Computation of WCET using Program Slicing and Real-Time Model-Checking

Franck Cassez  
National ICT Australia, Sydney, Australia  
Jean-Luc Béchennec  
LUNAM University, CNRS, IRCCyN, Nantes, France

Abstract—We address the problem of computing accurate Worst-Case Execution Time (WCET) on pipelined architectures with caches. 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 overpowered and expensive hardware.

The WCET Problem. Given a binary program P, input data d, and hardware H, the execution time of P for input d on H, is measured as the number of cycles of the fastest component of the hardware i.e., the processor. The worst-case execution time (WCET) of program P on hardware H, WCET(P, H), is the supremum, over all input data d, of the execution time 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.

If the input data are known or the program execution time is independent from the input, there is a single path and it is feasible to compute the WCET. Likewise, if we can determine some input data that produces the WCET we can compute the WCET on a single-path program. Unfortunately, this is not often the case, the values of the input data are usually unknown and the number of execution paths might be extremely large.

Programs run on increasingly complex architectures featuring multi-stage pipelines and fast memory components like caches: they both influence the WCET in a complicated manner. It is then a challenging problem to determine a precise WCET even for relatively small programs running on complex 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: Testing-based methods are based on experiments i.e., running the program on some data, using a simulator of the hardware or the real platform. No upper bound can be obtained this way and thus this scheme is unsafe for safety critical embedded systems. RapiTime [2] and Mtime [3] implement this technique. Verification-based methods rely on the computation of an abstract graph, the control flow graph (CFG), and an abstract model of the hardware. Together with a static analysis tool they can be combined to compute the WCET. The CFG should produce a super-set 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 [4], OTAWA [5], TuBound [6], Chronos [7], SWEET [8] and aIT [9], [10] are static analysis-based tools for computing WCET.

The verification-based tools mentioned above have some known disadvantages: (1) they rely on the construction of a CFG, and the determination of loop bounds. Building the CFG is done using some internal knowledge of the compiler that generated the binary programs and thus handcrafted programs or binary programs generated by an unsupported compiler can not be analyzed. Loop bounds are often given via user annotations, which is tedious, time-consuming and more importantly error-prone. The CFG is then 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 specifications to compute the CFG fed to the ILP solver. (2) the correctness of the abstract models and the efficient procedures implemented in the tools strongly rely on the assumption that the hardware is timing anomaly free [11], [1]. The architecture of the tools themselves are monolithic: it is not easy to add support for a new hardware. The later weakness was witnessed by the WCET’08 Challenge Report [12] that highlights the difficulties encountered by the participants to adapt their tools for a new hardware. (3) the WCET computed by the tools are not compared against execution times measured on a real platform. Some tools report comparisons with ARMulator (the ARM simulator of the RealView Development Suite) but this simulator is not cycle accurate as emphasized in ARM documentation for ARMulator [13]:

ARMulator consists of C based models of ARM cores and as such cannot be guaranteed to completely reproduce the behaviour of the real hardware. If 100% accuracy is required, an HDL model should be used.

The above mentioned drawbacks (1), (2) and (3) are reported in the most recent WCET’11 Challenge Report [12]: all the tools require manual annotations, some of them could not support the target hardware and the computed estimates were not checked against real measurements.

Related Work. 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 data from the cache), formal models like timed automata (TA) [14] and state-of-the-art real-time model-checkers like UPPAAL [15], [16] appear well-suited to address the WCET problem. A. Metzner already showed [17] that model-checking could be used to compute the WCET for programs running on pipelined processors with an instruction cache. More recently, Lv et al. [18] combined AI techniques with real-time model-checking (using UPPAAL) to compute WCET on multi-core platforms.

In [19], B. Huber and M. Schoeberl consider Java programs and compare ILP-based and model-checking techniques using the model-checker UPPAAL. The recommendation is to use ILP tools for large programs and model-checking tools for code fragments. The use of network of timed automata (NTA) and UPPAAL for computing WCET on pipelined processors with caches was already reported in [20], [21] where the METAMOC method is described. METAMOC consists in: 1) computing the CFG of a program, 2) composing this CFG with a (network of timed automata) model of the processor and the caches. Computing the WCET is then reduced to computing the longest path (time wise) in a NTA.

The previous framework is very elegant yet has some shortcomings: (1) METAMOC relies on a value analysis phase (to compute a CFG) that may not terminate, (2) some programs cannot be analyzed (if they contain register-indirect jumps), (3) manual annotations (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 [22] 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 is done in a totally different manner.

Program slicing to compute WCET was considered in [23]: in this work, slicing is performed on structured programs in the intermediate NIC format and the authors assume that the CFG of the program is available. In this respect this is different to what we propose as we are slicing binary programs and computing the CFG.

Our Contribution. Compared to [22], we propose three new original contributions: (1) an 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 as a composition of two automata: the hardware and the program to be analyzed; (3) the design of accurate hardware formal models (Section VII); and (4) a comparison of computed WCET with measured WCET on a real hardware. Noteworthy, our method is robust against timing anomalies.

Outline of the Paper. In Section II we give the specification of the hardware we use in the experiments. Section III gives some formal definitions for program execution on a given hardware. Section IV presents a modular way to compute the WCET of a given program. Section V shows how to compute a reduced WCET-equivalent program and Section VI describes how to automatically build the CFG. Section IX gives a summary of the results on benchmark programs (Table I) and also reports on the implementation and methodology we use to compare our computed WCET with measured WCET. A earlier version of this paper is available at http://www.irccyn.fr/franck/wcet as report [24] together with the detailed methodology and the UPPAAL models.

II. ARCHITECTURE OF THE ARM920T

The board we model in Section VII is an Armadadeus APF9328 board [25] which bears a 200MHz Freescale MC9328MXL micro-controller with an ARM920T processor which embeds an ARM9TDMI core that implements the ARM v4T architecture.

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. An instruction is defined by a mnemonic (e.g., \texttt{mov}) and the operands. In the sequel, we let \( \mathcal{R} = \{r_0, \ldots, r_{12}, sp, lr, pc\} \) be the set of registers of the architecture and \( \mathcal{I} \) be the (finite) set of RISC instructions.

The ARM920T uses a 5-stage execution pipeline (Fig. 1), the purpose of which is to execute concurrently the different stages (Fetch, Decode, Execute, Memory, Writeback) needed to perform an instruction. An instruction is fetched in F, decoding and operand register accesses occur in D, execution in E and load/store instructions do their memory accesses in M. The results are written back to registers in W. The (normal) flow of instructions in the pipeline is shown in Fig. 1. This optimal flow may be slowed down when pipeline stalls occur (e.g., due to register dependencies).

Remark 1: In order to determine pipeline stalls, it is enough 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). Both caches implement allocate-on-read-miss (data is inserted in the cache if missing when a read is performed). A 16-word write buffer helps to reduce stalls when a write to the main memory occurs because of a write miss or, if the cache is configured in write-back, when a dirty line has to be replaced. The write buffer is organized in 4 half-line entries to allow cache write-back on a half-line basis. Transfers between the caches and main memory are serialized.

Remark 2: To determine cache misses/hits it is enough to know (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 \( P \) we analyze: (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 do 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 \( \mathcal{B} = \{\textsc{true}, \textsc{false}\} \), \( \mathcal{LI} = \mathcal{M} \times \mathcal{I} \) and \( \mathcal{D} \) be a finite set of values that the memory cells and registers can hold e.g., 32-bit integers. Recall that \( \mathcal{R} \) is the set of registers, \( \mathcal{I} \) is the (finite) set of instructions the hardware can perform. \( \mathcal{M} \) is the (finite) set of main memory cells. In the sequel we use a set of predicates \( \mathcal{P} \) and for \( x \in \mathcal{R} \cup \mathcal{P} \cup \mathcal{M} \), \( [x] \) denotes the content of \( x \). A program state \( s \) is a mapping from \( \mathcal{R} \cup \mathcal{P} \cup \mathcal{M} \) to \( \mathcal{D} \). \( \mathcal{S} \) is the set of program states.

A program instruction is located in main memory and defined by a pair \((\ell : i) \in \mathcal{LI}\). A program \( P \) is simply a subset (necessarily finite) of \( \mathcal{LI} \). We use the notation \( <\ell>: s \rightarrow \mathcal{S} \) to denote the semantics of instruction \( i \in \mathcal{LI} \). Each program has a designated initial instruction \( i_0 = (\ell_0, i_0) \) and \( [pc] = \ell_0 \) in the initial program state.

```
function <fib>:
0: sub sp, sp, #32 \quad [sp] := [sp] - 32
4: str r0, [sp, #4] \quad [[sp] + 4] := [r0]
8: mov r3, #1 \quad \langle r3 \rangle \leftarrow 1
12: str r3, [sp, #16] \quad \langle [sp] + 16 \rangle := [r3]
16: mov r0, #0
20: str r3, [sp, #20] \quad \langle [sp] + 20 \rangle := [r3]
24: mov r3, #2
28: str r3, [sp, #12]
32: b 0x50 \#80
36: ldr r3, [sp, #16] \quad \langle r3 \rangle := \langle [sp] + 16 \rangle
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: ldr r3, [sp, #12]
80: ldr r2, [sp, #12]
84: ldr r3, [sp, #4]
88: cmp r2, r3
92: ble 0x24 \#36
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 \quad [pc] := [lr]
```

Program 1. \textit{FIBO}_0

As a running example we take the binary\(^2\) program \textit{FIBO}_0 (see Program 1). It has been compiled (\texttt{gcc}) and de-
assembled (objdump) using the GNU ARM tools from Codesourcery [26]. It computes the Fibonacci number \( u_{300} \) with \( u_0 = 1 \), \( u_1 = 1 \) and \( u_n = u_{n-1} + u_{n-2}, n \geq 2 \). It is composed of two parts: the main function which is entry point (starts at instruction 120) which then calls the \( fib \) function (at instruction 140).

We let \( P = \{ le, gt, \ldots \} \) be a finite set of predicates to hold the truth values of the conditions used in the conditional instructions of the program.\(^3\)

The semantics of program \( FIB0h \) is given in terms of assignments to registers (Program 1, right): \( [x] \) 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).

The real hardware (Section II) consists of the pipelined processor, instruction and data caches, write buffer and the main memory of the computer. The AMBA bus and MMU model remains simple and takes only the memory access latency and the bus occupation into account.

We choose to treat the content of the main memory and the registers to be part of the program state. A state \( \gamma \) of the hardware is then solely defined by the states of the different stages of the pipeline and the cache is fully determined by the states of the different stages of the pipeline and the cache.

To compute the execution times of program runs, we view the hardware as an abstract machine that reads sequences of triples \( (i, A, d) \in \mathcal{L} \times \mathcal{M} \times \mathbb{B} \) generated by \( P \). A program state \( \ell \) acts as a transducer and outputs the time (in cycles) it takes to process such a sequence. A triple \( (i, A, d) \) consists of an (labelled) instruction \( i = (E:i) \), the set \( A \) of memory addresses it references and a boolean \( d \) that indicates whether it is performed or not.\(^4\)

For example, from a state with \( [sp] = 100 \) executing the instruction \( i = (20 : str r_3, [sp, \#20]) \) will generate the triple \( (20, (120), \text{TRUE}) \). In a similar manner, from the state \( [le] = \text{FALSE} \) the instruction \( (92 : \text{ble} \ 0x24) \) generates the triple \( (92, \varnothing, \text{FALSE}) \). We use these triples (and sequences thereof) because they contain the relevant information to compute the execution time\(^5\) of a run: (i) pipeline stalls are fully determined by (a) the first component of the triple \( i \) that contains the complete description of the instruction and the read/written registers and the (immediate) constants and (b) the value of \( d \) (Cf. Remark 1); (ii) cache hits/misses are fully determined by the set \( A \) (Cf. Remark 2). Notice that there is no need for actual register values in \( H \) neither for performing the real computation as the timing of instructions in the pipeline and the cache is fully determined by the sequences of triple \( (i, A, d) \).\(^6\)

Given a sequence of triples \( w = q_0q_1q_2 \cdots q_n \in (\mathcal{L} \times \mathcal{M} \times \mathbb{B})^* \) and an state \( \gamma \) of \( H \), \( \text{time}_H(\gamma, w) \) is the execution time of \( w \) from \( \gamma \). It can be precisely defined using for instance the HDL model of the hardware or formal models (NTA, Section IX). At this stage, we do not require sequences of triples to be actual sequences produced by program \( P \).

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 \( D \). 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 \( P \)). A run of \( P \) is an alternating sequence of program states and instructions \( \varrho = s_0 \ i_0 \ s_1 \ i_1 \ s_2 \ i_2 \ \cdots \ s_{n-1} \ i_{n-1} \ s_n \) where \( s_0 \in I, s_k \) is a program state, \( i_k = (E:k) \in \mathcal{L} \), \( \ell_k = s_k(pc) \) and \( s_{k+1} =< i_{k+1} > (s_k) \). We let \( Runs(P) \) be the set of runs of \( P \). The \( trace, TR(\varrho) \), of the run \( \varrho \) is the sequence \( q_0q_1q_2 \cdots q_n \in (\mathcal{L} \times \mathcal{M} \times \mathbb{B})^* \) with \( q_i = (i_i, A_i, d_i) \) where \( A_i \) is the set of memory addresses referenced by instruction \( i_i \) in state \( s_i \) and \( d_i \in \mathbb{B} \) indicates whether the instruction is actually executed.\(^7\) The execution time of a run \( \varrho \) of \( P \) from initial state \( \gamma \) of \( H \) is given by \( \text{time}_H(\gamma, TR(\varrho)) \). The language \( \mathcal{L}_F^P \) of \( P \) is the set of sequences of triples (traces) generated by runs of \( P \) that start in \( I \) and end in \( F \) i.e., \( \mathcal{L}_F^P = \{ TR(\varrho) \mid \varrho = s_0q_0q_1q_2q_3 \cdots q_n, \varrho \in Runs(P), s_1 \in I, s_n \in F \} \). As we assume that \( P \) always terminates for any input data, this language is finite.

IV. COMPUTATION OF THE WCET

Modular Definition of WCET. Given a run \( \varrho \) of \( P \), the execution time of \( \varrho \) on \( H \) from state \( \gamma \) only depends on \( TR(\varrho) \). This means that WCET\( (P,H) \) is completely determined by \( \mathcal{L}_F^P \) and an initial state \( \gamma \) of \( H \)

\[
\text{WCET}(P,H) = \max_{w \in \mathcal{L}_F^P} \text{time}_H(\gamma, w). \tag{1}
\]

Computing WCET\( (P,H) \) thus amounts to: (i) generating \( \mathcal{L}_F^P \), (ii) feeding \( H \) with each \( w \in \mathcal{L}_F^P \) and tracking the maximal execution time. This gives a modular way of computing WCET\( (P,H) \) since a generator for \( \mathcal{L}_F^P \) and a model of the abstract hardware \( H \) (to be fed with \( \mathcal{L}_F^P \)) can be given independently of each other.

Remark 3: Notice that the initial state of \( H \) may not be unique: for instance the caches might be empty or all the

---

\(^3\)In the ARM 32-bit instruction set, the truth values are stored in the status bits \( N, Z, C, V \).

\(^4\)As said previously, instructions can be conditional and depend on the outcome of a previous comparison.

\(^5\)The fact that there is no branch prediction in the pipeline of the hardware in the ARM920T makes things simpler but the framework we present extends to the case with branch prediction (see [22]).

\(^6\)Some instructions (MUL/MLA/SMLUL) have data dependent durations. In this case, in the TA model of the execute stage of the pipeline, the duration is constrained to be in an interval.

\(^7\)\( S_i \) and \( d_i \) can always be computed from \( s_i \) and \( \epsilon_i \).
Extended Domain Abstraction. In order to take into account all the possible values of the input data, we use an extended domain for the values of the main memory cells. Let \( D_{\perp} = D \cup \{ \perp \} \) be the extended domain with \( \perp \) the unknown value. The semantics of instructions is extended in a straightforward manner to this extended domain: for instance, the semantics of add \( r_0, r_1, \#1 \) is given by \( [r_0] = \perp \) if \( ([r_1] = \perp) \) and \([r_1] + 1 \) otherwise. The semantics of comparison instructions e.g., cmp \( r_0, r_1 \) is extended as well to \( D_{\perp} \) e.g., for instruction at line 88 of program \( FIBO_0 \) \( [r_3] = \perp \) if \( ([r_2] = \perp) \) or \( ([r_3] = \perp) \) and \( ([r_2] \leq [r_3]) \) otherwise. When a conditional instruction is encountered and the condition is \( \perp \), the extended semantics considers two successors: one where the condition is TRUE and the other where the condition is FALSE. If a branching instruction like \( bx \ lr \) is encountered and \([r] = \perp \) the next instruction is undefined.\(^8\)

We may now define an extended symbolic semantics for a program \( P \). From an initial state \( s_0 : R \cup P \cup M \rightarrow D_{\perp} \), the symbolic semantics define a set of runs (non-determinism may arise if some conditions are tested and unknown.) Assume that the (fixed) values of the registers and predicates are given by \( s_0((R \cup P) \) and the input data is \( d \): the initial state of the memory is \( s_0(d) : M \rightarrow D \). The initial state of the program is thus defined by \( s_0(R \cup P) \) and \( s_0(d) \). Let \( s_0^1 : R \cup P \cup M \rightarrow D_{\perp} \) with \( s_0^1(x) = s_0(x) \) for \( x \in R \cup P \) and \( s_0^1(y) = \perp \) for \( y \in M \). The important property of the extended semantics is (\( \pi \)): if \( \varrho \) is a run of \( P \) from state \( (s_0(R \cup P), s_0(d)) \), then \( \varrho \) is a run of \( P \) from \( s_0^1 \) in the extended symbolic semantics. Let \( L_{\perp}(P) = \{ L_{\perp}^{\varrho_1}(P) \} \) be the set of sequences of triples (traces) when \( P \) is run using the extended domain. As the symbolic semantics over the extended 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 \( P \): \( L_{\perp}^{\varrho_1}(P) \subseteq L_{\perp}(P) \) and hence WCET\((P,H) \leq WCET_{\perp}(P,H) \)

\[
\begin{align*}
\max_{w \in L_{\perp}(P)} \text{time}_H(\gamma, w) = WCET_{\perp}(P,H).
\end{align*}
\]

We can thus reduce the computation of (an upper bound of the) WCET\((P,H) \) to a symbolic program \( P \) on the extended domain \( D_{\perp} \) from initial state \( s_0^1 \). All we require is that \( P \) also terminates over the extended domain.

\textbf{WCET Computation.} We can reduce the computation of the WCET to a reachability problem on a NTA. First, we build an automaton \( \text{Aut}(P) \) that generates \( L_{\perp}(P) \): the initial states of the automaton are given by \( I \) and the successor states by the symbolic 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.\(^9\) The hardware \( H \) is specified by a NTA \( \text{Aut}(H) \). Feeding \( H \) with \( L_{\perp}(P) \) amounts to building the synchronized product \( \text{Aut}(H) \times \text{Aut}(P) \). On this product we define \textit{final states} to be 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).

This can be done with \textsc{Uppaal} \cite{Uppaal} using a global clock and the \textit{sup} operator that gives the maximal value a clock can have in a given set of states. Assume a fresh\(^{10}\) clock \( x \) is reset in the initial state of \( \text{Aut}(H) \times \text{Aut}(P) \). The WCET of \( P \) on \( H \) is then the largest value \( x \) can take in a final state of \( \text{Aut}(H) \times \text{Aut}(P) \) (we make sure that time cannot progress from a final state).

We compute \( \max(x) \) using model-checking techniques and the \textit{sup} operator of \textsc{Uppaal} that gives the maximal value a clock can have in a given set of states.

Notice that to do this we have to explore the whole state space of \( \text{Aut}(H) \times \text{Aut}(P) \). To handle large case studies, we need to reduce the state space as much as possible. An important point to notice is that the tightness of the WCET we compute depends on how accurate is the model of \( H \). The more precise (time-wise) \( \text{Aut}(H) \) is, the more precise the computed WCET will be. It is thus not reasonable to take a very abstract \( H \) (e.g., with caches that always miss) as it will give poor WCET estimates. We can still have some control on the automaton \( \text{Aut}(P) \) that generates the traces to be fed to \( \text{Aut}(H) \). 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 \( \text{Aut}(P) \) while still generating the same sequences of triples we reduce the number of states of \( \text{Aut}(P) \) as well as the number of explored paths in \( \text{Aut}(H) \times \text{Aut}(P) \). In the next section we describe how to compute a reduced program \( P' \) that generates the same language as \( P \).

V. \textsc{Computing} of a WCET-Equivalent Program Using Slicing

In this section we show how to use program slicing to compute a WCET\textit{-equivalent} program. Equation (2) implies that for two programs \( P \) and \( P' \), if \( L_{\perp}(P) = L_{\perp}(P') \) then WCET\(_{\perp}(P,H) = WCET\(_{\perp}(P',H) \). Our goal is thus to compute such a WCET-equivalent program \( P' \) which (hopefully) has less states than \( P \) yet contains enough information to generate \( L_{\perp}(P) \). We can define \( P' \) as a slice

\(^8\)It can be encoded by a jump to an “error” state but assumption A1 ensures it never happens.

\(^9\)Some instructions (MUL/MLA/S Mull) have data dependent durations. In this case, in the TA model of the execute stage of the pipeline, the duration is in an interval.

\(^{10}\)\(x\) is not a clock of \( \text{Aut}(H) \).
of $P$. A slice of $P$ is a subset of instructions of $P$ together with a subset of the variables used in $P$.

The purpose of program slicing (Weiser [27]) 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 [27] 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. Program slicing is often used for checking properties of programs but we use it here to compute a reduced WCET-equivalent program. The reader is referred to [28] for a survey on the principles of (static and dynamic) slicing and Appendix A recalls the basics of program slicing.

The slice criterion we start with contains (i) the instructions that make memory accesses and (ii) conditional instructions: this ensures that the sliced program will generate the same triples $\{i, A, d\}$ for all these instructions. For example the slice criterion for the program FIBO contains the following pairs: $(12, \{sp\})$, $(92, \{le\})$. 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 not generate the same sequences of triples. It is defined as 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 semantics applies; if it is not in the slice, its semantics is $[pc] := [pc] + 4$, i.e., it is interpreted as a nop instruction. The instructions of $P$ are not discarded but their semantics interpreted as nop to preserve the generated triples. The correctness of this approach i.e., $L (P') = L (P)$, is ensured by the property of a slice (and the choice of the slice criterion.)

Slicing for binary program is not easy because (i) a binary program is not structured and (ii) the names of the variables used in the program have disappeared and are references to memory cells computed dynamically.

We first assume that the CFG of $P$ is known. The computation of a slice is based on an iterative solution of data-flow equations on the set of relevant variables for each instruction in the CFG of $P$. The relevant variables for an instruction are the variables read from/written to by the instruction. 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 $[[sp] + 4] := [r0]$ The value of $sp$ is only known at runtime. Other instructions like $str \ r2, [r1, r3 \ lsl \ #2]$ might (read or) write to arbitrary memory cells $[[r1] + ([r3] \times 2)] := [r2]$. To overcome this difficulty, we start by defining the sets of REF (read) and DEF (written) variables for each instruction (which is mandatory in order to automatically compute a slice) as follows;

- for instructions that do not make main memory references or stack references, we know the exact set of DEF and REF variables e.g., for instruction $i = add \ r2, r1, #1$ $REF(i) = \{r1\}$ and $DEF(i) = \{r2\}$.
- for instructions that make stack references, e.g., $i = push(\ r1)$, we define $REF(i) = \{r, sp\}$ and $DEF(i) = \{sp, stack\}$; all we know at that stage is that the stack is modified without any knowledge of the precise index of the item.
- 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\textsuperscript{11}. We use a definition similar to the one for the stack: For an instruction like $i = str \ r2, [r1, r3 \ lsl \ #2]$ we thus have $REF(i) = \{r1, r2, r3\}$ and $DEF(i) = \emptyset$ as we assume that the content of the main memory is always unknown.

In a first phase we define a slice criterion $SC_0$ that consists of every instruction that reads or writes the stack pointer (for FIBO, $SC_0$ contains $\{0, \{sp\}\}, \{4, \{sp\}\}, \ldots, \{120, \{sp\}\}$). We compute the corresponding slice and simulate it (initial values of the stack pointer must be given) to gather the possible values of $sp$ at the points defined by $SC_0$. This means that for each instruction in $SC_0$, we collect the set of possible stack pointer values. This enables us to define more precisely the set of DEF and REF variables for each instruction that makes a stack access in the program: we can now replace the variable stack by actual stack addresses that are referenced. For instance, if the collected values of register $sp$ for instruction $i = 4 : str \ r0, [sp, #4]$ are $\{60, 64\}$, we set $REF(i) = \{sp, r0\}$ and $DEF(i) = \{stack64, stack68\}$.

Given the precise knowledge of the DEF and REF variables for each instruction, we can compute a WCET-equivalent program in a second slice. The slice criterion $SC_1$ contains (as said at the beginning of this section) the instructions that (i) make memory accesses and (ii) are conditional. This ensures that the sliced program generate the same triples $\{i, A, d\}$ for all these instructions.

Slices are computed using the standard definition of data dependence and control dependence (see [28]).

An example of a slice for the Fibonacci program FIBO that computes $u_{300}$ is given in Fig. 2: only 13 instructions (highlighted in red) out of 41 need be really simulated and the needed variables in the sliced program are $\{pc, r0, r2, r3\}$ and 3 stack values.

Slicing enables us to avoid loop counter annotations. Indeed, as witnessed by the previous example in Fig. 2, all the instructions that control the loops 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

\textsuperscript{11}In reality we can differentiate memory zones and process them in a different manner
and updates are captured by the slice.

For instance, the initial value of the would-be “loop-counter” is stored in \( [sp] + 12 \) (at instructions 24 and 28), the bound 300 stored at \( [sp] + 4 \) (instructions 128 and 132). When the loop is executed these values are used and updated (instruction 72 increments \( r_3 \) which is later stored in \( [sp] + 12 \) at 76). Thus we are not inferring loop bounds nor need loop bounds annotations as it is implicitly captured by the slice.

A 2-stage Slicing Algorithm. The reason is efficiency and accuracy. If we would compute the WCET-equivalent program using the \( \text{stack} \) variable only, we could get 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: an instruction that reads stack variable \( \text{stack}_1 \) will not depend on an instruction that writes \( \text{stack}_1 \). Moreover, the slices computed for the first phase are small 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 13, column “Abs” \( (\alpha/\beta) \) gives, for each program \( P \), the number of nodes \( a \) that are to be simulated in the sliced program \( P' \) compared to the total number of nodes in the CFG of \( P \). The number of slice variables is also drastically smaller in the WCET-equivalent slice.

Remark 4: The WCET-equivalent program we compute does not suffer from the so-called domino effect that appears only when performing loop unrolling. More generally, our method is robust against timing anomalies as we exercise all the possible paths and we do not use abstract caches.

Correctness and Termination. The correctness of the algorithm relies on the definition of the slice criterion: as it comprises of the conditional instructions and the instructions that do memory transfers, we guarantee that the slice preserves the triples generated by \( L_\perp(P) \). Termination is not guaranteed as we symbolically simulate the programs in the extended domain. Nonetheless, for all the programs we processed so far (Table I), the symbolic simulation terminated.

VI. Computation of the CFG

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. This gives a (partial) CFG \( C \) where some dynamically computed branching 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 \( \text{succ}(R) \) for \( R \). Add edges from the nodes in \( R \) to their computed successors \( \text{succ}(R) \). Set \( S \) to the set of nodes in \( \text{succ}(R) \) 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 previous algorithm on the example of 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. 3), 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 \( EXIT_{\text{top}} \)) 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. 3) 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 \( \text{succ}(R) = \{144\} \) is created and an edge from 116 to 144.
Remark 5: Notice that when we simulate the slices, we set the initial value of $sp$ and $pc$ to known values and set the values of the other registers to $\bot$.

In the next iteration of the 3steps, we start with $S = \{144\}$. We extend this CFG (Step 1) from 144 to obtain a CFG (Fig. 4) comprising of all the instructions up to instruction $R = \{160\}$ which is a dynamically computed branching again.

This second CFG is sliced (Step 2) to compute the successor of instruction 160: the new slice (red nodes in Fig. 4) 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 there 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 with is a program final node and thus the algorithm terminates. The final CFG is given in Fig. 2. To recognize this situation we use the following trick: we assume that before the first instruction of the program is performed, $[\text{lr}] = \beta$ where $\beta$ is a special value that cannot correspond to any valid instruction. We can take for example $\beta = 3$. When we compute a target which is $\beta$ we know that we have reached the end of the program because this returns to the initial caller. This situation occurs when we simulate the second slice and after the instruction 160:$\text{bx lr}$ the program returns to the initial caller.

The computation of the possible values of stack/memory references 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/memory references at each node.

Remark 6: If the symbolic simulation generates an unknown target value $\bot$, 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 a known target value $\bot$. 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.

Remark 7: The case of switch statements is different: there are several known (at compile time) targets and the CFG is built by having as many successors as switches.
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 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 [29], 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 of the hardware, 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.

These programs exhibit only one path and the execution time does not depend on any input data. Consequently their execution time is equal to their WCET. The basis of this identification phase consists in measuring the difference in execution times of two variants of a loop. The second variant contains a sequence of instructions for which we want a precise timing. The execution time difference between the two variants is the execution time of this sequence multiplied by the number of iterations.

These program are run on the real hardware to measure their execution times and on the formal model to compute their execution times. As each program stresses a particular feature of the hardware, a difference between the measured execution time and the computed one indicates that our formal model does not model correctly the feature. Running a large set of special-purpose programs, we were able to refine the formal model of the pipeline, caches and write buffer. This identification method gives formal models having a timing behaviour very close to the actual hardware timing behaviour and contributes to the remarkably accurate results we obtain (see Table I).

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.

Remark 8: Compared to other tools, the hardware models we use are freely available as UPPAAL timed automata. It is thus possible to check their accuracy and level of details. METAMOC models are also freely available: they are simple generic pipelines and caches for the ARM920T and our formal models are a lot more detailed and model features of the ARM920T hardware like the write buffer, or the alignment of addresses in the caches.

VIII. UPPAAAL AUTOMATA

In this section we present some features of the formal models (timed automata) of the hardware. The automata are given using the UPPAAL syntax: initial locations are identified by double circles, guards are green, synchronisation signals (channels) are light blue and assignments are dark blue. A C in a location means committed: when an automaton enters a committed location, it cannot be interrupted and proceeds immediately to one of the successors of this location (the guards determine the transitions that can be taken). The UPPAAL models are available from http://www.irccyn.fr/franck/wcet.

A. Main Memory

![Figure 5. Main Memory TA](image)

The main memory model is a very simple two-location automaton (Fig. 5). When a memory transfer is required, signal MainMemStart? is received and clock t is reset. After a delay of MAINMEMTRANS the transfer is completed and signal MainMemEnd! is issued. Main memory transfers are triggered by either the instruction or data cache and accesses to main memory is serialised.

B. Caches

The model of the instruction cache is given in Fig. 6. The state of the cache contains an array (64 x 8 array) to record the addresses stored in the cache and whether a line is dirty or not.

The instruction cache is simpler than the data cache because no write can occur in this cache, so a line cannot be dirty. After the initialisation of the cache (initial state of the cache by the function initCache()), the automaton is ready for receiving the signal CacheReadStart[num]?. This signal will be triggered by the fetch stage of the pipeline Fig. 7. The memory address to read is m. If m is in the cache (function is_in(m) returns TRUE), there is no need for a memory transfer and variable PMT (Pending Memory Transfers) is assigned 0. Otherwise function insert(m) inserts m in the cache and returns the number of memory transfers to be performed: for the instruction cache it is always 1 because a line cannot be dirty (see Section II) but for the data cache it can be either one or 2 if a dirty
line has to be saved from the cache. As soon as the memory transfer is completed (PMT=0) transition Hurry! is fired (it is urgent). Then, after CACHE_SPEED time units (value is 1 for the our testbed) the read request completes and the signal CacheReadEnd[num]! is issued.

A special signal prog_completed? is issued from the program and marks the last instruction of the program. The program is completed when this last instruction flows out of the last stage (W) of the pipeline and this corresponds to reaching location DONE of the W stage. The automaton for the M stage is given in Fig. 8: when an instruction is performed and it is a memory transaction, it issues a sequence of read/write requests to the data cache.

![Figure 6. Instruction Cache](image)

The data cache is a bit more involved (Fig. 8). For a read/hit operation it behaves almost like the instruction cache described above. For write operations, a write buffer (not given here) is used and moreover the timing depends on the type (load/store), addresses involved in the operation, and whether another write/read operation is already in progress (and to which line in the write buffer). We have tried to design an accurate model of the data cache: data cache operations are the major factor in the WCET for most of the programs and a faithful model is required to compute tight bounds.

### C. Pipeline Model

The model of the pipeline is rather simple except the memory stage (M) which is a bit more complicated. The F stage automaton fetches the next instruction if no branch delay stall occurs. The function stall() of the F stage automaton determines whether such a stall should occur or not. If the next instruction can be fetched, it is fetched from the instruction cache CacheReadStart[num]! (this signal is urgent and synchronised with the instruction cache). When the fetch is completed the instruction is transferred to the next D stage, as soon as it is ready to be fed with a new instruction. The D stage, E stage and W stage are similar. Notice that the duration of an instruction may vary from one instruction to the other (e.g., long multiplication may take longer than additions) or because a conditional instruction is not executed: the actual duration is set when a new instruction arrives in the E stage (DUR_INSTR=dur()).

IX. Implementation & Experiments

**Implementation.** The binary programs are computed from C/C++ programs with the GCC tool suite for ARM (gcc, objdump) from Codesourcery [26].

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 3000 thousand C++ lines of code. We have implemented very efficient versions of post-dominators algorithms [30], [31] and post dominance frontiers algorithms [32] 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 format (see www.irccyn.fr/franck/wcet for the files containing the full description of the programs of Table I).

**Methodology.** The precise methodology to measure the execution time on our testbed is available 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 produce the WCET; in this sense we obtain a measured lower bound of the WCET.

**Experiments on Benchmark Programs.** The results we have obtained on some benchmark programs<sup>12</sup> from Mälardalen University [33] are reported 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.

<sup>12</sup>http://www.mrtc.mdh.se/projects/wcet/benchmarks.html
• we have increased the number of iterations of the benchmarks (e.g., we compute the WCET of Fib(300) instead of Fib(30))\(^1\); this way a modelling error (e.g., \(t = 1\) cycle per loop) is revealed and will incur a huge overapproximation in the computed value.

The programs we have experimented are not toy examples. 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 counts that depend on the outer loop counter, the maximum error is 2.4\%. This validates the accuracy of the program 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 some branching that are input data dependent. The measured WCET is the execution time (on the hardware) obtained with input data that should produce the WCET. The computed WCET considers all the possible input data. For bs-O0, O1, O2 the WCET is very small and measurement errors are more than 1\%.

**Remark 9:** According to the results, there does not seem to be any relationship between the type of a program (data-intensive or control-intensive) and the sizes of the slices.

An important question is the practical effect of slicing and the use of the WCET-equivalent program. What happened when no slicing is used is given by the METAMOC results: they show (http://metamoc.dk/, benchmark results), that even with abstract caches, and a large amount of memory (32GB) for UPAAAL, 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.

**X. DISCUSSION**

In this section we discuss the strengths and current 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. 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 yes generate corresponding initial input data; in case the witness trace is not feasible, counter example guided

---

\(^1\) The computation of Fib(300) results in an arithmetic overflow but we can still compute the time it takes to compute the result.
Figure 8. Memory Stage and Data Cache
refinement can be carried out to refine the model (these lines of research are currently being investigated).

The use of TA models for the hardware is also a clear advantage as the specifications are readable, formal, compact, amenable to changes (e.g., LRU vs FIFO replacement, always-miss caches) and can be simulated (using UPPAAL).

Modularity is a nice feature as well: improvements can be made on the slicing algorithms regardless of the target hardware. 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 is modeled using an extra TA that specifies how the processor speed changes.

The current limitation of the approach is the model-checking phase (with UPPAAL) but there is room for improvements in this area. Indeed, UPPAAL is not optimized for computing WCET and the special nature of this problem can be used to prune the search space. For instance, if two states of the product Aut(H) × Aut(P) differ only on the global time (time elapsed since the program started), the one with the smallest timestamp 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. 9 shows.

Assume that each instruction takes 1 cycle. The first comparison (instruction 4) is input data dependent (r0 and r1 are always unknown). Thus we have to consider both branches each time instruction 8 is encountered. What is remarkable is that whatever the branch taken, 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

<table>
<thead>
<tr>
<th>Program</th>
<th>loc</th>
<th>UPPAAL</th>
<th>Computed WCET (C)</th>
<th>Measured WCET (M)</th>
<th>Error (%)</th>
<th>Abs</th>
</tr>
</thead>
<tbody>
<tr>
<td>lb-O0</td>
<td>74</td>
<td>2.32/74181</td>
<td>8096</td>
<td>8064</td>
<td>0.42%</td>
<td>47/131</td>
</tr>
<tr>
<td>lb-O1</td>
<td>74</td>
<td>0.76/2333</td>
<td>2597</td>
<td>2544</td>
<td>2.0%</td>
<td>18/2</td>
</tr>
<tr>
<td>lb-O2</td>
<td>74</td>
<td>0.43/9711</td>
<td>1209</td>
<td>1164</td>
<td>3.3%</td>
<td>22/7</td>
</tr>
<tr>
<td>janne-complex-O0</td>
<td>65</td>
<td>2.05/38014</td>
<td>4264</td>
<td>4164</td>
<td>2.4%</td>
<td>78/173</td>
</tr>
<tr>
<td>janne-complex-O1</td>
<td>65</td>
<td>0.76/14600</td>
<td>1715</td>
<td>1680</td>
<td>2.0%</td>
<td>38/89</td>
</tr>
<tr>
<td>janne-complex-O2</td>
<td>65</td>
<td>13.59/11903</td>
<td>1557</td>
<td>1536</td>
<td>1.3%</td>
<td>32/78</td>
</tr>
<tr>
<td>fdct-O1</td>
<td>238</td>
<td>25.48/60418</td>
<td>4245</td>
<td>4092</td>
<td>3.7%</td>
<td>100/263</td>
</tr>
<tr>
<td>fdct-O2</td>
<td>238</td>
<td>1.34/35285</td>
<td>19231</td>
<td>18984</td>
<td>1.3%</td>
<td>166/5534</td>
</tr>
<tr>
<td>matmult-O0</td>
<td>162</td>
<td>5np/95/1031230</td>
<td>[502850,529250]</td>
<td>[511584,528684]</td>
<td>0.1%</td>
<td>158/314</td>
</tr>
<tr>
<td>matmult-O1</td>
<td>162</td>
<td>32.58/1112527</td>
<td>[130001,156367]</td>
<td>[127356,135000]</td>
<td>2.2%</td>
<td>71/172</td>
</tr>
<tr>
<td>matmult-O2</td>
<td>162</td>
<td>31.23/1015180</td>
<td>[122048,149259]</td>
<td>[110484,140664]</td>
<td>5.4%</td>
<td>78/288</td>
</tr>
<tr>
<td>jfjcirc-O0</td>
<td>374</td>
<td>109.38/100861</td>
<td>[12669,12918]</td>
<td>12588</td>
<td>2.6%</td>
<td>159/792</td>
</tr>
<tr>
<td>jfjcenter-O1</td>
<td>374</td>
<td>14.82/35419</td>
<td>[14979,5072]</td>
<td>4668</td>
<td>8.6%</td>
<td>253/225</td>
</tr>
<tr>
<td>jfjcenter-O2</td>
<td>374</td>
<td>5.38/217561</td>
<td>[16746,16938]</td>
<td>16380</td>
<td>3.0%</td>
<td>54/2512</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program</th>
<th>loc</th>
<th>UPPAAL</th>
<th>Computed WCET (C)</th>
<th>Measured WCET (M)</th>
<th>Error (%)</th>
<th>Abs</th>
</tr>
</thead>
<tbody>
<tr>
<td>iso-O0</td>
<td>174</td>
<td>37.59/142124</td>
<td>9088</td>
<td>1056</td>
<td>1.1%</td>
<td>75/151</td>
</tr>
<tr>
<td>iso-O1</td>
<td>174</td>
<td>29.61/1214053</td>
<td>758</td>
<td>720</td>
<td>2.5%</td>
<td>26/92</td>
</tr>
<tr>
<td>iso-O2</td>
<td>174</td>
<td>15.86/55870</td>
<td>628</td>
<td>600</td>
<td>4.6%</td>
<td>28/85</td>
</tr>
<tr>
<td>cint-O0</td>
<td>115</td>
<td>5.21/77002</td>
<td>9027</td>
<td>8836</td>
<td>2.1%</td>
<td>99/235</td>
</tr>
<tr>
<td>cint-O1</td>
<td>115</td>
<td>1.75/2146</td>
<td>4123</td>
<td>3996</td>
<td>3.1%</td>
<td>42/129</td>
</tr>
<tr>
<td>cint-O2</td>
<td>115</td>
<td>1.51/11490</td>
<td>3067</td>
<td>2928</td>
<td>4.6%</td>
<td>39/263</td>
</tr>
<tr>
<td>insertsort-O0</td>
<td>91</td>
<td>353.86/11455293</td>
<td>1533</td>
<td>1500</td>
<td>2.2%</td>
<td>40/115</td>
</tr>
<tr>
<td>insertsort-O1</td>
<td>91</td>
<td>11.68/387292</td>
<td>1326</td>
<td>1320</td>
<td>0.4%</td>
<td>43/108</td>
</tr>
<tr>
<td>mat-O0</td>
<td>497</td>
<td>7.89/30644616</td>
<td>30968</td>
<td>30722</td>
<td>0.8%</td>
<td>172/215</td>
</tr>
<tr>
<td>mat-O1</td>
<td>497</td>
<td>10.29/3687230</td>
<td>11701</td>
<td>11568</td>
<td>1.1%</td>
<td>61/124</td>
</tr>
<tr>
<td>mat-O2</td>
<td>497</td>
<td>70.93/1030746</td>
<td>7280</td>
<td>7236</td>
<td>0.6%</td>
<td>566/363</td>
</tr>
</tbody>
</table>

Table I: Summary of the Results.

![Figure 9. An example Program](attachment:figure.png)
predicate does not influence the set of future traces and thus can be discarded. Clearly, the state with the lowest current time should not be explored as the other one will for sure yield a larger execution time. This kind of pruning is not possible in the current version of UPPAAL as such two states will always be considered different and both of them explored. If the CFG has nested branching, this results in an exponential reduction of the explored state space. Notice also that the previous pruning is valid whether the hardware exhibits timing anomalies or not.

XI. Conclusion and Future Work

We have presented a framework based on program slicing and model-checking to compute WCET for programs running on architectures featuring pipelining and caching. We have exemplified the method by providing formal models of the ARM920T. Moreover we have compared the computed results with real execution times on the real hardware and showed we can achieve unmatched tightness. Our work supports the claim that real-time model checking, when combined with advanced techniques like slicing to build program models, is a practical method to compute tight WCET.

Our method has several advantages: (1) it is based on a formal approach, based on efficient techniques; (2) it is 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; as already emphasized in [22], changes in the processor speed can also be modelled easily (using a timed automaton that sets the processor speed); (3) it computes the CFG automatically. This avoids an error-prone, time-consuming and tedious step of the standard methods and tools and in this sense it is infinitely faster than the other methods because they all require a manual intervention (loop bound annotations). (4) it is insensitive to the so-called timing anomalies; (5) a witness (symbolic) trace, that produces the WCET, can be generated.

The current limitation of the implementation is the model-checking phase as already stated in Section X. The (UPPAAL) model-checking algorithm should be customized for the WCET problem and this is what we are currently investigating. The slicing algorithms [31], [30] we have implemented are linear in the size of the CFG and and scale up to very large programs.

References


In this section, we assume that we have the control flow graph of \( P \), \( CFG(P) \), which is a directed graph, the nodes of which are in \( P \). \( CFG(P) \) has a single entry node (initial instruction of the \( P \)) and a single exit node (end of \( P \)). An example of a CFG for the Fibonacci Program 1 is given in Figure 2.

A slice criterion \( C \) for \( P \) is a subset \( I' \subseteq P \), and for each instruction \( \iota \in I' \) an associated subset of “variables” \( V(\iota) \subseteq R \cup P \cup M \). We assume that \( V(\iota) \) is actually included in the set of registers that the instruction operates on but this is inessential. For instance, a slice criterion for program \( FIBO_0 \) of listing 1 can be instruction 88 : \( cmp r2, r3 \) and associated set \{ \( r_2 \), \( r_3 \) \}.

Given input data \( d \in D \), we write \( run(P, d) \) to denote the (unique) run of \( P \) on \( d \). Let \( S \subseteq P \). The runs\(^\text{14} \) of \( P \) and \( S \) on input data \( d \in D \) are denoted

\[
run(P, d) = s_1 t_1 s_2 t_2 \cdots s_k t_k \cdots s_n t_n s_{n+1}
\]

\[
run(S, d) = s'_1 t'_1 s'_2 t'_2 \cdots s'_k t'_k \cdots s'_m t'_m s'_{m+1}
\]

Define the projection \( proj(s, \iota) \) for a pair \((s, \iota) \in LI\) by:

\[
proj(s, \iota) = \begin{cases} 
\varepsilon & \text{if } \iota \not\in S \\
(proj_{V(\iota)}(s), \iota) & \text{otherwise.}
\end{cases}
\]

i.e., instructions not in the subset \( S \) are ignored (replaced by \( \varepsilon \), the empty word) and for instructions in \( S \) we keep the projection on \( V(\iota) \) of the program state. \( proj \) is extended in the natural way to traces and we let \( proj^*(\varepsilon) = \varepsilon \) and \( proj^*(w.(s, \iota)) = proj^*(w).proj^*(s, \iota) \) with \( s \) a program state and \( \iota \in LI \).

\( S \) is a slice of \( P \) for the slice criterion \( C \) if it satisfies, for every input data \( d \in D \):

1) if \( P \) terminates on input \( d \) then \( S \) terminates on input \( d \) and

2) \( proj^*(run(P, d)) = proj^*(run(S, d)) \). Notice that by definition of \( proj \), all the instructions of \( S \) are in \( proj^*(run(S, d)) \) but the projection restricts the set \( s'_k \) to the variables in \( V(i'_k) \).

Given a slice criterion, it is possible to compute a slice by computing data dependences and control dependences on the CFG. This can be effectively implemented using post-dominators algorithms [30], [31] and post dominance frontiers algorithms [32].

Once the slice is obtained, it suffices to simulate the sliced program to collect the values of the variables of the slice.

\(^{14}\)Notice that at this stage, \( run(S, d) \) may not be finite and program \( S \) may not terminate.