared non schedulable, or lead to the choice of an over-powered and expensive hardware.

The WCET Problem. Given a binary program P, input data d, and hardware H, the *execution time* of P and d on H, is the number of processor cycles from the beginning until the end of the computation of the result of P for d. The *worst-case execution time (WCET)* of P on H[1], WCET(P, H), is the supremum, over all input data d, of the execution times of P for H. The WCET problem is to compute 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. Notice that this does not mean that the problem is tractable though.

Programs run on complex architectures featuring multi-stage *pipelines* and *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 architectures.

Methods and Tools for the WCET Problem. The reader is referred to [1] for an exhaustive presentation of WCET computation techniques and tools. There are two main streams for computing WCET: *Simulation-based* [2], [3] methods are based on experiments i.e., running the program on some input data, using a simulator or the real platform. The upper bound of the simulations is unreliable if the simulation is not exhaustive, and this scheme is often *unsafe* for safety critical embedded systems. *Verification-based or static methods*) rely on the computation of an *abstract* graph (AG) obtained from the control flow graph (CFG) and an abstract model of the hardware. The AG generates a superset of the set of all feasible paths and thus the largest execution time on the AG is an upper bound of the WCET. Such methods produce *safe* WCET and tools implementing this scheme are Bound-T [4] and aiT [5], [6].

The verification-based tools mentioned above have some disadvantages though: (1) they rely on the construction of a CFG, and the determination of *loop bounds* and often they both require manual annotations. The algorithms implemented in the tools use both the program and the hardware specifications to compute the CFG fed to an Integer Linear Programming (ILP) solver. (2) the correctness of the abstract models used in the tools is not easy to establish; the *efficiency* of the procedures implemented in the tools strongly rely on the assumption that the hardware is *timing anomaly free* [7], [1], [8]. The architectures of the tools themselves are monolithic: it is not easy to add support for a new hardware. (3) to the best of our knowledge, the accuracy of the results has not been thoroughly evaluated and results are not compared (using benchmarks) against execution times measured on a real platform. Some tools report comparisons with ARMulator, which is not cycle accurate. The above mentioned drawbacks (1), (2) and (3) are reported in the *WCET'11 Challenge Report* [9].

Related Work. 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 data from the cache), formal models like *timed automata* (TA) [10] and state-of-the-art *real-time model-checkers* like UPPAAL [11] appear well-suited to address the WCET problem. A. Metzner already showed [12] that model-checking could be used to compute the WCET for programs running on pipelined processors. More recently,

*NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

[1]We assume that H has a single initial state. A more precise definition using hardware initial states is given on page 3, equation (1).

Lv *et al.* [13] combined AI techniques with real-time model-checking to compute WCET on multi-core platforms.

The use of network of timed automata (NTA) and UPPAAL for computing WCET on pipelined processors with caches was already reported in [14], [15] where the METAMOC[2] method is described. METAMOC consists in: 1) computing the CFG of a program, 2) composing this CFG with an NTA model of the processor and the caches and 3) computing the longest path (time wise) in this NTA. This framework is very elegant yet shares some of the disadvantages we mentioned previously: (1) METAMOC relies on a *value analysis* phase (to compute a CFG) that may not terminate, (2) some programs cannot be analyzed (if they contain register-indirect jumps), (3) manual annotations (for loop bounds) are still required on the binary program, and (4) the *unrolling* of loops is not safe for some cache replacement policies (FIFO). In [16] we have already reported some similar results on the computation of WCET: we used NTA to model the caches and pipeline but the computation of the CFG was done in a totally different manner.

In this paper we introduce *program slicing* as a key ingredient to make the model-checking approach feasible. Program slicing to compute WCET was considered in [17] but 1) slicing is performed on *structured programs* in the intermediate “NIC” format (different from binary) and 2) the authors assume that the CFG of the program is available. In this paper we are slicing *binary* programs and computing *automatically* the CFG.

Our Contribution. Compared to [16], we propose three new original contributions: (1) a method to automatically compute a CFG (Section VI) and a reduced abstract program equivalent WCET-wise to the original program (Section V); (2) a *modular* technique (Section IV) to compute WCET; and (3) a comparison of computed WCET with measured WCET on a real hardware. Noteworthy, our method is robust against timing anomalies.

An extended version of this paper [18] is available at http://www.irccyn.fr/franck/wcet.

II. ARCHITECTURE OF THE ARM920T

The hardware we model in this paper is an Armadeus APF9328 board which bears a 200MHz Freescale MC9328MXL micro-controller with an ARM920T processor which embeds an ARM9TDMI core that implements the ARM v4T architecture. Formal models of this board are discussed in Section VII and available as UPPAAL TA from http://www.irccyn.fr/franck/wcet.

The ARM architecture is a *Reduced Instruction Set Computer* (RISC) architecture. The instruction set consists of fixed size instructions and a few simple addressing modes. There are 16 general purpose registers r_0 to r_{15} (r_{13} to r_{15} are also called sp: stack pointer, lr: link register and pc: program counter), specialized memory transfer instructions (load/store), and data-processing instructions that operate on registers only. Other relevant features are *multiple* load/store instructions and *conditional* execution of instructions. The ARM920T uses a 5-stage *execution pipeline* the purpose of which is to execute concurrently the different stages (**F**etch, **D**ecode, **E**xecute, **M**emory, **W**riteback) needed to perform an instruction. The optimal execution flow may be slowed down when pipeline *stalls* occur.

Fact 1: In order to determine pipeline stalls, it is sufficient to know what registers are read from/written to by an instruction and if an instruction is performed (if conditional). Both instruction/data caches of the ARM920T have the same architecture. They are 16KB, 64-way set associative caches with 8 sets and 512 32-byte lines. Replacement policy may be set to pseudo-random or round-robin (FIFO). Transfers between the caches and main memory are serialized.

Fact 2: To determine cache misses/hits it is enough to know the current state of the cache and (i) the location (address) of an instruction for the instruction cache and (ii) the addresses referenced by an instruction for the data cache.

III. PROGRAM SEMANTICS

We make the following assumptions on a binary program P: (A1) P always terminates and there is a uniform upper bound $\kappa(P)$ s.t. for every input data, P terminates in at most $\kappa(P)$ steps; (A2) P does not contain recursive calls.

(A1) rules out programs that have to traverse an array with *unknown* size at compile time: the (maximal) size of the array has to be hard encoded in the program. A program doing a binary sort on an array of at most 10 items satisfies (A1) as there is a bounded number of steps over all input data. A program that first reads the size of an array (unconstrained) and does a binary search does not satisfy (A1): the WCET in this case is unbounded. (A2) ensures that the CFG can be computed automatically.

We let $\mathcal{P} = \{le, gt, \cdots\}$ be a finite set of *predicate variables* to hold the truth values of the *conditions* used in the conditional instructions of the program.[3] An example program $FIBO_0$ (its CFG) is given in Fig. 1 (the full text is given in appendix A.) The semantics of programs is given in terms of assignments to registers, predicates and memory locations. For instance, a comparison operator sets the truth value of the predicate variables.

To compute the execution times of program runs, we view the hardware as an abstract machine H that reads sequences of *triples* (ι, A, d) generated by P. H acts as a transducer and outputs the *time* (in cycles) it takes to process such a sequence. A triple (ι, A, d) consists of an instruction $\iota =$

[2]METAMOC stands for Modular Execution Time Analysis using Model Checking.

[3]In the ARM 32-bit instruction set, the truth values are stored in the status bits N, Z, C, V.

$(\ell : i)$, the set A of memory addresses ι references and a boolean d that indicates whether ι is performed or not.[4]

We use these triples (and sequences thereof) because they contain the relevant information to compute the execution time of a run: (i) pipeline stalls are fully determined by (a) the first component of the triple ι that contains the complete description of the instruction and the read/written registers and the (immediate) constants and (b) the value of d (Fact 1); (ii) cache hits/misses are determined by the set A (Fact 2).

Given a sequence of triples w and a state γ of H, $\mathsf{time}_H(\gamma, w)$ is the execution time of w from γ. The set of *initial* states for program P and registers/memory contents is denoted I (register *pc* is set to the initial instruction) and the contents of the registers, predicates and main memory is an admissible value in the finite set $\mathcal{D}$. Notice that there can be many initial states as the input data of P can range over large sets. P also has a set of *final* states, F, that can be defined by the value of register *pc* (the last instruction of P). The language $\mathcal{L}_I^F(P)$ of P is the set of sequences of triples generated by P that start in I and end in F. As we assume that P terminates for each input data, this language is finite ($\mathcal{D}$ is finite).

IV. Computation of the WCET

Modular Definition of WCET. WCET(P, H) is completely determined by $\mathcal{L}_I^F(P)$ and an initial state γ of H:

$$\text{WCET}(P, H) = \max_{w \in \mathcal{L}_I^F(P)} \mathsf{time}_H(\gamma, w). \tag{1}$$

Computing WCET(P, H) thus amounts to: (i) generating $\mathcal{L}_I^F(P)$, (ii) feeding H with each $w \in \mathcal{L}_I^F(P)$ and tracking the maximal execution time. It is easy to see that $\mathcal{L}_I^F(P)$ and H are independent of each other. In this sense the definition we have is *modular* as generating $\mathcal{L}_I^F(P)$ and building H are two separate problems. Moreover, the algorithm to track the maximal execution time is also independent from the language $\mathcal{L}_I^F(P)$ and the hardware H. The initial state of H may not be unique (caches lines might be *dirty*.) We can accommodate this in Equation 1 and the UPPAAL models as well by setting the initial state of the hardware.

Abstract Domain Semantics. In order to take into account all the possible values of the input data, we use an *extended domain* for the values of the registers and memory contents. Let $\mathcal{D}_\perp = \mathcal{D} \cup \{\perp\}$ be the extended abstract domain with $\perp$ the *unknown* value. The semantics of instructions is extended in a straightforward manner to this domain: for instance, the abstract semantics[5] of `add r0,r1,#1` is given by $[\![r_0]\!] = \perp$ if $([\![r_1]\!] = \perp)$ and $[\![r_1]\!] + 1$ otherwise. The semantics of comparison instructions e.g., `cmp r0,r1` is extended as well to $\mathcal{D}_\perp$: for predicate le, $[\![le]\!] = \perp$ if $(([\![r_0]\!] = \perp)$ or $([\![r_1]\!] = \perp))$ and $([\![r_0]\!] \leq [\![r_1]\!])$ otherwise. When a conditional instruction is encountered and the condition is $\perp$ (unknown), the abstract semantics generates two successors: one where the condition is TRUE and the other where the condition is FALSE. The abstract semantics of P on the abstract domain is defined in the usual standard way by the abstract semantics of each instruction. Let $\mathcal{L}_\perp(P)$ be the set of sequences of triples (traces) generated by runs of P in the abstract domain. As the abstract semantics over the abstract domain allows more runs than in the concrete domain, the set of traces generated by P on the extended domain is a super set of the set of possible traces of P: $\mathcal{L}_I^F(P) \subseteq \mathcal{L}_\perp(P)$ and hence replacing $\mathcal{L}_I^F(P)$ by $\mathcal{L}_\perp(P)$ in Equation 1 gives an upper bound of WCET(P, H) that we denote WCET$_\perp(P, H)$ in the sequel.

WCET Computation. We can reduce the computation of the WCET to a *reachability* problem on an NTA. We first build an automaton *Aut*(P) that generates $\mathcal{L}_\perp(P)$: the initial states of the automaton are given by I and the successor states by the abstract semantics of each instruction. The hardware components, pipeline stages, caches and main memory are modelled by TA because they induce timing constraints: duration of transaction between the cache and main memory, duration of a cache hit/miss, processing time in each stage of the pipeline.[6] The model of hardware H is thus specified by an NTA *Aut*(H). Feeding H with $\mathcal{L}_\perp(P)$ amounts to building the *synchronized* product *Aut*$(H) \times$*Aut*(P) (the final states are the states where the last instruction of P flows out of the last stage of pipeline). Computing the WCET amounts to determining the longest path (time-wise) and can be done with UPPAAL [19] using a global clock and the *sup* operator. Notice that to do this we have to explore the whole state space of *Aut*$(H) \times$ *Aut*(P). To handle large case studies, we need to reduce the state space as much as possible. For instance, we should avoid generating two runs of the program P that give the same sequences of triples (same trace) as both runs will result in the same execution time (from the same initial state of H). Thus if we can *minimize* *Aut*(P) while still generating the same sequences of triples we reduce the number of states of *Aut*(P) as well as the number of explored paths in *Aut*$(H) \times$ *Aut*(P). In the next section we describe how to compute a reduced program P' that generates the same language as P.

V. Computation of a WCET-Equivalent Program

In this section we show how to use program slicing to compute a *WCET-equivalent* program. Given two programs P and P', if $\mathcal{L}_\perp(P) = \mathcal{L}_\perp(P')$ then WCET$_\perp(P, H) =$ WCET$_\perp(P', H)$. Our goal is thus to compute such a WCET-equivalent program P', which has less states than P yet contains enough information to generate $\mathcal{L}_\perp(P)$. We can

[4]As said previously, some instructions are conditional and performed only if the condition is true; if the condition is false they bahave as *nop* (increment *pc* only).

[5]$[\![x]\!]$ reads "content of predicate/register/memory cell x".

[6]Some instructions (MUL/MLA/SMULL) have data dependent durations. In this case, in the TA model of the execute stage of the pipeline, the duration is in an interval.

define P' as a *slice* of P: a *slice* is a subset of instructions of P together with a subset of the variables used in P.

The purpose of *program slicing* [20] is to compute a program *slice* by removing some statements of the original program s.t. the slice computes the same values for some variables at some given points in the programs. The points of interests together with the variables of interest at these points constitute the *slice criterion*. The main result of [20] is that, given a slice criterion, it is possible to compute a sub-program, the slice, that will produce exactly the same values for the variables at the points of interest.

In this work, we use program slicing for two separate purposes: 1) to generate the CFG (Section VI) by determining the target addresses of dynamic jumps and 2) to identify the instructions that have to be tracked to preserve the timing information (rest of this section).

The slice criterion contains memory transfer and conditional instructions: this is the minimum set of instructions to track in order to generate the sequences of triples (ι, A, d) for a program. Program slicing usually discards a subset of instructions irrelevant to the slice criterion. In our case, the reduced program P' is not defined as the sliced program alone as it would obviously not generate the same sequences of triples. P' is the original program P operating on the reduced set of slice variables with the semantics of instructions altered as follows: if an instruction is in the slice, its standard abstract semantics applies; otherwise, its semantics is *nop* (increment pc register without modifying the other variables.) No instruction of P is discarded but the semantics of the instructions not in the slice is interpreted as *nop*. This preserves the generated triples. The correctness of this approach i.e., $\mathcal{L}_\perp(P) = \mathcal{L}_\perp(P')$, is ensured by the property of a slice (and the choice of the slice criterion.) Let us first assume that the CFG of P is known. Slicing binary programs is still not easy because (i) the names of the variables used in a program have disappeared and are references to memory locations using a base register and an offset register and (ii) the computation of a slice is based on an iterative solution of *data-flow* equations on the set of relevant[7] variables for each instruction in the CFG of P. The addresses of the relevant variables for an instruction might not be known at compile time. Consider the instruction `str r0,[sp,#4]`. It semantics is $[\![[\![sp]\!]+4]\!] := [\![r_0]\!]$ but the value of sp (stack pointer) is fixed only when the program executes. Other instructions like `str r2,[r1, r3 lsl #2]` ($[\![[\![r_1]\!] + ([\![r_3]\!] \times 2)]\!] := [\![r_2]\!]$) might (read or) write to arbitrary memory locations. To overcome this difficulty, we start by defining the sets of *REF* (read) and *DEF* (written) variables for each instruction as follows:

- for instructions that do not involve memory transfers, we know the exact set of *DEF* and *REF* variables e.g., for instruction i = `add r2,r1,#1` $REF(i) = \{r_1\}$ and $DEF(i) = \{r_2\}$.
- for instructions that use stack references, e.g., i = `push(lr)`, we define $REF(i) = \{lr, sp\}$ and $DEF(i) = \{sp, stack\}$; all we know at that stage is that the *stack* is modified without any knowledge of the precise index in the stack that is involved.
- for instructions that make main memory references, we abstract away their contents: we assume that the content of the memory outside the stack is unknown. For instruction i = `str r2,[r1, r3 lsl #2]` we set $REF(i) = \{r_1, r_2, r_3\}$ and $DEF(i) = \varnothing$ (the content of the main memory not in the stack is always unknown.)

[7]Relevant variables for an instruction are the read from/written variables.

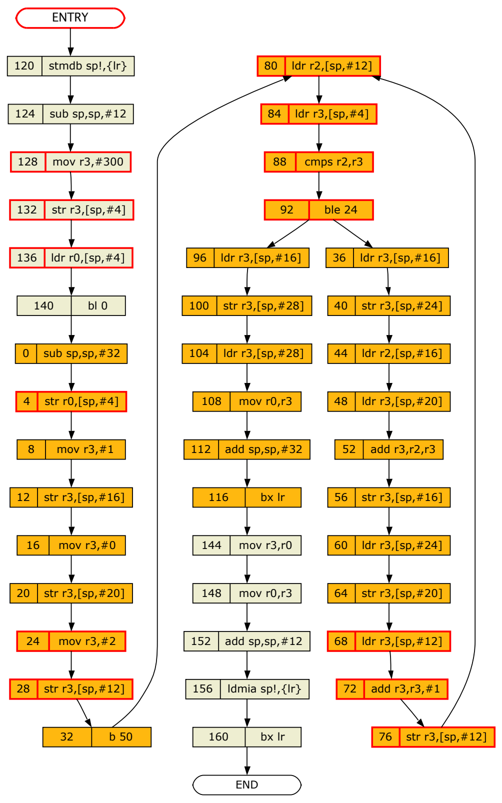

Figure 1. WCET-equivalent Slice for $FIBO_0$.

In a first phase we define a slice criterion SC_0 that consists of every instruction that reads[8] or writes the stack pointer (for $FIBO_0$, Fig. 1, SC_0 contains $(0, \{sp\})$, $(4, \{sp\}), \ldots, (120, \{sp\}), \ldots$. We compute the corresponding slice, simulate it (initial values of the stack pointer must be given) and collect the possible values of sp at the points defined by SC_0. This enables us to define more precisely the set of *DEF* and *REF* variables for each instruction referencing the stack pointer: we can now replace the variable *stack* (in *REF* and *DEF*) by actual stack indices.

[8]In the implementation we start with instructions writing to sp and then forward propagate the values to instructions reading sp.

For instance, if the set of values of register sp for instruction $i =$ `str r0,[sp]` is $\{60, 64\}$, we set $\mathit{REF}(i) = \{sp, r_0\}$ and $\mathit{DEF}(i) = \{stack_{60}, stack_{64}\}$.

Given this precise definition of the *DEF* and *REF* variables for each instruction, we can compute a WCET-equivalent program in a second phase. The slice criterion we start with is SC_1 and contains (as said at the beginning of this section) the instructions that (i) make memory transfers or (ii) are conditional. This ensures that the sliced program generate the same triples (ι, A, d) for all these instructions.

An example of a slice for FIBO_0 is given in Fig. 1: there are 7 variables in the slice ($\{pc, r_0, r_2, r_3\}$ and 3 stack values), 13 instructions (highlighted in red) out of 41 (the ones not in the slice do not modify the sliced variables and solely increment pc.) Notice that instructions 8 and 16 are not in the slice: they load values in r_3 which are pushed onto the stack but these values do not influence the control flow later on. This enables us to avoid manual *loop counter annotations*. The variables in the slice suffice to precisely control the number of loop iterations. Indeed, as witnessed by example in Fig. 1, all the instructions that control the loop are in the slice and the variables used for temporary storage (on the stack) are included as well. It follows that all the necessary variables and updates are captured by the slice.

A 2-stage Slicing Algorithm. Computing a WCET-equivalent program using the $stack$ variable only (without pre-slicing with sp) generates large slices: indeed, any instruction that reads the stack depends on any instruction that writes to the stack. Computing first the precise values encountered for sp enables us to obtain small slices in the subsequent step: an instruction that reads stack variable $stack_4$ will not depend on an instruction that writes $stack_{16}$. The slices computed for the first phase are very small and cheap to compute as the stack pointer is usually used to allocate some space on the stack on function calls and release it when the function returns.

Table I, page 8, column *"Slice"* (a/b) gives, for each program P, the number of instructions a that are in the slice of P compared to the total number of instructions b in the CFG of P; it clearly shows a drastic reduction (and the number of slice variables is also very small compared to the full set of registers and full stack). This is essential for the scalability of the method as discussed in Section VIII. In the worst-case the slice contains all the nodes of the CFG.

Correctness and Termination. The correctness of the algorithm follows from the slice criterion: as it comprises of the conditional and memory transfer instructions, we guarantee that the slice preserves the triples generated by $\mathcal{L}_\bot(P)$. Termination is not guaranteed as we simulate the programs in the extended abstract domain. Nonetheless, for all the programs we processed so far (Table I), the abstract simulation terminated.

VI. Computation of the Control Flow Graph

The algorithm to automatically compute the CFG consists in three steps that are iterated:

- **Step 1, Unfold.** Given a set of source nodes S, unfold the CFG as much as possible, i.e., until dynamic branching instructions are encountered. This gives a (partial) CFG C where some dynamically computed branching (register-indirect jumps) are considered as nodes to be resolved (set R).
- **Step 2, Slice.** Slice C with the slice criterion that corresponds to the set R.
- **Step 3, Simulate.** Simulate the previous slice and compute the successors nodes $succ(R)$ for R. Add edges from the nodes in R to their computed successors $succ(R)$. Set S to the set of nodes in $succ(R)$ that are not final nodes (of the program) and goto Step 1.

The algorithm starts with S being the entry point of the program to be analyzed. It stops when S is empty. When the algorithm terminates we have the CFG of the program.

We illustrate the algorithm on program FIBO_0. In the first iteration, Step 1, $S = \{120\}$, we unfold the program up to instruction 116 which is a dynamically computed branching: `116: bx lr` "branch to $[\![lr]\!]$" and $[\![lr]\!]$ is unknown at compile time. In this CFG (Fig. 2, left), the successor of instruction 116 is unknown (stored in lr) and thus the unfolding terminates at this node (this is denoted by the special successor node $EXIT_{0xN}$) and $R = \{116\}$. To compute the successor of this node we slice (Step 2) this CFG with the slice criterion $SC_0 = \{(116, \{lr\})\}$ that contains instruction 116 and the associated variable lr. The sliced program (red nodes in Fig. 2, left) is composed of instructions 140 ("(b)ranch to 0 and save return address to (l)ink register lr") and 116. Simulating (Step 3) this two-instruction program we get the possible value of lr at instruction 116 which is 144: thus a new node $succ(R) = \{144\}$ is created and an edge from 116 to 144. As can be readily seen this slice is very small (two instructions). When we simulate the slices, we set the initial values of sp and pc to known values and set the initial values of the other registers to $\bot$.

In the next iteration of the 3 steps, we start with $S = \{144\}$ (the newly discovered successor of 116). We unfold this CFG (Step 1) from 144 to obtain a CFG (Fig. 2, right) comprising of all the instructions up to instruction $R = \{160\}$ which is a dynamically computed branching (register-indirect jump) again.

This second CFG is sliced (Step 2) to compute the successor of instruction 160: the new slice (red nodes in Fig. 2, right) contains 6 nodes, 120, 124, 0, 112, 152, 156 and 160 with associated variables. Notice that we first have to determine the stack variables used by instructions $156, 120$ as they are used to store the return address on top of the stack: this is done using the algorithm described in the previous section. We can then simulate (Step 3) the new slice and compute the successor of node 160 which is a program final node and thus the algorithm terminates. The final CFG is given in Fig. 1.

The computation of the possible values of sp described in Section V is actually performed when computing the CFG. When we have computed the final CFG we also have the possible values of the stack indices at each node of the CFG.

Remark 1: If the abstract semantics simulation generates an unknown target value $\perp$ for pc, the construction of the CFG aborts. This never happened on the benchmark programs and can only happen if the (unknown) input data have an influence on a dynamically computed target. Dynamically computed targets are used as "return" statements in functions and thus rarely depend on input data.

Correctness and Termination. Correctness is guaranteed by the slice criterion and the property of slices: each dynamically computed target of a branching instruction is in a partial unfolding of a program. Termination is not guaranteed as we simulate the program on an extended abstract domain.

VII. Hardware Formal Models

The hardware formal model of the platform described in Section II is a NTA comprising of the following components: pipeline stages, instruction and data caches, write-buffer and main memory. The hardware model is deterministic (we use priorities in UPPAAL to enforce this.)

The pipeline, caches and write buffer operations greatly impact the WCET. This is especially true for caches as small variations like the base address of the data can double the WCET. To get tight bounds, we have to build very accurate models of the hardware. The official documentation, the ARM9TDMI Technical Reference Manual [21], gives some information (mostly examples) about the pipeline timings and caches. It is not detailed and systematic enough to cover all the situations and build a formal model of the hardware. To overcome this problem and build accurate formal models, we have carefully designed custom binary programs to stress particular features of the hardware and determine the precise timing of some sequences of instructions, caches, write buffer and memory accesses.

Notice that we used this method because we were not able to obtain detailed hardware specifications from the vendor, but ideally, formal models of the hardware (or good abstractions thereof) could be directly provided by the vendor. The timed automata for the caches, pipelines are available at www.irccyn.fr/franck/wcet. Some of the models are rather involved and we do not detail them in this paper due to lack of space.

Compared to other tools, our hardware formal models are available as UPPAAL timed automata. It is thus possible to check their accuracy and level of details. METAMOC models simpler: our formal models are a lot more detailed and capture advanced features of the ARM920T hardware like the write buffer, alignment.

VIII. Implementation & Experiments

Implementation. The binary programs are computed from C/C++ programs with the GCC tool suite for ARM (gcc, objdump) from Codesourcery.

We have implemented the construction of the CFG (Section VI) and the computation of the WCET-equivalent program (Section IV). Together with a parser of ARM binary programs it comprises of three thousand C++ lines of code. We have implemented very efficient versions of *post-dominators algorithms* [22] and *post dominance frontiers algorithms* [23] as they are used intensively for computing slices. We have also implemented a functional software simulator for ARM programs to simulate the slices and collect the results. Using our implementation we can generate the WCET-equivalent program together with the model of the hardware in a UPPAAL.

Methodology. The precise methodology to measure the execution time on our testbed is described at http://www.irccyn.fr/franck/wcet. What should be noticed is that we (really) measure, on the real hardware, the execution times (in cycles) of the programs. For programs with multiple paths, we supply the data that *should*[9] produce the WCET: this way we obtain a measured lower bound of the WCET.

Experiments. Results on the Mälardalen University benchmarks [24] are reported[10] in Table I (models, programs are available from http://www.irccyn.fr/franck/wcet). Regarding the benchmarks themselves, we point out that:

- the difficulty of computing the WCET is not related to the *size* of the program; some programs are huge but contain few paths, others are very compact but have a huge number of paths.
- they are designed to be representative of the difficulties encountered when computing WCET: for instance janne-complex contains two loops and the number of iterations of the inner loop depends on the current value of the counter of the outer loop (in a non regular way).
- we have experimented with different compiled versions (O0, O1, O2) of the same program because the binary code produced stresses different parts of the hardware. Moreover, the strongest optimization (O2) produces a binary program which is very different from the original C program and it is almost impossible to infer loop bounds automatically (a disadvantage for standard techniques). Our method is automatic and accurate.
- we have increased the number of iterations of the benchmarks (e.g., we compute the WCET of $Fib(300)$

[9]Of course this only gives a lower bound on the execution time as we cannot prove that the supplied data corresponds to the worst case. Worst-case input data are included in the source code of the benchmarks.

[10]We also compute the Best case execution time (BCET) using the *inf* UPPAAL operator.

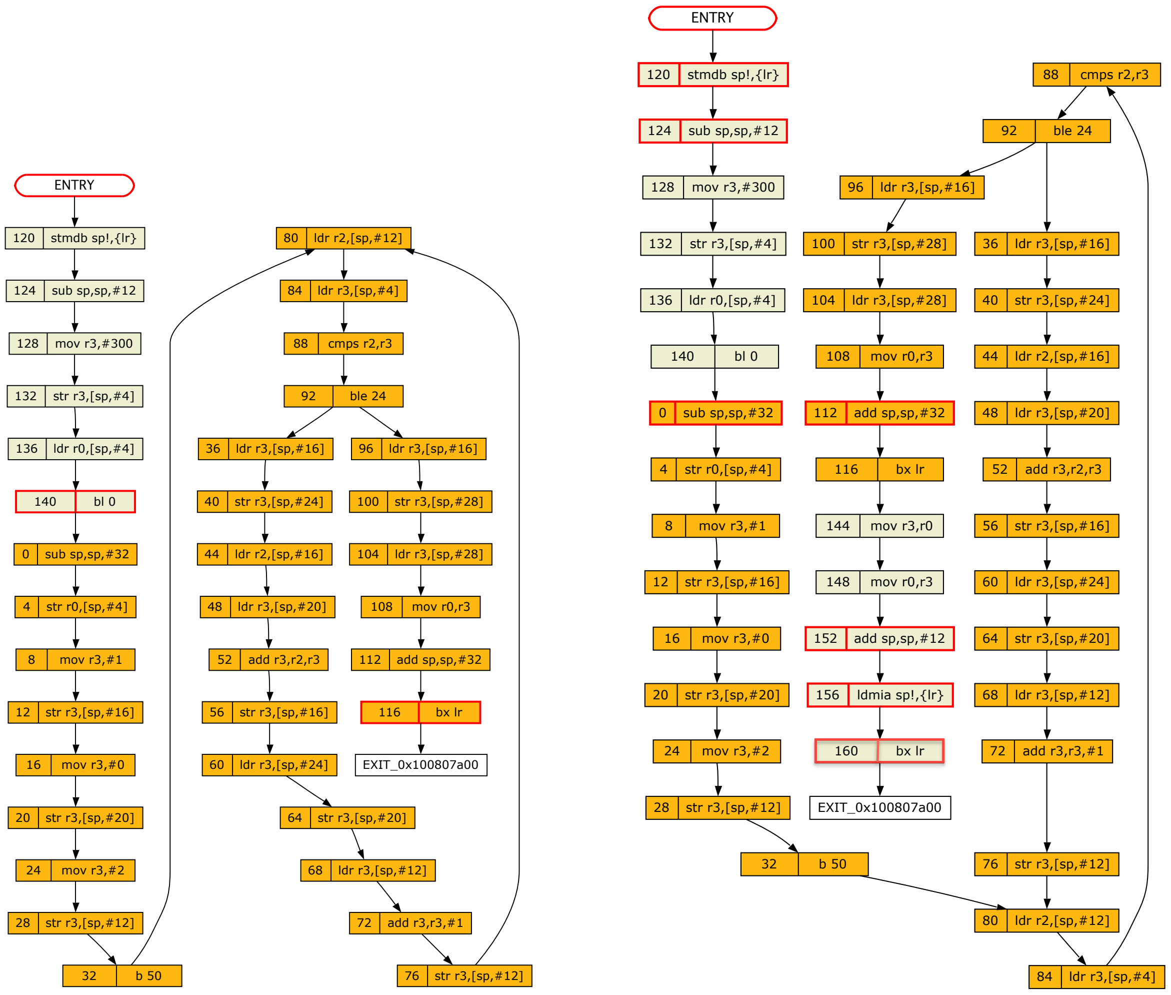

Figure 2. Unfoldings of the CFG of $FIBO_0$.

instead of $Fib(30)$[11]); this way, modelling errors/inaccuracies (e.g., producing an extra cycle per loop in our model) will incur a magnified over-approximation in the result and be glaring.

The results in Table I fall into three main sections:

Single-Path programs. For this program there is only one execution path and thus for a given initial hardware state the measured execution time is the WCET. The results show that the abstract models (program and hardware) we have designed are adequate for obtaining tight bounds for the WCET. Even for janne-complex and its inner loop counter that depends on the outer loop counter, the maximum error is 2.4%. This validates the accuracy of the program/hardware model we have computed.

Single-Path programs with data dependent instruction durations. Some programs use variable execution time instructions like MUL/MLA/SMULL. For these instructions the time spent in the E stage is within an interval (this is another strength of using time automata as timing can be easily and precisely defined including uncertainty). This explains the difference between the computed and the measured WCETs because in the measured WCET the worst-case duration for the MUL/MLA/SMULL instructions is never encountered. In this case, column "Error (%)" of Table I does not represent the over-approximation of the computed WCET but rather the under-approximation of the measured WCET with the supplied input data.

Multiple-path programs. These programs contain branchings that are input data dependent. The measured WCET is the execution time (on the hardware) obtained with input data that should produce the WCET. As emphasized earlier we cannot guarantee that the input data produces the WCET. Our computed WCET however considers all the possible input data. Notice that when the measure M is less than 1000, measurement error exceeds 1%.

An important question is the practical effect of slicing and

[11]The computation of $Fib(300)$ results in an arithmetic overflow but we can still compute the time it takes to compute the result.

Program$^{\oplus}$	loc†	UPPAAL Time/States Explored¶	Computed BCET/WCET (C)	Measured BCET/WCET (M)	Error (%)‡	Slice§
Single-Path Programs						
fib-O0	74	2s/74181	8098	8064	**0.42%**	47/131
fib-O1	74	0.6s/22333	2597	2544	**2.0%**	18/72
fib-O2	74	0.3s/9711	1209	1164	**3.8%**	22/71
janne-complex-O0*	65	1.7s/38038	4264	4164	**2.4%**	78/173
janne-complex-O1*	65	0.5s/14600	1715	1680	**2.0%**	30/89
janne-complex-O2*	65	0.5s/13004	1557	1536	**1.3%**	32/78
fdct-O1	238	21s/60534	4245	4092	**3.7%**	100/363
fdct-O2	238	3.24s/55285	19231	18984	**1.3%**	166/3543
Single-Path Programs‡ **with MUL/MLA/SMULL instructions (duration of instruction depends on data)**						
fdct-O0	238	124s/85008	11242/11800	11448	**3.0%**	253/831
matmult-O0*	162	217s/10531262	502849/529250	511584/528684	**0.1%**	158/314
matmult-O1*	162	25s/1112527	129967/156367	127356/153000	**2.2%**	71/172
matmult-O2*	162	121s/6780931	122045/148299	116844/140664	**5.4%**	75/288
jfdcint-O0	374	92s/100861	12726/12918	12588	**2.6%**	159/792
jfdcint-O1	374	12s/35419	4880/5072	4668	**8.6%**	25/325
jfdcint-O2	374	5.38s/175661	[16746,16938]	16380	**3.4%**	56/2512
Multiple-Path Programs						
bs-O0	174	30s/1421274	478/1068	1056	**1.1%**	75/151
bs-O1	174	23s/1214673	321/738	720	**2.5%**	28/82
bs-O2	174	12s/655870	273/628	600	**4.6%**	28/65
cnt-O0*	115	4s/77002	9025/9027	8836	**2.1%**	99/235
cnt-O1*	115	1.4s/27146	4123/4123	3996	**3.1%**	42/129
cnt-O2*	115	9s/11490	3067/3067	2928	**4.6%**	39/263
insertsort-O0*	91	598.98s/24250738	3133	3108	**0.8%**	79/175
insertsort-O1*	91	353.80s/11455293	1533	1500	**2.2%**	40/115
insertsort-O2*	91	11.68s/387292	1326	1320	**0.4%**	43/108
ns-O0*	497	60s/3064316	940/30968	30732	**0.8%**	132/215
ns-O1*	497	8s/368720	605/11701	11568	**1.1%**	61/124
ns-O2*	497	55s/1030746	441/7280	7236	**0.6%**	566/863

$^{\oplus}$ file-Ox indicates that file was compiled using gcc -Ox
† lines of code in the C source file
‡ $\frac{(C-M)}{M} \times 100$ computed using the upper bound for C and M
§Instructions in Slice/Instructions in Program
*Program selected for the WCET Challenge 2006
¶UPPAAL 4.1.11/Intel Pentium 5/3.1Ghz/16GB

Table I
SUMMARY OF THE RESULTS.

the use of the WCET-equivalent program. What happened when no slicing is used is given by the METAMOC results[12]: they show (http://metamoc.dk/, benchmark results), that even with abstract caches, and a large amount of memory (32GB) for UPPAAL, it is impossible to compute a WCET for large programs (compiled with options O0, O1) and only the (small) programs compiled with option O2 can be handled (and with simple cache models). This clearly shows that slicing and the computation of a reduced WCET-equivalent program is a critical step in the model-checking method.

Statistical Model-Checking. The WCET is an upper bound and as any upper bound it may happen only for a few input data. A complementary approach is to get a *distribution* of execution times to gain more insight on the timing behaviour of the program. This is usually done by simulating the program. Doing this requires setting up a testbed, providing (random) input data and measuring execution times.

The UPPAAL models obtained after the slicing phase can also be used to for simulation. This is an advantage of our method that we have executable UPPAAL formal models. This is a very nice feature as it is possible to use the statistical model-checking [26] approach to gain more insight into the distribution of the execution times. It is also a good alternative to the computation of the WCET using the exhaustive approach in case it is impractical (due to the state explosion problem).

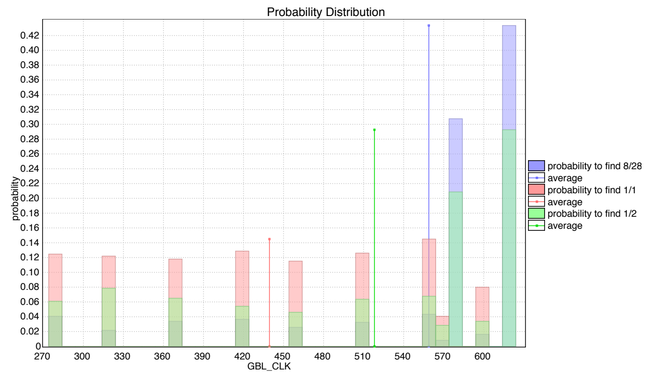

Figure 3. Distribution of Execution Times for `bs-O2`.

The distribution of execution times for program `bs` is given in Figure 3 ('k/n' in the legend represents p). This program performs a binary search on an 800 elements array. The "probability to find" the element searched for is p, and if it is in the array we assume it can be at any position with uniform probability $\frac{p}{800}$ (this the probability that the element is not found is $1-p$). The probability distribution, Fig. 3, for this program can be computed by UPPAAL-SMC [26].

[12]We cannot compare the WCET estimates from METAMOC with ours as the hardware is different.

IX. Strengths and Limitations of our Method

The most valuable advantage is certainly the automatic computation of the CFG and the reduced WCET-equivalent programs. This avoids the tedious and error-prone task of loop bound annotations of all other methods and tools. The slicing algorithm [22] we have implemented is linear in the size of the CFG and and scale up to very large programs. On the sliced program P' we can apply standard model-checking (without time) and compute: loop bounds, maximum stack size, maximum number of cache misses, etc. There are other advantages for using model-checking techniques: the model-checker can output a witness program trace for the longest path; we can check whether this trace is feasible and if so generate corresponding initial input data; in case the witness trace is not feasible, counter example guided refinement (CEGAR) can be carried out to refine the model.

The use of TA models for the hardware is also a clear advantage as the specifications are readable, formal, compact, executable (we can simulate them in UPPAAL) amenable to changes/substitutions (e.g., LRU vs FIFO replacement, *always-miss* caches) and can be tested/validated (using UPPAAL). Moreover, we can accommodate changes in processor speed as well: for instance, in the first 10ms the processor runs at half speed and then switches to full speed (this can be modeled using an extra TA.) Modularity

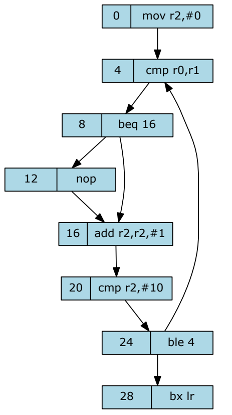

Figure 4. A short Program

is a nice and important feature: (1) hardware models can be designed and validated, and derived from VHDL specification; replacing a hardware by another amounts to a simple selection of hardware formal model templates in UPPAAL. (2) the program model (slicing) can be built based on the sole knowledge of the semantics of the language; support for other assembly languages is made easy. (3) the model-checking algorithm can be improved: strategies to prune the state space can be integrated or multi-core versions can be safely used to speed up the computation.
The current limitation of the approach is the model-checking phase but there is room for improvements in this area. Contrary to well-established static analysis based techniques that are optimized for computing WCET, UPPAAL does not (right now) take full advantage of the particular nature of the WCET problem. A first option is to use the simulation based statistical model-checking approach which does not suffer from the state explosion problem. The drawback is that we only get an under-approximation of the WCET, but we get a distribution. A second option is to tune UPPAAL to get advantage of the particular nature of the problem at hand. Indeed, UPPAAL is not optimized for computing WCET. For instance, if two states of the product $Aut(H) \times Aut(P)$ differ only on the global time (time elapsed since the program started), the one with the smallest time stamp can be discarded and need not be explored further: the program and the hardware are deterministic, and thus the set of possible traces from the two states are equal. This can result in an exponential reduction of the explored state space as example of Fig. 4 shows. Assume that each instruction takes 1 cycle and the comparison instruction 4 is input data dependent: r_0 and r_1 are always unknown. Thus we have to consider both branches each time instruction 8 is encountered. What is remarkable is that whatever the taken branch is, the two states of the program and hardware when reaching instruction 16 are almost identical: the registers have the same values, the caches as well, only the value of the predicate eq set at instruction 4 is different. Nevertheless the value of this predicate does not influence the set of future traces and thus can be discarded. Clearly, the state with the lowest current time at instruction 16 should not be explored as the other one will for sure yield a larger execution time. This kind of pruning is not possible yet in UPPAAL. Note that partial-order reduction techniques [25] can also be effective.

X. Conclusion and Future Work

We have presented a framework based on program slicing and model-checking to compute WCET. We have compared the computed results with real execution times on the real hardware and showed we can achieve unmatched tightness.

Our method has several advantages: (1) it is based on a formal approach, based on efficient techniques; (2) it is fully modular and for instance, altering/using a new model of the hardware can be done easily by providing the timed automata models of the hardware; (3) it computes the CFG automatically. This avoids an error-prone, time-consuming and tedious step of the standard methods and tools that all require a manual intervention (loop bound annotations).

References

[1] R. Wilhelm *et al.*, "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." www.rapitasystems.com/.

[3] 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 Analysis,” in *WISES*, 2008, Regensburg, Germany.

[4] Tidorum Ltd., www.tidorum.fi/bound-t/.

[5] AbsInt Angewandte Informatik, www.absint.com/ait/.

[6] C. Ferdinand, R. Heckmann, and R. Wilhelm, “Analyzing the worst-case execution time by abstract interpretation of executable code,” in *ASWSD*, ser. LNCS, vol. 4147, 2004, pp. 1–14.

[7] T. Lundqvist and P. Stenstrom, “Timing Anomalies in Dynamically Scheduled Microprocessors,” in *IEEE Real-Time Systems Symposium*, 1999, pp. 12–21.

[8] F. Cassez, M. C. Olesen and R. R. Hansen, “What is a Timing Anomaly?,” in *WCET* Workshop, Pisa, Italy, 2012, pp. 1–12.

[9] R. von Hanxleden *et al.*, “Wcet tool challenge 2011: Report,” in *WCET* Workshop, Porto, Portugal, Jun. 2011.

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

[11] 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.

[12] A. Metzner, “Why Model Checking Can Improve WCET Analysis,” in *CAV*, ser. LNCS, vol. 3114, 2004, pp. 334–347.

[13] M. Lv, W. Yi, N. Guan, and G. Yu, “Combining Abstract Interpretation with Model Checking for Timing Analysis of Multicore Software,” in *IEEE Real-Time Systems Symposium*, 2010, pp. 339–349.

[14] A. E. Dalsgaard, M. C. Olesen, and M. Toft, “Modular execution time analysis using model checking,” Master’s thesis, Dpt. of Comp. Sc., Aalborg Univ., Denmark, 2009.

[15] 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* Workshop, Brussels, Belgium, 2010, pp. 113–123.

[16] F. Cassez, “Timed Games for Computing WCET for Pipelined Processors with Caches,” in *ACSD*, IEEE Comp. Soc., 2011, pp. 195–204.

[17] C. Sandberg, A. Ermedahl, J. Gustafsson, and B. Lisper, “Faster wcet flow analysis by program slicing,” in *LCTES*, ACM, 2006, pp. 103–112.

[18] J.-L. Béchennec and F. Cassez, “Computation of WCET using Program Slicing and Real-Time Model-Checking,” CNRS, Research Report, October 2011, 31 pages.

[19] 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.

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

[21] ARM Limited, *ARM9TDMI Technical Reference Manual*, 2000. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0180a/DDI0180.pdf

[22] T. Lengauer and R. E. Tarjan, “A fast algorithm for finding dominators in a flowgraph,” *ACM Trans. Program. Lang. Syst.*, vol. 1, no. 1, pp. 121–141, 1979.

[23] K. Cooper, T. Harvey, and K. Kennedy, “A Simple, Fast Dominance Algorithm,” *Software – Practice and Experience*, vol. 4, pp. 1–10, 2001.

[24] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, “The Mälardalen WCET benchmarks – past, present and future,” in *WCET* Workshop, Brussels, Belgium: 2010, pp. 137–147.

[25] F. Cassez, T. Chatain and C. Jard, “Symbolic unfoldings for networks of timed automata,” in *ATVA*, ser. LNCS, vol. 4218, 2006, pp. 307–321.

[26] A. David, K. G. Larsen, A. Legay, M. Mikucionis, and Z. Wang, “Time for statistical model checking of real-time systems,” in *CAV*, ser. LNCS, vol. 6806, 2011, pp. 349–355.

APPENDIX A. EXAMPLE PROGRAM *FIBO*$_0$

```
function <fib>:
0:   sub   sp, sp, #32         ⟦sp⟧ := ⟦sp⟧ − 32
4:   str   r0, [sp, #4]        ⟦⟦sp⟧ + 4⟧ := ⟦r0⟧
8:   mov   r3, #1              ⟦r3⟧ := 1
12:  str   r3, [sp, #16]       ⟦⟦sp⟧ + 16⟧ := ⟦r3⟧
16:  mov   r3, #0
20:  str   r3, [sp, #20]       ⟦⟦sp⟧ + 20⟧ := ⟦r3⟧
24:  mov   r3, #2
28:  str   r3, [sp, #12]
32:  b     0x50 [#80]          ⟦pc⟧ := 80
36:  ldr   r3, [sp, #16]       ⟦r3⟧ := ⟦⟦sp⟧ + 16⟧
40:  str   r3, [sp, #24]
44:  ldr   r2, [sp, #16]
48:  ldr   r3, [sp, #20]
52:  add   r3, r2, r3
56:  str   r3, [sp, #16]
60:  ldr   r3, [sp, #24]
64:  str   r3, [sp, #20]
68:  ldr   r3, [sp, #12]
72:  add   r3, r3, #1
76:  str   r3, [sp, #12]
80:  ldr   r2, [sp, #12]
84:  ldr   r3, [sp, #4]
88:  cmp   r2, r3              ⟦le⟧ := (⟦r2⟧ ≤ ⟦r3⟧)
92:  ble   0x24 [#36]          ⟦le⟧?(⟦pc⟧ := 36) : (⟦pc⟧ := 96)
96:  ldr   r3, [sp, #16]
100: str   r3, [sp, #28]
104: ldr   r3, [sp, #28]
108: mov   r0, r3
112: add   sp, sp, #32
116: bx    lr                  ⟦pc⟧ := ⟦lr⟧

function <main>:
120: push  {lr}                ⟦sp⟧ := ⟦sp⟧ − 4; ⟦⟦sp⟧⟧ := ⟦lr⟧
124: sub   sp, sp, #12
128: mov   r3, #300
132: str   r3, [sp, #4]
136: ldr   r0, [sp, #4]
140: bl    0  <fib>            ⟦lr⟧ := 144; ⟦pc⟧ := 0
144: mov   r3, r0
148: mov   r0, r3
152: add   sp, sp, #12
156: pop   {lr}                ⟦lr⟧ := ⟦⟦sp⟧⟧; ⟦sp⟧ := ⟦sp⟧ + 4
160: bx    lr
```

FIBO$_0$ computes the Fibonacci number u_{300} with $u_0 = 1$, $u_1 = 1$ and $u_n = u_{n-1} + u_{n-2}, n \geq 2$. The semantics of this program is given in terms of assignments to registers: $[\![x]\!]$ denotes the content of a register or memory cell. Each instruction assigns a new value to *pc*: except for branching instructions the assignment is $[\![pc]\!] := [\![pc]\!] + 4$ and we omit it. A comparison operator (e.g., instruction 88) sets the truth value of the predicates that are used later in the program (e.g., *le* for instruction 88 used in the branch instruction 92).

From state $[\![sp]\!] = 100$ executing the instruction $\iota = (20 : \texttt{str r}_3, [\text{sp}, \#20])$ generates $(20, \{120\}, \text{TRUE})$; from state $[\![le]\!] = \text{FALSE}, [\![pc]\!] = 92$, the instruction $(92 : \texttt{ble 0x24})$ generates $(92, \varnothing, \text{FALSE})$.