WCET of OCaml Bytecode on Microcontrollers: An Automated Method and Its Formalisation

Steven Varoumas
Sorbonne Université, CNRS, Laboratoire d'Informatique de Paris 6, LIP6
F-75005, Paris, France
Cnam, Centre d'études et de recherche en informatique et communications, Cédric
292 rue Saint Martin, 75003, Paris, France

Tristan Crolard
Cnam, Centre d'études et de recherche en informatique et communications, Cédric
292 rue Saint Martin, 75003, Paris, France

Abstract

Considering the bytecode representation of a program written in a high-level programming language enables portability of its execution as well as a factorisation of various possible analyses of this program. In this article, we present a method for computing the worst-case execution time (WCET) of an embedded bytecode program fit to run on a microcontroller. Due to the simple memory model of such a device, this automated WCET computation relies only on a control-flow analysis of the program, and can be adapted to multiple models of microcontrollers. This method evaluates the bytecode program using concrete as well as partially unknown values, in order to estimate its longest execution time. We present a software tool, based on this method, that computes the WCET of a synchronous embedded OCaml program. One key contribution of this article is a mechanically checked formalisation of the aforementioned method over an idealised bytecode language, as well as its proof of correctness.

2012 ACM Subject Classification Computer systems organization → Embedded software; Computer systems organization → Real-time systems

Keywords and phrases Worst-case execution time, microcontrollers, synchronous programming, bytecode, OCaml

Digital Object Identifier 10.4230/OASIcs.WCET.2019.5

Supplement Material `http://stevenvar.github.io`

1 Introduction

Due to their low cost and efficient power use, microcontrollers are heavily used by the embedded system industry. Nonetheless, the very limited memory and power resources of these devices has lead developers to use traditional low-level languages such as C or assembly. While being precise and powerful, these languages often lack the hardware abstraction that would allow programmers that are not system programming experts to write various applications, from home automation to more critical applications. For this reason, many projects have emerged that make it possible to run, on microcontrollers with less than 10 kilo-bytes (kb) of RAM, programs written in higher-level languages such as Java [3], Scheme [19] [6], or OCaml [24]. The runtime environments developed in these projects usually include an optimised *virtual machine* that interprets bytecode of the language directly on the device. Beyond the usual hardware abstraction layer, these languages offer a higher-level expressiveness for the development of programs, and more guarantees over them (thanks to strong static type systems and automatic memory management among other features).

Timing constraints are frequent in embedded systems: they must often retain quick reaction times, in order to handle rapid changes in their environments. Computing the worst-case execution time (WCET) of the program's routine responsible for reacting to input stimuli is thus necessary, in order to ensure that no input is ignored during execution, and thus prevent a potential incorrect or dangerous behaviour from the system. A usual method of handling these timing constraints is to rely on schedulability analyses together with the use of *real-time operating systems* such as FreeRTOS [7]. However, those systems are often quite heavy and not particularly adapted for microcontrollers with low memory resources. For this reason, the approach we present relies instead on the synchronous programming model. This model offers a lightweight reactive and concurrent programming model, particularly adapted to the applications of microcontrollers and their material limitations [21], as well as advantageous properties for WCET computing: by construction, the generated code does not contain loops and does not need annotations.

In this article, we use a toolchain stemming from our previous works that consists in an OCaml virtual machine implementation fit for running on various models of microcontrollers, as well as a compatible *synchronous programming* extension to the OCaml language. This solution enjoys the high-level features of the OCaml programming language in order to develop *safer* programs for embedded system, and benefits from a model of concurrent programming adapted to the scarce memory resources of microcontrollers. Our approach, which uses a bytecode representation of embedded OCaml programs, provides *portability*, and enables a *modularisation* of various analyses done over these programs.

The main contribution presented in this article is an automated method that computes the WCET of a bytecode program on partially unknown inputs, together with its correctness proof. The method is implemented as a prototype tool called *Bytecrawler* which supports any OCaml bytecode instruction found in a compiled synchronous program. The formalisation is currently based on a small idealised bytecode language, all the proofs have however been mechanically checked.

Section 2 presents some preliminaries: we introduce the OMicroB virtual machine [23] and OCaLustre, a synchronous extension to the OCaml language [21]. Section 3 describes our method for computing the WCET of OCaml bytecode, and its implementation. Section 4 contains the formalisation and the correctness proof of the method. Section 5 concludes with a short discussion about related and future works.

2 Preliminaries

2.1 OMicroB: an OCaml virtual machine for microcontrollers

OMicroB [23] is an implementation of the OCaml virtual machine (named *ZAM - Zinc Abstract Machine* [11]) dedicated to run OCaml bytecode on microcontrollers with very scarce resources (typically, less than 10kb of RAM and less than 100kb of flash memory). Lightweight and configurable, it provides every feature of the language and its runtime, such as a strong static type system with type inference, the use of various programming paradigms (functional, imperative, modular, and object-oriented), as well as automatic memory management with garbage collection (GC). OMicroB is designed to be portable, due to its implementation in standard C, that we consider as a *portable assembly language*, since most of the target models of microcontrollers come with a dedicated C compiler. We were successful in porting OMicroB to AVR microcontrollers, and ports to ARM Cortex-M0 (used by BBC micro:bit cards) and PIC32 are promising works in progress.

The main component of OMicroB is an optimised stack-based bytecode interpreter, capable of running all the 149 bytecode instructions of the language. This interpreter handles various registers (such as a stack pointer, a code pointer, an accumulator, etc.) whose values are accessed and modified by the instructions of the running OCaml program. Moreover, the interpreter may invoke various low-level primitives (mostly dealing with input/output operations) that are directly implemented in C.

As shown in Figure 1, in order to execute an OCaml program on a microcontroller, the source code is first compiled into bytecode (by the standard *ocamlc* compiler), which is then "cleaned" by the *ocamlclean* tool which performs dead-code elimination. This bytecode program is then embedded into a C source code as an array of bytes by a tool called *bc2c*. It is then compiled together with the OCaml runtime library and interpreter by some platform-dependent C compiler and linker, and thus turned into an executable. For simulation purposes, the program can also be compiled for (and executed on) a generic PC.

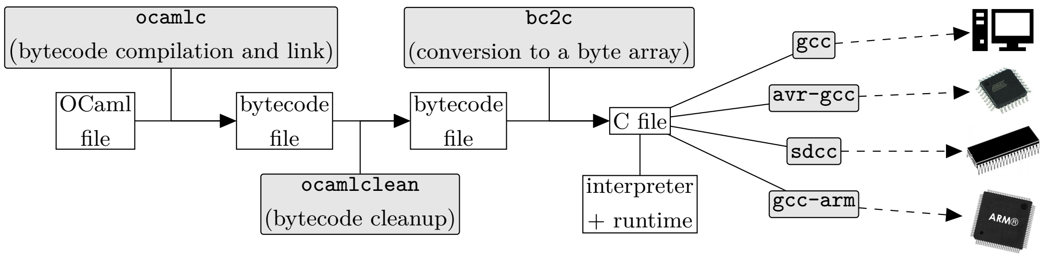

Figure 1 Compilation of an OCaml program with OMicroB (taken from [23]).

OMicroB offers a safer and more expressive model of programming than the classical C and assembly languages that are traditionally used to program microcontrollers. The robustness and the greater hardware abstractions gained by such a higher-level programming language makes it easier for developers that are not particularly well-versed in hardware specifics to develop programs for embedded systems.

2.2 OCaLustre: a synchronous extension of OCaml

Due to their constant interactions with their environment, embedded systems must react quickly to various stimuli. A model of concurrent programming, suitable for the size of the limited resources of microcontrollers might thus be a welcomed addition to the programming of microcontrollers [21]. Therefore, we use OCaLustre [22], a synchronous programming extension of OCaml, inspired by the Lustre [4] language. Synchronous programming rely on the main principle that computations made during a *logical instant* are instantaneous: that is, the time required at each instant by the program to compute its output from its input should be ignored when reasoning about programs.

In OCaLustre, as in Lustre, the elementary synchronous component is called a *node*: a function that takes flows of values as input and computes output flows. The body of a node is a system of equations, defining the set of output or local values. For example, the `count` node of figure 2 takes an `r` flow as an input, and returns the `cpt` flow as its output. The `->>` operator, equivalent to the `fby` ("followed-by") Lustre operator, has the following semantics: `x = a ->> b` means that `x` is equal to `a` for the first instant, and then to the *previous* value of `b` (i.e. the value computed for `b` at the preceding synchronous instant) for the subsequent instants. Therefore, `count` computes the series of natural numbers, reset to zero when the value of the `r` flow is true.

```
let%node count r ~return:cpt =
  aux = 0 ->> (cpt + 1);
  cpt = if r then 0 else aux
```

Figure 2 A synchronous node.

During the compilation of an OCaLustre program, each node is separately turned into standard OCaml code, compatible with the `ocamlc` bytecode compiler, and thus with OMicroB. Our compilation method, derived from the *single loop code generation* used by most Lustre compilers, produces lightweight code, compatible with the resources of microcontrollers. It also provides guarantees over the program, such as the absence of *causality loops* (i.e. flows should not mutually depend on each other during the same instant).

3 Worst case execution time analysis of OCaml bytecode

In order to compute the WCET of a synchronous OCaLustre program, we created *Bytecrawler*, a tool that computes an upper bound of the execution time of each synchronous instant. This tool has the main advantage of working over the *bytecode* instructions of the virtual machine, rather than the underlying native-code instructions. It relies on the fact that bytecode analyses and platform-dependent analyses are separated ([9]) to free the application developer from needing to put additional annotations in their program: loop bounds for the program are not required due to the chosen programming model, and lower-level annotations, that can appear in the bytecode interpreter or the runtime library, are provided by the platform developer.

3.1 Bytecrawler WCET computation function

Values of variables that come from the environment of the program (such as values returned by electronic sensors) are unknown at compile-time, and thus introduce an *external non-determinism.* Bytecrawler can thus be seen as an *abstract* bytecode interpreter, extended to deal with unknown values. It is based on the standard method of [14] that considers all possible paths of the program to compute the costlier one, but Bytecrawler refines this evaluation by performing a hybrid execution that uses *concrete* values whenever possible. The program is normally executed by Bytecrawler with all values that are *known* at compile-time, but the evaluation function is generalised to handle *unknown* values: when reaching a conditional branching bytecode instruction (namely, `BRANCHIF` and `BRANCHIFNOT`) whose condition's value is unknown, Bytecrawler will explores the different possible paths.

Figure 3 is an excerpt of Bytecrawler's WCET computation function. It computes an upper bound of the timing cost (in cycles) of a given OCaml program using a *cost* function that maps each bytecode instruction to its cycle count, and following the bytecode semantics. In particular, the `CCALL` instruction corresponds to the call to an I/O primitive (written in C): the return value of such a call is always unknown at compile time.

Note that, while this method might work for *any* OCaml program, some limitations quickly appear: any looping program whose number of iterations depends of an unknown value could not be bounded, and the triggering of the garbage collection (GC) algorithm could break the WCET estimation. For this reason, Bytecrawler is better suited to work only on the synchronous extension of OCaml, which enjoys the nice properties that instants do not contain any loop, and they handle only basic data types which never trigger the GC.

```
let rec wcet state =
  let state' = {state with pc = state.pc + 1} in
  let instr = state.instrs.(pc) in
  cost instr + match instr with
  | CONST i -> wcet {state' with accu = Int i}
  | BRANCH ptr -> wcet {state with pc = ptr}
  | BRANCHIF ptr ->
    (match state.accu with
    | Int 0 -> wcet state'
    | Int _ -> wcet {state with pc = ptr}
    | Unknown -> max (wcet {state with pc = ptr}) (wcet state'))
  | ADDINT ->
    (match state.accu, state.stack with
    | Int x, (Int y)::s -> wcet {state' with accu = Int (x+y); stack = s}
    | _, _ -> wcet {state' with accu = Unknown})
  | CCALL _ -> wcet {state' with accu = Unknown}
  | STOP -> 0
```

Figure 3 Bytecrawler WCET computation function (excerpt).

▶ Remark. Symbolic execution [1] might allow us to refine even more the WCET estimate by pruning unreachable paths. Such an analysis for a stack-based virtual machine seems more involved than for imperative languages. For instance, the symbolic execution of Java bytecode is clearly not straightforward [13]). Moreover, functional programs usually feature higher-order functions and the OCaml virtual machine is tailored for an efficient evaluation of such programs. A recent experiment with a symbolic execution for Haskell [8] shows that defunctionalisation into some intermediate first-order functional language might be required. While our synchronous extension does not currently include higher-order functions, translating back from this higher-order bytecode language to a first-order intermediate representation would already prove difficult. We thus keep the study of this method for a future work.

3.2 Cost function for bytecode instructions

The cost function which associates each bytecode instruction to a cycle count is a finite map, represented as a table. This table can be computed by a classic WCET tool (such as the Bound-T execution time analyser [10]), configured for the correct model of microcontroller. A key benefit of our approach resides in the fact that this table must be computed *only once* for each microcontroller. The targeted devices are supposed to not induce *timing anomalies* [15] due to their simple hardware model (cache-less, with in-order pipelines which are not timing anomalous [5]): this ensures that the computed cost of each bytecode instruction remains the same for every given OCaml program, and provides *compositionality* of the timing analysis.

For most of the bytecode instructions, the execution time is a constant, and our method thus does not overestimate their execution time. For some other instructions, the execution time is dependent on some argument (for example, the `APPTERM` instruction which performs tail calls depends on the number of parameters of the function) which is explicit in the bytecode: their cost can be pre-computed for every realistic instruction/parameter pair. For some of the remaining bytecode instructions (for instance, those dealing with higher-order closures), the execution time might be more difficult to bound statically. However, these instructions do not appear, by hypothesis, in the considered bytecode. Lastly, calls to C primitives (via the `CCALL` bytecode instruction) may require an execution time that depends on the values of the arguments of the function. The WCET of these calls might thus overestimate the actual execution time of the `CCALL` evaluation. Handling these primitive calls relies on another cost table that maps each C primitive name to its maximum cycle count. The costs of these primitives are hardware-specific and need to be provided by the platform developer. Each primitive cost should be computed using classical methods and tools for WCET analysis.

▶ **Example.** The excerpt of OCaml bytecode on the left side of figure 4 corresponds to the body of the step function created by compilation of the `count` node of figure 2. On the right side of this figure is displayed a table associating a cost to every bytecode instruction, which has been computed by *Bound-T* (configured for an AVR ATmega32U4 microcontroller). From this table, Bytecrawler computes a maximum cost of 2121 cycles for this function. For sake of simplicity, calls to I/O primitives (via the `CCALL` bytecode instruction) are not mentioned in this example. For instance, in a complete synchronous program, calls to input (resp. output) primitives happen in the beginning (resp. end) of each synchronous instant. These calls should thus also be taken into account.

```
(...)
69  ACC0
70  GETFIELD0
71  PUSHACC2
72  BRANCHIFNOT 75
73  CONST0
74  BRANCH 76
75  ACC0
76  PUSHACC0
77  OFFSETINT 1
78  PUSHACC0
79  PUSHACC4
80  SETFIELD0
81  ACC1
82  PUSHACC4
83  SETFIELD1
84  CONST0
```

Instruction	Cost (cycles)
ACC0	74
ACC1	74
CONST0	66
GETFIELD0	96
SETFIELD0	145
SETFIELD1	150
PUSHACC0	95
PUSHACC2	115
PUSHACC4	115
BRANCH	299
BRANCHIFNOT	315
OFFSETINT	301
(...)	(...)

Figure 4 OCaml bytecode (left) and instructions cost table (right).

4 Formalisation and correctness proof

In this section, we describe a formalisation of the inner working of the Bytecrawler tool. For this purpose, we first introduce an idealised bytecode language that serves as an illustration on a small subset of OCaml bytecode instructions. We then formalise how to compute an upper bound of the WCET for a program written with these instructions, and we prove the correctness of this computation. Our conjecture is that, because of the imperative nature of the OCaLustre programs, and because of the absence of dynamic memory allocation during the execution of a synchronous instant, the results over this idealised bytecode language can be transposed to the actual subset of the OCaml instructions coming from the compilation of an OCaLustre source code.

The specifications presented in this section have been developed with the *Ott* tool [18], and the various lemmas and theorems were proven using the *Coq* proof assistant [20]. The corresponding proof scripts are available at `http://stevenvar.github.io`.

The idealised bytecode language contains instructions for initialising variables (Init), assigning values to variables (Assign), arithmetic operations over integers (Sub and Add), branching (Branch and Branchif), as well as an instruction to terminate execution (Stop).

$$instr ::= \mathsf{Init}\ x\ v \mid \mathsf{Assign}\ x\ y \mid \mathsf{Add}\ x\ y \mid \mathsf{Sub}\ x\ y \mid \mathsf{Branch}\ v \mid \mathsf{Branchif}\ x\ v \mid \mathsf{Stop}$$
$$values, v, w ::= int\_litteral \mid v + w \mid v - w$$

A state σ of a program is a tuple (P, pc, M) that contains an array P representing the instructions of the program, a code pointer pc, and a memory M that associates any initialised variable name to its value. The small-step operational semantics of the language is defined on figure 5.

$$\frac{P[pc] = \mathsf{Init}\, x\, v}{(P, pc, M) \longrightarrow (P, pc+1, M[x := v])} \qquad \frac{P[pc] = \mathsf{Branch}\, v}{(P, pc, M) \longrightarrow (P, v, M)}$$

$$\frac{P[pc] = \mathsf{Assign}\, x\, y \quad M[y] = v}{(P, pc, M) \longrightarrow (P, pc+1, M[x := v])} \qquad \frac{P[pc] = \mathsf{Branchif}\, x\, v \quad M[x] = 0}{(P, pc, M) \longrightarrow (P, pc+1, M)}$$

$$\frac{P[pc] = \mathsf{Add}\, x\, y \quad M[x] = v \quad M[y] = w}{(P, pc, M) \longrightarrow (P, pc+1, M[x := v+w])} \qquad \frac{P[pc] = \mathsf{Branchif}\, x\, v \quad M[x] \neq 0}{(P, pc, M) \longrightarrow (P, v, M)}$$

$$\frac{P[pc] = \mathsf{Sub}\, x\, y \quad M[x] = v \quad M[y] = w}{(P, pc, M) \longrightarrow (P, pc+1, M[x := v-w])}$$

Figure 5 Operational semantics of the idealised bytecode language.

▶ **Definition 1** (Execution trace). *An execution trace T is the sequence of states taken by the program, following the operational semantics of the language.*

During the actual execution of a program, the values of all variables are *known*, and the corresponding execution trace is unique. Moreover, when the computation terminates normally, the trace ends on instruction Stop. We call such a trace *deterministic*:

▶ **Definition 2** (Deterministic trace). *The* run *function computes the deterministic execution trace beginning with a state σ:*

$$\frac{P[pc] = \mathsf{Stop}}{run((P, pc, M)) = [(P, pc, M)]} \qquad \frac{\sigma \longrightarrow \sigma' \quad run(\sigma') = \mathcal{T}}{run(\sigma) = \sigma \ :: \ \mathcal{T}}$$

▶ Remark. Note that, by definition, a deterministic trace is always finite. Indeed, since it is a certainty (due to absence of loops or recursion) that the actual execution of a synchronous instant will terminate, we are only interested in finite traces.

We associate a cost to every language instruction with the $cost_{instr} : instr \to nat$ function. The cost of a transition is equal to the cost of the corresponding instruction, and the cost of a trace is the sum of the costs of all transitions:

$$cost_{step}(P, pc, M) = cost_{instr}(P[pc]) \qquad cost(\mathcal{T}) = \sum_{\sigma \in \mathcal{T}} cost_{step}(\sigma)$$

4.1 Erasure of variables

In order to compute an upper-bound for the execution time of an OCaLustre program, we reason over an *abstract* representation of the program which considers every possible path that the control flow can take. In this abstract representation, variables can hold *unknown* values, that represent the values that depend on the state of the program environment during execution. The grammar of the idealised bytecode language is extended accordingly, where unknown values are denoted by $\bot$:

$$values, v, w ::= int\_litteral \mid v + w \mid v - w \mid \bot$$

The memory of an abstract execution of a program might thus contain unknown values. We call such a memory an *erasure* of the actual memory of the program:

▶ **Definition 3** (Erasure). *A memory M' is an erasure of a memory M (written $M \searrow M'$) if M' and M share the same set of variables, but M' contains more unknown variables than M. The following induction rules formally define the erasure of a memory:*

$$\frac{}{\varnothing \searrow \varnothing} \qquad \frac{M \searrow M'}{M \cup x = v \searrow M' \cup x = v} \qquad \frac{M \searrow M'}{M \cup x = v \searrow M' \cup x = \bot}$$

By extension, if two states σ and σ' differ only by the fact that the memory of σ' is an erasure of the memory of σ, we will use the same notation: $\sigma \searrow \sigma'$ and say that σ' is an erasure of σ.

Since it depends only on the code pointer and the program instructions, the cost of a transition stays the same after an erasure of the memory of the program:

▶ **Lemma 4.** $\forall \sigma\ \sigma', \sigma \searrow \sigma' \Rightarrow cost_{step}(\sigma) = cost_{step}(\sigma')$

The `wcet` function of figure 3 computes the worst-case execution time of a program by always considering the maximum cost between two possible branches. In our formalism, this function is implemented by the $cost_{max}$ function as follows:

▶ **Definition 5** (Maximum cost). *The $cost_{max}$ function computes the maximum cost of a program:*

$$\frac{P[pc] = \mathsf{Stop}}{cost_{max}((P, pc, M)) = cost_{instr}(\mathsf{Stop})}$$

$$\frac{\sigma \longrightarrow \sigma' \quad cost_{step}(\sigma) = c \quad cost_{max}(\sigma') = k}{cost_{max}(\sigma) = c + k}$$

$$\frac{\begin{array}{l} P[pc] = \mathsf{Branchif}\ x\ v \\ M[x] = \bot \\ cost_{step}((P, pc, M)) = c \\ cost_{max}((P, pc+1, M)) = k \\ cost_{max}((P, v, M)) = k' \end{array}}{cost_{max}((P, pc, M)) = c + max(k, k')}$$

The main result of this paper can now be stated as follows: the maximum cost of a program, computed from an erasure of its initial state, is greater than the actual execution time of the program.

▶ **Theorem 6** (Correctness). $\forall\ \sigma\ \mathcal{T}, run(\sigma) = \mathcal{T} \Rightarrow \forall\ \sigma',\ \sigma \searrow \sigma' \Rightarrow cost_{max}(\sigma') \geq cost(\mathcal{T})$

In particular, the $cost_{max}$ function can be applied to a memory where all variables are unknown (equivalent to the initial state of a synchronous instant) since this memory is an obvious erasure of all possible initial states. The result obtained is thus an upper bound of the execution times of all possible program runs.

The remainder of this section is devoted to the proof of this theorem.

4.2 Non-deterministic evaluation

The use of unknown values ($\bot$) induces a non-determinism on the evaluation semantics of the language: branching choices might depend on such variables, and thus it becomes statically impossible to guess which path would be the actual one. We thus introduce a *non-deterministic operational semantics* of the language which is a standard abstraction of the deterministic operational semantics where conditional branching is generalised to non-deterministic branching. The rules for this new semantics, denoted by $\rightsquigarrow$, are the same as the rules for the deterministic semantics, extended with the two rules given in figure 6 (and the obvious lifting of arithmetic operations to account for $\bot$).

$$\frac{P[pc] = \mathsf{Branchif}\ x\ v \quad M[x] = \bot}{(P, pc, M) \rightsquigarrow (P, pc+1, M)} \qquad \frac{P[pc] = \mathsf{Branchif}\ x\ v \quad M[x] = \bot}{(P, pc, M) \rightsquigarrow (P, v, M)}$$

Figure 6 Rules for non-deterministic branching.

▶ **Definition 7** (Non-deterministic trace). *A non-deterministic trace is a finite sequence of transitions following the non-deterministic semantics of the language.*

Because the non-deterministic semantics of the language is an extension of its deterministic semantics, a transition in the former stays the same in the latter. We say that it is *embedded* into the non-deterministic semantics:

▶ **Lemma 8.** $\forall\ \sigma\ \sigma',\ (\sigma \to \sigma') \Rightarrow (\sigma \rightsquigarrow \sigma')$

Moreover, after erasing a memory, any transition from the deterministic semantics is still possible (possibly among other possible non-deterministic transitions):

▶ **Lemma 9.** $\forall \sigma_1\ \sigma_2\ \sigma_1', (\sigma_1 \rightsquigarrow \sigma_2) \wedge (\sigma_1 \searrow \sigma_1') \Rightarrow (\exists\ \sigma_2',\ (\sigma_1' \rightsquigarrow \sigma_2') \wedge (\sigma_2 \searrow \sigma_2'))$

By combining the two previous lemmas, we state that any erasure preserves the transition when going from the deterministic semantics to the non-deterministic one:

▶ **Theorem 10** (Preservation). $\forall\ \sigma_1\ \sigma_2\ \sigma_1'$,
$(\sigma_1 \to \sigma_2) \wedge (\sigma_1 \searrow \sigma_1') \Rightarrow (\exists\ \sigma_2',\ (\sigma_1' \rightsquigarrow \sigma_2') \wedge (\sigma_2 \searrow \sigma_2'))$

As illustrated by figure 7, the previous embedding and preservation properties hold for traces.

$$\begin{tikzcd}
\sigma_1 \arrow[r] \arrow[dr, dashed] & \sigma_2 \arrow[r] \arrow[dr, dashed] & \cdots \arrow[r] & \sigma_n \arrow[dr, dashed] & & \\
& \sigma_1' \arrow[r, rightsquigarrow] \arrow[dr, dashed] & \sigma_2' \arrow[r, rightsquigarrow] \arrow[dr, dashed] & \cdots \arrow[r, rightsquigarrow] & \sigma_n' \arrow[dr, dashed] & \\
& & \sigma_1'' \arrow[r, rightsquigarrow] & \sigma_2'' \arrow[r, rightsquigarrow] & \cdots \arrow[r, rightsquigarrow] & \sigma_n''
\end{tikzcd}$$

Figure 7 Trace preservation.

4.3 Maximum trace

In order to prove that $cost_{max}$ is the cost of the "most expensive" run of the program, we need to formally define the notion of a maximum trace. We thus defined the run_{max} function and proved that it actually computes a non-deterministic trace with the maximum cost.

▶ **Definition 11** (Maximum trace). *The run_{max} function is inductively defined as follows:*

$$\frac{P[pc] = \mathsf{Stop}}{run_{max}((P, pc, M)) = (P, pc, M) \ :: \ \varnothing} \qquad \frac{\sigma \longrightarrow \sigma' \quad run_{max}(\sigma') = \mathcal{T}}{run_{max}(\sigma) = (\sigma \ :: \ \mathcal{T})}$$

$$\frac{\begin{array}{l} P[pc] = \mathsf{Branchif}\ x\, v \\ M[x] = \bot \\ run_{max}((P, pc+1, M)) = \mathcal{T} \\ run_{max}((P, v, M)) = \mathcal{T}' \\ cost(\mathcal{T}) \leq cost(\mathcal{T}') \end{array}}{run_{max}((P, pc, M)) = ((P, pc, M) \ :: \ \mathcal{T}')} \qquad \frac{\begin{array}{l} P[pc] = \mathsf{Branchif}\ x\, v \\ M[x] = \bot \\ run_{max}((P, pc+1, M)) = \mathcal{T} \\ run_{max}((P, v, M)) = \mathcal{T}' \\ cost(\mathcal{T}) > cost(\mathcal{T}') \end{array}}{run_{max}((P, pc, M)) = ((P, pc, M) \ :: \ \mathcal{T})}$$

As expected, $cost_{max}$ and run_{max} are related by the following equality:

▶ **Lemma 12.** $\forall\ \sigma,\ cost_{max}(\sigma) = cost(run_{max}(\sigma))$

Using the trace preservation lemma, we derive the key property of this proof of correctness: the cost of the maximum trace is greater than the cost of every other finite trace, even when considering an erasure of their memories. In other words, the cost of the maximum trace computed from an erased state is greater than the actual cost of the program:

▶ **Lemma 13.** $\forall\ \sigma\ \mathcal{T},\ run(\sigma) = (\sigma :: \mathcal{T}) \Rightarrow \forall\ \sigma',\ \sigma \searrow \sigma' \Rightarrow cost_{max}(\sigma') \geq cost(\sigma :: \mathcal{T})$

The combination of the previous lemmas allows us to conclude with our main theorem stating that the cost, estimated from an erasure of the initial memory, is an upper bound of the real cost of any actual run of the program:

▶ **Theorem 14** (Correctness). $\forall\ \sigma\ \mathcal{T},\ run(\sigma) = \mathcal{T} \Rightarrow \forall\ \sigma',\ \sigma \searrow \sigma' \Rightarrow cost_{max}(\sigma') \geq cost(\mathcal{T})$

5 Conclusion, related and future works

We presented a virtual machine approach to microcontrollers programming that makes it possible to run programs developed with a high-level, multi-paradigms, programming language on devices with limited resources. This language, OCaml, is extended with a synchronous programming model which is adapted to the concurrent nature of embedded programs. In particular, our approach provides a tool, called Bytecrawler, that makes it possible to compute the WCET of a synchronous instant by reasoning over the bytecode instructions of the program, thus providing an automated, factorised method to compute the reaction time of a program. The aforementioned method has been formalised on a idealised language, and a proof of its correctness has been developed in the Coq proof assistant. To the best of our knowledge, a formalised and automated WCET estimation of OCaml bytecode has not been done before. Similar studies exist on other embedded virtual machines for different programming languages (in particular, multiple works have been done on Java bytecode [2][16][9]), although these results do not seem to have been formally verified. Some recent work on proving the correctness of a WCET estimation tool is available [12], but this tool is integrated within the CompCert C compiler, and thus operates only over C programs.

As a work in progress, we are adapting the formalisation and proof presented in this article to the actual, stack-based, language of OCaml bytecode instructions, in order to extract a certified implementation of Bytecrawler from Coq. As discussed in section 3, symbolic execution could improve the WCET estimate, but adapting it to the OCaml virtual machine might be difficult. Concolic execution [17], a hybrid model of execution that mixes *concrete* and *symbolic* execution, seems close to our method, and might also refine WCET estimation.

Finally, it is worth mentioning that our solution could quite easily be adapted to other kinds of analysis: for example, modifying the cost function to indicate the number of words allocated in memory by each bytecode instruction could provide an upper bound for the total amount of memory used by the program, and thus be an estimate of the maximum memory usage of any given OCaml program.

References

1 Clément Ballabriga, Julien Forget, and Giuseppe Lipari. Symbolic WCET computation. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(2):39, 2018. `doi:10.1145/3147413`.

2 Guillem Bernat, Alan Burns, and Andy J. Wellings. Portable worst-case execution time analysis using Java Byte Code. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems (ECRTS 2000), Stockholm, Sweden, June 19-21, 2000*, pages 81–88, New York, NY, USA, 2000. IEEE. `doi:10.1109/EMRTS.2000.853995`.

3 Niels Brouwers, Koen Langendoen, and Peter Corke. Darjeeling, a Feature-rich VM for the Resource Poor. In *Proceedings of the 7th International Conference on Embedded Networked Sensor Systems (SenSys 2009), Berkeley, CA, USA, November 4-6, 2009*, pages 169–182, New York, NY, USA, 2009. ACM. `doi:10.1145/1644038.1644056`.

4 Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A Declarative Language for Programming Synchronous Systems. In *Proceedings of the 14th annual ACM Symposium on Principles of Programming Languages (POPL 1987), Munich, Germany, January 21-23, 1987*, pages 178–188, New York, NY, USA, 1987. ACM. `doi:10.1145/41625.41641`.

5 Franck Cassez, René Rydhof Hansen, and Mads Chr. Olesen. What is a Timing Anomaly? In *Proceedings of the 12th International Workshop on Worst-Case Execution Time Analysis (WCET 2012), Pisa, Italy, July 10, 2012*, volume 23 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–12, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/OASIcs.WCET.2012.1`.

6 Marc Feeley and Danny Dubé. PICBIT: A Scheme system for the PIC microcontroller. In *Proceedings of the 4th Workshop on Scheme and Functional Programming, Boston, MA, USA*, pages 7–15, 2003. URL: `http://www.schemeworkshop.org/2003`.

7 Fei Guan, Long Peng, Luc Perneel, and Martin Timmerman. Open Source FreeRTOS As a Case Study in Real-time Operating System Evolution. *Journal of Systems and Software*, 118(C):19–35, August 2016. `doi:10.1016/j.jss.2016.04.063`.

8 William Hallahan, Anton Xue, and Ruzica Piskac. Building a Symbolic Execution Engine for Haskell. In *Proceedings of the 8th Workshop on Tools for Automatic Program Analysis (TAPAS 2017), New York, NY, USA, August 29, 2017*, 2017. URL: `https://cs.nyu.edu/acsys/tapas2017`.

9 Trevor Harmon and Raymond Klefstad. A Survey of Worst-Case Execution Time Analysis for Real-Time Java. In *Proceedings of the 21th International Parallel and Distributed Processing Symposium (IPDPS 2007), 26-30 March 2007, Long Beach, CA, USA*, pages 1–8. IEEE, 2007. `doi:10.1109/IPDPS.2007.370422`.

10 Niklas Holsti and Sam Saarinen. Status of the Bound-T WCET tool. In *Proceedings of the 2nd International Workshop on Worst-Case Execution Time Analysis (WCET 2002)*, Technical University of Vienna, Austria, June 2002. URL: `http://www.cs.york.ac.uk/rts/wcet2002`.

11 Xavier Leroy. The ZINC experiment: an economical implementation of the ML language. Technical Report 117, INRIA, 1990. URL: https://hal.inria.fr/inria-00070049/file/RT-0117.pdf.

12 André Oliveira Maroneze, Sandrine Blazy, David Pichardie, and Isabelle Puaut. A Formally Verified WCET Estimation Tool. In *Proceedings of the 14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014), Ulm, Germany, July 8, 2014*, volume 39 of *OpenAccess Series in Informatics (OASIcs)*, pages 11–20, Dagstuhl, Germany, 2014. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. doi:10.4230/OASIcs.WCET.2014.11.

13 Corina S. Pasareanu and Neha Rungta. Symbolic PathFinder: symbolic execution of Java bytecode. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE 2010), Antwerp, Belgium, September 20-24, 2010*, pages 179–180, New York, NY, USA, 2010. ACM. doi:10.1145/1858996.1859035.

14 Peter P. Puschner and Christian Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Real-Time Systems*, 1(2):159–176, 1989. doi:10.1007/BF00571421.

15 Jan Reineke, Björn Wachter, Stefan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A Definition and Classification of Timing Anomalies. In *Proceedings of the 6th International Workshop on Worst-Case Execution Time Analysis (WCET '06), Dresden, Germany, July 4, 2006*, volume 4 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2006. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/OASIcs.WCET.2006.671.

16 Martin Schoeberl and Rasmus Pedersen. WCET analysis for a Java processor. In *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems (JTRES '06), Paris, France, October 11-13, 2006*, pages 202–211, New York, NY, USA, 2006. ACM. doi:10.1145/1167999.1168033.

17 Koushik Sen. Concolic testing. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), Atlanta, Georgia, USA, November 5-9, 2007*, pages 571–572, New York, NY, USA, 2007. ACM. doi:10.1145/1321631.1321746.

18 Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, et al. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming*, 20(1):71–122, 2010. doi:10.1017/S0956796809990293.

19 Vincent St-Amour and Marc Feeley. PICOBIT: a compact Scheme system for microcontrollers. In *Proceedings of the 21st International Symposium on Implementation and Application of Functional Languages (IFL 2009), South Orange, NJ, USA, September 23-25, 2009*, pages 1–17. Springer, 2009. doi:10.1007/978-3-642-16478-1_1.

20 The Coq Development Team. The Coq Proof Assistant, version 8.9.0, January 2019. doi:10.5281/zenodo.2554024.

21 Steven Varoumas, Benoît Vaugon, and Emmanuel Chailloux. Concurrent Programming of Microcontrollers, a Virtual Machine Approach. In *Proceedings of the 8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, Toulouse, France, 2016. URL: https://hal.archives-ouvertes.fr/ERTS2016.

22 Steven Varoumas, Benoît Vaugon, and Emmanuel Chailloux. OCaLustre : une extension synchrone d'OCaml pour la programmation de microcontrôleurs. In *Vingt-huitièmes Journées Francophones des Langages Applicatifs (JFLA 2017)*, Gourette, France, 2017. URL: https://hal.archives-ouvertes.fr/JFLA2017.

23 Steven Varoumas, Benoît Vaugon, and Emmanuel Chailloux. A Generic Virtual Machine Approach for Programming Microcontrollers: the OMicroB Project. In *Proceedings of the 9th European Congress on Embedded Real Time Software and Systems (ERTS 2018)*, Toulouse, France, 2018. URL: https://hal.archives-ouvertes.fr/ERTS2018.

24 Benoît Vaugon, Philippe Wang, and Emmanuel Chailloux. Programming Microcontrollers in OCaml: The OCaPIC Project. In *Proceedings of the 17th International Symposium on Practical Aspects of Declarative Languages (PADL 2015), Portland, OR, USA, June 18-19, 2015*, pages 132–148. Springer, 2015. doi:10.1007/978-3-319-19686-2_10.