Certifying and Reasoning on Cost Annotations in C Programs

Nicolas Ayache[1,2], Roberto M. Amadio[1], and Yann Régis-Gianas[1,2]

[1] Université Paris Diderot (UMR-CNRS 7126)
[2] INRIA (Team πr^2)

Abstract. We present a so-called labelling method to enrich a compiler in order to turn it into a "cost annotating compiler", that is, a compiler which can *lift* pieces of information on the execution cost of the object code as cost annotations on the source code. These cost annotations characterize the execution costs of code fragments of constant complexity. The first contribution of this paper is a proof methodology that extends standard simulation proofs of compiler correctness to ensure that the cost annotations on the source code are sound and precise with respect to an execution cost model of the object code.

As a second contribution, we demonstrate that our label-based instrumentation is scalable because it consists in a modular extension of the compilation chain. To that end, we report our successful experience in implementing and testing the labelling approach on top of a prototype compiler written in ocaml for (a large fragment of) the C language.

As a third and last contribution, we provide evidence for the usability of the generated cost annotations as a mean to reason on the concrete complexity of programs written in C. For this purpose, we present a Frama-C plugin that uses our cost annotating compiler to automatically infer trustworthy logic assertions about the concrete worst case execution cost of programs written in a fragment of the C language. These logic assertions are synthetic in the sense that they characterize the cost of executing the entire program, not only constant-time fragments. (These bounds may depend on the size of the input data.) We report our experimentations on some C programs, especially programs generated by a compiler for the synchronous programming language Lustre used in critical embedded software.

1 Introduction

The formal description and certification of software components is reaching a certain level of maturity with impressing case studies ranging from compilers to kernels of operating systems. A well-documented example is the proof of functional correctness of a moderately optimizing compiler from a large subset of the C language to a typical assembly language of the kind used in embedded systems [11].

In the framework of the *Certified Complexity* (CerCo) project[1] [4], we aim to refine this line of work by focusing on the issue of the *execution cost* of

[1] CerCo project `http://cerco.cs.unibo.it`

M. Stoelinga and R. Pinger (Eds.): FMICS 2012, LNCS 7437, pp. 32–46, 2012.
© Springer-Verlag Berlin Heidelberg 2012

the compiled code. Specifically, we aim to build a formally verified C compiler that given a source program produces automatically a functionally equivalent object code plus an annotation of the source code which is a sound and precise description of the execution cost of the object code.

We target in particular the kind of C programs produced for embedded applications; these programs are eventually compiled to binaries executable on specific processors. The current state of the art in commercial products such as Scade[2] [8] is that the *reaction time* of the program is estimated by means of abstract interpretation methods (such as those developed by AbsInt[3] [7]) that operate on the binaries. These methods rely on a specific knowledge of the architecture of the processor and may require explicit (and uncertified) annotations of the binaries to determine the number of times a loop is iterated (see, *e.g.*, [14] for a survey of the state of the art).

In this context, our aim is to produce a mechanically verified compiler which can *lift* in a provably correct way the pieces of information on the execution cost of the binary code to cost annotations on the source C code. Then the produced cost annotations are manipulated with the Frama − C[4] [5] automatic tool to infer synthetic cost annotations. We stress that the practical relevance of the proposed approach depends on the possibility of obtaining accurate information on the execution cost of relatively short sequences of binary instructions. This seems beyond the scope of current Worst-Case Execution Time (WCET) tools such as AbsInt or Chronos[5] which do not support a *compositional* analysis of WCET. For this reason, we focus on processors with a simple architecture for which manufacturers can provide accurate information on the execution cost of the binary instructions. In particular, our experiments are based on the 8051 [10][6]. This is a widely popular 8-bits processor developed by Intel for use in embedded systems with no cache and no pipeline. An important characteristic of the processor is that its cost model is 'additive': the cost of a sequence of instructions is exactly the sum of the costs of each instruction.

The rest of the paper is organized as follows. Section 2 describes the labelling approach and its formal application to a toy compiler. The report [2] gives standard definitions for the toy compiler and sketches the proofs. A formal and browsable Coq development composed of 1 *Kloc* of specifications and 3.5 *Kloc* of proofs is available at `http://www.pps.univ-paris-diderot.fr/cerco`. Section 3 reports our experience in implementing and testing the labelling approach for a compiler from C to 8051 binaries. The CerCo compiler is composed of 30 *Kloc* of ocaml code; it can be both downloaded and tested as a web application at the URL above. More details are available in report [2] Section 4 introduces the automatic Cost tool that starting from the cost annotations produces certified synthetic cost bounds. This is a Frama − C plug-in composed of 5 *Kloc* of ocaml code also available at the URL above.

[2] Esterel Technologies. `http://www.esterel-technologies.com`

[3] AbsInt Angewandte Informatik. `http://www.absint.com/`

[4] Frama − C software analyzers. `http://frama-c.com/`

[5] Chronos tool. `www.comp.nus.edu.sg/~rpembed/chronos`

[6] The recently proposed ARM Cortex M series would be another obvious candidate.

2 A "Labelling" Method for Cost Annotating Compilation

In this section, we explain in general terms the so-called "labelling" method to turn a compiler into a cost annotating compiler while minimizing the impact of this extension on the proof of the semantic preservation. Then to make our purpose technically precise, we apply the method to a toy compiler.

2.1 Overview

As a first step, we need a clear and flexible picture of: (i) the meaning of cost annotations, (ii) the method to provide them being sound and precise, and (iii) the way such proofs can be composed. The execution cost of the source programs we are interested in depends on their control structure. Typically, the source programs are composed of mutually recursive procedures and loops and their execution cost depends, up to some multiplicative constant, on the number of times procedure calls and loop iterations are performed. Producing a *cost annotation* of a source program amounts to:

- enrich the program with a collection of *global cost variables* to measure resource consumption (time, stack size, heap size,...)
- inject suitable code at some critical points (procedures, loops,...) to keep track of the execution cost.

Thus, producing a cost-annotation of a source program P amounts to build an *annotated program* $An(P)$ which behaves as P while self-monitoring its execution cost. In particular, if we do *not* observe the cost variables then we expect the annotated program $An(P)$ to be functionally equivalent to P. Notice that in the proposed approach an annotated program is a program in the source language. Therefore, the meaning of the cost annotations is automatically defined by the semantics of the source language and tools developed to reason on the source programs can be directly applied to the annotated programs too. Finally, notice that the annotated program $An(P)$ is *only* meant to *reason* on the execution cost of the unannotated program P and it will never be compiled or executed.

Soundness and precision of cost annotations. Suppose we have a functionally correct compiler $\mathcal{C}$ that associates with a program P in the source language a program $\mathcal{C}(P)$ in the object language. Further suppose we have some obvious way of defining the execution cost of an object code. For instance, we have a good estimate of the number of cycles needed for the execution of each instruction of the object code. Now, the annotation of the source program $An(P)$ is *sound* if its prediction of the execution cost is an upper bound for the 'real' execution cost. Moreover, we say that the annotation is *precise* with respect to the cost model if the *difference* between the predicted and real execution costs is bounded by a constant which only depends on the program.

Compositionality. In order to master the complexity of the compilation process (and its verification), the compilation function $\mathcal{C}$ must be regarded as the result of the composition of a certain number of program transformations $\mathcal{C} = \mathcal{C}_k \circ \cdots \circ \mathcal{C}_1$. When building a system of cost annotations on top of an existing compiler, a certain number of problems arise. First, the estimated cost of executing a piece of source code is determined only at the *end* of the compilation process. Thus, while we are used to define the compilation functions $\mathcal{C}_i$ in increasing order, the annotation function An is the result of a progressive abstraction from the object to the source code. Second, we must be able to foresee in the source language the looping and branching points of the object code. Missing a loop may lead to unsound cost annotations while missing a branching point may lead to rough cost predictions. This means that we must have a rather good idea of the way the source code will eventually be compiled to object code. Third, the definition of the annotation of the source code depends heavily on *contextual information.* For instance, the cost of the compiled code associated with a simple expression such as $x+1$ will depend on the place in the memory hierarchy where the variable x is allocated. A previous experience described in [1] suggests that the process of pushing 'hidden parameters' in the definitions of cost annotations and of manipulating directly numerical cost is error prone and produces complex proofs. For this reason, we advocate next a 'labelling approach' where costs are handled at an abstract level and numerical values are produced at the very end of the construction.

2.2 The Labelling Approach, Formally

The 'labelling' approach to the problem of building cost annotations is summarized in the following diagram.

$$\begin{tikzcd}
L_1 & L_{1,\ell} \arrow[l, "\mathcal{I}"] \arrow[r, "\mathcal{C}_1"] \arrow[d, bend left, "er_1"] & L_{2,\ell} \arrow[d, "er_2"] & \cdots \arrow[r, "\mathcal{C}_k"] & L_{k+1,\ell} \arrow[d, "er_{k+1}"] \\
 & L_1 \arrow[r, "\mathcal{C}_1"] \arrow[u, bend left, "\mathcal{L}"] & L_2 & \cdots \arrow[r, "\mathcal{C}_k"] & L_{k+1}
\end{tikzcd}$$

$$\begin{aligned}
er_{i+1} \circ \mathcal{C}_i &= \mathcal{C}_i \circ er_i \\
er_1 \circ \mathcal{L} &= id_{L_1} \\
An &= \mathcal{I} \circ \mathcal{L}
\end{aligned}$$

For each language L_i considered in the compilation process, we define an extended *labelled* language $L_{i,\ell}$ and an extended operational semantics. The labels are used to mark certain points of the control. The semantics makes sure that whenever we cross a labelled control point a labelled and observable transition is produced.

For each labelled language there is an obvious function er_i erasing all labels and producing a program in the corresponding unlabelled language. The compilation functions $\mathcal{C}_i$ are extended from the unlabelled to the labelled language so that they enjoy commutation with the erasure functions. Moreover, we lift the soundness properties of the compilation functions from the unlabelled to the labelled languages and transition systems.

A *labelling* $\mathcal{L}$ of the source language L_1 is a function such that $er_{L_1} \circ \mathcal{L}$ is the identity function. An *instrumentation* $\mathcal{I}$ of the source labelled language $L_{1,\ell}$ is a function replacing the labels with suitable increments of, say, a fresh global *cost* variable. Then, an *annotation* An of the source program can be derived simply as the composition of the labelling and the instrumentation functions: $An = \mathcal{I} \circ \mathcal{L}$.

Suppose s is some adequate representation of the state of a program. Let P be a source program. The judgement $(P, s) \Downarrow s'$ is the big-step evaluation of P transforming state s into a state s'. Let us write $s[v/x]$ to denote a state s in which the variable x is assigned a value v. Suppose now that its annotation satisfies the following property:

$$(An(P), s[c/cost]) \Downarrow s'[c + \delta/cost] \tag{1}$$

where c and δ are some non-negative numbers. Then, the definition of the instrumentation and the fact that the soundness proofs of the compilation functions have been lifted to the labelled languages allows to conclude that

$$(\mathcal{C}(\mathcal{L}(P)), s[c/cost]) \Downarrow (s'[c/cost], \lambda) \tag{2}$$

where $\mathcal{C} = \mathcal{C}_k \circ \cdots \circ \mathcal{C}_1$ and λ is a sequence (or a multi-set) of labels whose 'cost' corresponds to the number δ produced by the annotated program. Then, the commutation properties of erasure and compilation functions allows to conclude that the *erasure* of the compiled labelled code $er_{k+1}(\mathcal{C}(\mathcal{L}(P)))$ is actually equal to the compiled code $\mathcal{C}(P)$ we are interested in. Given this, the following question arises: under which conditions the sequence λ, *i.e.*, the increment δ, is a sound and possibly precise description of the execution cost of the object code?

To answer this question, we observe that the object code we are interested in is some kind of assembly code and its control flow can be easily represented as a control flow graph. The idea is then to perform two simple checks on the control flow graph. The first check is to verify that all loops go through a labelled node. If this is the case then we can associate a finite cost with every label and prove that the cost annotations are sound. The second check amounts to verify that all paths starting from a label have the same cost. If this check is successful then we can conclude that the cost annotations are precise.

2.3 A Toy Compiler

As a first case study, we apply the labelling approach to a *toy compiler*.

The syntax of the source, intermediate and target languages is given in Figure 1. The three languages considered can be shortly described as follows: Imp is a very simple imperative language with pure expressions, branching and looping commands, Vm is an assembly-like language enriched with a stack, and Mips is a Mips-like assembly language [9] with registers and main memory.

The semantics of Imp is defined over configurations (S, K, s) where S is a statement, K is a continuation and s is a state. A *continuation* K is a list of

Syntax for Imp

$$
\begin{array}{rl}
id ::= & x \parallel y \parallel \ldots \\
n ::= & 0 \parallel -1 \parallel +1 \parallel \ldots \\
v ::= & n \parallel \mathsf{true} \parallel \mathsf{false} \\
e ::= & id \parallel n \parallel e + e \\
b ::= & e < e \\
S ::= & \mathsf{skip} \parallel id := e \parallel S; S \\
\parallel & \mathsf{if}\ b\ \mathsf{then}\ S\ \mathsf{else}\ S \\
\parallel & \mathsf{while}\ b\ \mathsf{do}\ S \\
P ::= & \mathsf{prog}\ S
\end{array}
$$

Syntax for Vm

$$
\begin{array}{rl}
instr_{\mathsf{Vm}} ::= & \mathtt{cnst}(n) \parallel \mathtt{var}(n) \\
\parallel & \mathtt{setvar}(n) \parallel \mathtt{add} \\
\parallel & \mathtt{branch}(k) \parallel \mathtt{bge}(k) \parallel \mathtt{halt}
\end{array}
$$

Syntax for Mips

$$
\begin{array}{rl}
instr_{\mathsf{Mips}} ::= & \mathtt{loadi}\, R, n \parallel \mathtt{load}\, R, l \\
\parallel & \mathtt{store}\, R, l \parallel \mathtt{add}\, R, R, R \\
\parallel & \mathtt{branch}\, k \parallel \mathtt{bge}\, R, R, k \parallel \mathtt{halt}
\end{array}
$$

Fig. 1. Syntax definitions

commands which terminates with a special symbol halt. The semantics of Vm is defined over stack-based machine configurations $C \vdash (i, \sigma, s)$ where C is a program, i is a program counter, σ is a stack and s is a state. The semantics of Mips is defined over register-based machine configurations $C \vdash (i, m)$ where C is a program, i is a program counter and m is a machine memory (with registers and main memory).

The first compilation function $\mathcal{C}$ relies on the stack of the Vm language to implement expression evaluation while the second compilation function $\mathcal{C}'$ allocates (statically) the base of the stack in the registers and the rest in main memory. This is of course a naive strategy but it suffices to expose some of the problems that arise in defining a compositional approach. The formal definitions of these compilation functions $\mathcal{C}$ from Imp to Vm and $\mathcal{C}'$ from Vm to Mips are standard and thus eluded. (See report [2] for formal details about semantics and the compilation chain.)

Applying the labelling approach to this toy compiler results in the following diagram. The next sections aim at describing this diagram in details.

$$\begin{tikzcd}
\mathsf{Imp} & \mathsf{Imp}_\ell \arrow[l, "\mathcal{I}"'] \arrow[r, "\mathcal{C}"] \arrow[d, bend left, "er_{\mathsf{Imp}}"] & \mathsf{Vm}_\ell \arrow[r, "\mathcal{C}'"] \arrow[d, "er_{\mathsf{Vm}}"] & \mathsf{Mips}_\ell \arrow[d, "er_{\mathsf{Mips}}"] \\
 & \mathsf{Imp} \arrow[u, bend left, "\mathcal{L}"] \arrow[r, "\mathcal{C}"] & \mathsf{Vm} \arrow[r, "\mathcal{C}'"] & \mathsf{Mips}
\end{tikzcd}$$

$$
\begin{array}{rcl}
er_{\mathsf{Vm}} \circ \mathcal{C} & = & \mathcal{C} \circ er_{\mathsf{Imp}} \\
er_{\mathsf{Mips}} \circ \mathcal{C}' & = & \mathcal{C}' \circ er_{\mathsf{Vm}} \\
er_{\mathsf{Imp}} \circ \mathcal{L} & = & id_{\mathsf{Imp}} \\
An_{\mathsf{Imp}} & = & \mathcal{I} \circ \mathcal{L}
\end{array}
$$

2.4 Labelled languages: Syntax and Semantics

Syntax The syntax of Imp is extended so that statements can be labelled: $S ::= \ldots \parallel \ell : S$. A new instruction $\mathsf{emit}(\ell)$ (resp. $(\mathsf{emit}\ \ell)$) is introduced in the syntax of Vm (resp. Mips).

Semantics. The small step semantics of Imp statements is extended as described by the following rule.

$$(\ell : S, K, s) \xrightarrow{\ell} (S, K, s)$$

We denote with $\lambda, \lambda', \ldots$ finite sequences of labels. In particular, the empty sequence is written ϵ. We also identify an unlabelled transition with a transition labelled with ϵ. Then, the small step reduction relation we have defined on statements becomes a *labelled transition system.* We derive a *labelled* big-step semantics as follows: $(S, s) \Downarrow (s', \lambda)$ if $(S, \mathsf{halt}, s) \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_n} (\mathsf{skip}, \mathsf{halt}, s')$ and $\lambda = \lambda_1 \cdots \lambda_n$.

Following the same pattern, the small step semantics of Vm and Mips are turned into a labelled transition system as follows:

$$\begin{array}{ll} C \vdash (i, \sigma, s) \xrightarrow{\ell} (i+1, \sigma, s) & \text{if } C[i] = \mathsf{emit}(\ell)\ . \\ M \vdash (i, m) \xrightarrow{\ell} (i+1, m) & \text{if } M[i] = (\mathsf{emit}\ \ell)\ . \end{array}$$

The evaluation predicate for labelled Vm is defined as $(C, s) \Downarrow (s', \lambda)$ if $C \vdash (0, \epsilon, s) \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_n} (i, \epsilon, s')$, $\lambda = \lambda_1 \cdots \lambda_n$ and $C[i] = \mathsf{halt}$. The evaluation predicate for labelled Mips is defined as $(M, m) \Downarrow (m', \lambda)$ if $M \vdash (0, m) \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_n} (j, m')$, $\lambda = \lambda_1 \cdots \lambda_n$ and $M[j] = \mathsf{halt}$.

2.5 Erasure Functions

There is an obvious *erasure* function er_{Imp} from the labelled language to the unlabelled one which is the identity on expressions and boolean conditions, and traverses commands removing all labels.

The erasure function er_{Vm} amounts to remove from a Vm code C all the $\mathsf{emit}(\ell)$ instructions and recompute jumps accordingly. Specifically, let $n(C, i, j)$ be the number of emit instructions in the interval $[i, j]$. Then, assuming $C[i] = \mathsf{branch}(k)$ we replace the offset k with an offset k' determined as follows:

$$k' = \begin{cases} k - n(C, i, i+k) & \text{if } k \geq 0 \\ k + n(C, i+1+k, i) & \text{if } k < 0 \end{cases}$$

The *erasure function* er_{Mips} is also similar to the one of Vm as it amounts to remove from a Mips code all the $(\mathsf{emit}\ \ell)$ instructions and recompute jumps accordingly. The compilation function $\mathcal{C}'$ is extended to Vm_ℓ by simply translating $\mathsf{emit}(\ell)$ as $(\mathsf{emit}\ \ell)$:

$$\mathcal{C}'(i, C) = (\mathsf{emit}\ \ell) \text{ if } C[i] = \mathsf{emit}(\ell)$$

2.6 Compilation of Labelled Languages

The compilation function $\mathcal{C}$ is extended to Imp_ℓ by defining:

$$\mathcal{C}(\ell : b, k) = (\mathsf{emit}(\ell)) \cdot \mathcal{C}(b, k) \qquad \mathcal{C}(\ell : S) = (\mathsf{emit}(\ell)) \cdot \mathcal{C}(S)\ .$$

Proposition 1. *For all commands S in Imp_ℓ, we have that:*

(1) $er_{\mathsf{Vm}}(\mathcal{C}(S)) = \mathcal{C}(er_{\mathsf{Imp}}(S))$.

(2) *If* $(S, s) \Downarrow (s', \lambda)$ *then* $(\mathcal{C}(S), s) \Downarrow (s', \lambda)$.

The following proposition relates Vm_ℓ code and its compilation and it is similar to proposition 1. Here $m \Vdash \sigma, s$ means "the low-level Mips memory m realizes the Vm stack σ and state s".

Proposition 2. *Let C be a Vm_ℓ code. Then:*

(1) $er_{\mathsf{Mips}}(\mathcal{C}'(C)) = \mathcal{C}'(er_{\mathsf{Vm}}(C))$.

(2) *If* $(C, s) \Downarrow (s', \lambda)$ *and* $m \Vdash \epsilon, s$ *then* $(\mathcal{C}'(C), m) \Downarrow (m', \lambda)$ *and* $m' \Vdash \epsilon, s'$.

2.7 Labellings and Instrumentations

Assuming a function κ which associates an integer number with labels and a distinct variable *cost* which does not occur in the program P under consideration, we abbreviate with $inc(\ell)$ the assignment $cost := cost + \kappa(\ell)$. Then we define the instrumentation $\mathcal{I}$ (relative to κ and *cost*) as follows:

$$\mathcal{I}(\ell : S) = inc(\ell); \mathcal{I}(S) \ .$$

The function $\mathcal{I}$ just distributes over the other operators of the language. We extend the function κ on labels to sequences of labels by defining $\kappa(\ell_1, \ldots, \ell_n) = \kappa(\ell_1) + \cdots + \kappa(\ell_n)$. The instrumented Imp program relates to the labelled one as follows.

Proposition 3. *Let S be an Imp_ℓ command. If* $(\mathcal{I}(S), s[c/cost]) \Downarrow s'[c+\delta/cost]$ *then* $\exists\, \lambda \;\; \kappa(\lambda) = \delta$ *and* $(S, s[c/cost]) \Downarrow (s'[c/cost], \lambda)$.

Definition 1. *A* labelling *is a function $\mathcal{L}$ from an unlabelled language to the corresponding labelled one such that $er_{\mathsf{Imp}} \circ \mathcal{L}$ is the identity function on the* Imp *language.*

Proposition 4. *For any labelling function $\mathcal{L}$, and* Imp *program P, the following holds:*

$$er_{\mathsf{Mips}}(\mathcal{C}'(\mathcal{C}(\mathcal{L}(P))) = \mathcal{C}'(\mathcal{C}(P)) \ . \tag{3}$$

Proposition 5. *Given a function κ for the labels and a labelling function $\mathcal{L}$, for all programs P of the source language if* $(\mathcal{I}(\mathcal{L}(P)), s[c/cost]) \Downarrow s'[c+\delta/cost]$ *and* $m \Vdash \epsilon, s[c/cost]$ *then* $(\mathcal{C}'(\mathcal{C}(\mathcal{L}(P))), m) \Downarrow (m', \lambda)$, $m' \Vdash \epsilon, s'[c/cost]$ *and* $\kappa(\lambda) = \delta$.

2.8 Sound and Precise Labellings

With any Mips_ℓ code M, we can associate a directed and rooted (control flow) graph whose nodes are the instruction positions $\{0, \ldots, |M| - 1\}$, whose root is the node 0, and whose directed edges correspond to the possible transitions between instructions. We say that a node is labelled if it corresponds to an instruction $\mathsf{emit}\ \ell$.

Definition 2. *A* simple path *in a* Mips_ℓ *code* M *is a directed finite path in the graph associated with* M *where the first node is labelled, the last node is the predecessor of either a labelled node or a leaf, and all the other nodes are unlabelled.*

Definition 3. *A* Mips_ℓ *code* M *is* soundly labelled *if in the associated graph the root node* 0 *is labelled and there are no loops that do not go through a labelled node. Besides, we say that a soundly labelled code is* precise *if for every label* ℓ *in the code, the simple paths starting from a node labelled with* ℓ *have the same cost.*

In a soundly labelled graph there are finitely many simple paths. Thus, given a soundly labelled Mips code M, we can associate with every label ℓ a number $\kappa(\ell)$ which is the maximum (estimated) cost of executing a simple path whose first node is labelled with ℓ. Thus for a soundly labelled Mips code the sequence of labels associated with a computation is a significant information on the execution cost.

For an example of command which is not soundly labelled, consider ℓ : $\mathsf{while}\ 0 < x\ \mathsf{do}\ x := x + 1$, which when compiled, produces a loop that does not go through any label. On the other hand, for an example of a program which is not precisely labelled consider $\ell : (\mathsf{if}\ 0 < x\ \mathsf{then}\ x := x + 1\ \mathsf{else\ skip})$. In the compiled code, we find two simple paths associated with the label ℓ whose cost will be quite different in general.

Proposition 6. *If* M *is soundly (resp. precisely) labelled and* $(M, m) \Downarrow (m', \lambda)$ *then the cost of the computation is bounded by* $\kappa(\lambda)$ *(resp. is exactly* $\kappa(\lambda)$*).*

The next point we have to check is that there are labelling functions (of the source code) such that the compilation function does produce sound and possibly precise labelled Mips code. To discuss this point, we introduce in table 1 a labelling function $\mathcal{L}_p$ for the Imp language. This function relies on a function "*new*" which is meant to return fresh labels and on an auxiliary function $\mathcal{L'}_p$ which returns a labelled command and a binary directive $d \in \{0, 1\}$. If $d = 1$ then the command that follows (if any) must be labelled.

Table 1. A labelling for the Imp language

$$\begin{array}{ll}
\mathcal{L}_p(\mathsf{prog}\ S) & = \mathsf{prog}\ \mathcal{L}_p(S) \\
\mathcal{L}_p(S) & = \mathit{let}\ \ell = \mathit{new},\ (S', d) = \mathcal{L'}_p(S)\ \mathit{in}\ \ell : S' \\
\mathcal{L'}_p(S) & = (S, 0) \quad \text{if } S = \mathsf{skip} \text{ or } S = (x := e) \\
\mathcal{L'}_p(\mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2) & = (\mathsf{if}\ b\ \mathsf{then}\ \mathcal{L}_p(S_1)\ \mathsf{else}\ \mathcal{L}_p(S_2), 1) \\
\mathcal{L'}_p(\mathsf{while}\ b\ \mathsf{do}\ S) & = (\mathsf{while}\ b\ \mathsf{do}\ \mathcal{L}_p(S), 1) \\
\mathcal{L'}_p(S_1; S_2) & = \mathit{let}\ (S'_1, d_1) = \mathcal{L'}_p(S_1),\ (S'_2, d_2) = \mathcal{L'}_p(S_2)\ \mathit{in} \\
 & \quad\ \mathit{case}\ d_1 \\
 & \quad\ 0 : (S'_1; S'_2, d_2) \\
 & \quad\ 1 : \mathit{let}\ \ell = \mathit{new}\ \mathit{in}\ (S'_1; \ell : S'_2, d_2)
\end{array}$$

Proposition 7. *For all* Imp *programs P, $\mathcal{C}'(\mathcal{C}(\mathcal{L}_p(P))$ is a soundly and precisely labelled* Mips *code.*

Once a sound and possibly precise labelling $\mathcal{L}$ has been designed, we can determine the cost of each label and define an instrumentation $\mathcal{I}$ whose composition with $\mathcal{L}$ will produce the desired cost annotation.

Definition 4. *Given a labelling function $\mathcal{L}$ for the source language* Imp *and a program P in the* Imp *language, we define an annotation for the source program as follows:*

$$An_{\mathsf{Imp}}(P) = \mathcal{I}(\mathcal{L}(P)) \ .$$

Proposition 8. *If P is a program and $\mathcal{C}'(\mathcal{C}(\mathcal{L}(P)))$ is a sound (sound and precise) labelling then $(An_{\mathsf{Imp}}(P), s[c/cost]) \Downarrow s'[c+\delta/cost]$ and $m \Vdash \epsilon, s[c/cost]$ entails that $(\mathcal{C}'(\mathcal{C}(P)), m) \Downarrow m'$, $m' \Vdash \epsilon, s'[c/cost]$ and the cost of the execution is bounded by (is exactly) δ.*

3 A C Compiler Producing Cost Annotations

We now consider an untrusted C compiler prototype in ocaml in order to experiment with the scalability of our approach. Its architecture is described below:

C → Clight → Cminor → RTLAbs (front end)
↓
Mips or 8051 ← LIN ← LTL ← ERTL ← RTL (back-end)

The most notable difference with CompCert [11] is that we target the Intel 8051 [10] and Mips assembly languages (rather than PowerPc). The compilation from C to Clight relies on the CIL front-end [13]. The one from Clight to RTL has been programmed from scratch and it is partly based on the Coq definitions available in the CompCert compiler. Finally, the back-end from RTL to Mips is based on a compiler developed in ocaml for pedagogical purposes[7]; we extended this back-end to target the Intel 8051. The main optimizations the back-end performs are liveness analysis and register allocation, and graph compression. We ran some benchmarks to ensure that our prototype implementation is realistic. The results are given in report [2].

This section informally describes the labelled extensions of the languages in the compilation chain (see report [2] for details), the way the labels are propagated by the compilation functions, and the (sound and precise) labelling of the source code. A related experiment concerning a higher-order functional language of the ML family is described in [3].

3.1 Labelled Languages

Both the Clight and Cminor languages are extended in the same way by labelling both statements and expressions (by comparison, in the toy language Imp we

[7] `http://www.enseignement.polytechnique.fr/informatique/INF564/`

just used labelled statements). The labelling of expressions aims to capture precisely their execution cost. Indeed, Clight and Cminor include expressions such as $a_1?a_2;a_3$ whose evaluation cost depends on the boolean value a_1. As both languages are extended in the same way, the extended compilation does nothing more than sending Clight labelled statements and expressions to those of Cminor.

The labelled versions of RTLAbs and the languages in the back-end simply consist in adding a new instruction whose semantics is to emit a label without modifying the state. For the CFG based languages (RTLAbs to LTL), this new instruction is emit *label* → *node*. For LIN, Mips and 8051, it is emit *label*. The translation of these label instructions is immediate.

3.2 Labelling of the Source Language

As for the toy compiler, the goals of a labelling are soundness, precision, and possibly economy. Our labelling for Clight resembles that of Imp for their common instructions (e.g. loops). We only consider the instructions of Clight that are not present in Imp[8].

Ternary expressions. They may introduce a branching in the control flow. We achieve precision by associating a label with each branch.

Program Labels and Gotos. Program labels and gotos are intraprocedural. Their only effect on the control flow is to potentially introduce an unguarded loop. This loop must contain at least one cost label in order to satisfy the soundness condition, which we ensure by adding a cost label right after each program label.

Function calls. In the general case, the address of the callee cannot be inferred statically. But in the compiled assembly code, we know for a fact that the callee ends with a return statement that transfers the control back to the instruction following the function call in the caller. As a result, we treat function calls according to the following invariants: (1) the instructions of a function are covered by the labels inside this function, (2) we assume a function call always returns and runs the instruction following the call. Invariant (1) entails in particular that each function must contain at least one label. Invariant (2) is of course an over-approximation of the program behavior as a function might fail to return because of an error or an infinite loop. In this case, the proposed labelling remains correct: it just assumes that the instructions following the function call will be executed, and takes their cost into consideration. The final computed cost is still an over-approximation of the actual cost.

4 A Tool for Reasoning on Cost Annotations

Frama − C is a set of analysers for C programs with a specification language called ACSL. New analyses can be dynamically added through a plug-in system.

[8] We do not consider expressions with side-effects because they are eliminated by CIL.

For instance, the Jessie plug-in allows deductive verification of C programs with respect to their specification in ACSL, with various provers as back-end tools.

We developed the Cost plug-in for the Frama – C platform as a proof of concept of an automatic environment exploiting the cost annotations produced by the CerCo compiler. It consists of an ocaml program of 5 *Kloc* which in first approximation takes the following actions: (1) it receives as input a C program, (2) it applies the CerCo compiler to produce a related C program with cost annotations, (3) it applies some heuristics to produce a tentative bound on the cost of executing the C functions of the program as a function of the value of their parameters, (4) the user can then call the Jessie tool to discharge the related proof obligations. In the following we elaborate on the soundness of the framework, the algorithms underlying the plug-in, and the experiments we performed with the Cost tool.

4.1 Soundness

The soundness of the whole framework depends on the cost annotations added by the CerCo compiler, the synthetic costs produced by the Cost plug-in, the verification conditions (VCs) generated by Jessie, and the external provers discharging the VCs. The synthetic costs being in ACSL format, Jessie can be used to verify them. Thus, even if the added synthetic costs are incorrect (relatively to the cost annotations), the process in its globality is still correct: indeed, Jessie will not validate incorrect costs and no conclusion can be made about the WCET of the program in this case. In other terms, the soundness does not really depend on the action of the Cost plug-in, which can in principle produce *any* synthetic cost. However, in order to be able to actually prove a WCET of a C function, we need to add correct annotations in a way that Jessie and subsequent automatic provers have enough information to deduce their validity. In practice this is not straightforward even for very simple programs composed of branching and assignments (no loops and no recursion) because a fine analysis of the VCs associated with branching may lead to a complexity blow up.

4.2 Inner Workings

The cost annotations added by the CerCo compiler take the form of C instructions that update by a constant a fresh global variable called the *cost variable*. Synthesizing a WCET of a C function thus consists in statically resolving an upper bound of the difference between the value of the cost variable before and after the execution of the function, i.e. find in the function the instructions that update the cost variable and establish the number of times they are passed through during the flow of execution. The plug-in proceeds as follows.

- Each function is independently processed and is associated a WCET that may depend on the cost of the other functions. This is done with a mix between abstract interpretation [6] and syntactic recognition of specific loops.

- As result of the previous step, a system of inequations is built and its solution is attempted by an iterative process. At each iteration, one replaces in all the inequations the references to the cost of a function by its associated cost if it is independent of the other functions. This step is repeated till a fixpoint is reached.
- ACSL annotations are added to the program according to the result of the above fixpoint. The two previous steps may fail in finding a concrete WCET for some functions, because of imprecision inherent in abstract interpretation, and because of recursive definitions in the source program not solved by the fixpoint. At each program point that requires an annotation (function definitions and loops), annotations are added if a solution was found for the program point.
- The most difficult instructions to handle are loops. We consider loops for which we can syntactically find a counter (its initial, increment and last values are domain dependent). Other loops are associated an undefined cost ($\top$). When encountering a loop, the analysis first sets the cost of its entry point to 0. The cost inside the loop is thus relative to the loop. Then, for each exit point, we fetch the value of the cost at that point and multiply it by an upper bound of the number of iterations (obtained through arithmetic over the initial, increment and last values of the counter); this results in an upper bound of the cost of the whole loop, which is sent to the successors of the considered exit point.

Figure 2 shows the action of the Cost plug-in on a C program. The most notable differences are the added so-called *cost variable*, some associated update (increment) instructions inside the code, and an `ensures` clause that specifies the WCET of the `is_sorted` function with respect to the cost variable. One can notice that this WCET depends on the inputs of the function. Running Jessie on the annotated and specified program generates VCs that are all proved by the automatic prover AltErgo[9].

4.3 Experiments

The Cost plug-in has been developed in order to validate CerCo's framework for modular WCET analysis. The plug-in allows (semi-)automatic generation and certification of WCET for C programs. Also, we designed a wrapper for supporting Lustre files. Indeed, Lustre is a data-flow language to program synchronous systems and the language comes with a compiler to C. The C function produced by the compiler implements the *step function* of the synchronous system and computing the WCET of the function amounts to obtain a bound on the reaction time of the system.

We tested the Cost plug-in and the Lustre wrapper on the C programs generated by the Lustre compiler. We also tested it on some basic algorithms and cryptographic functions; these examples, unlike those generated by the Lustre

[9] AltErgo prover. `http://ergo.lri.fr/`

```
int is_sorted (int *tab, int size) {
  int i, res = 1;
  for (i = 0 ; i < size-1 ; i++) if (tab[i] > tab[i+1]) res = 0;
  return res; }
```

(a) The initial C source code.

```
int _cost = 0;

/*@ ensures (_cost ≤ \old(_cost)+(101+(0<size-1?(size-1)*195:0))); */
int is_sorted (int *tab, int size) {
  int i, res = 1, _cost_tmp0;
  _cost += 97; _cost_tmp0 = _cost;
  /*@ loop invariant (0 < size-1) ⇒ (i ≤ size-1);
    @ loop invariant (0 ≥ size-1) ⇒ (i ≡ 0);
    @ loop invariant (_cost ≤ _cost_tmp0+i*195);
    @ loop variant (size-1)-i; */
  for (i = 0; i < size-1; i++) {
    _cost += 91;
    if (tab[i] > tab[i+1]) { _cost += 104; res = 0; }
    else _cost += 84; }
  _cost += 4; return res; }
```

(b) The annotated source code generated by Cost.

Fig. 2. An example of the Cost plug-in action

File	Type	Description	LOC	VCs
`3-way.c`	C	Three way block cipher	144	34
`a5.c`	C	A5 stream cipher, used in GSM cellular	226	18
`array_sum.c`	S	Sums the elements of an integer array	15	9
`fact.c`	S	Factorial function, imperative implementation	12	9
`is_sorted.c`	S	Sorting verification of an array	8	8
`LFSR.c`	C	32-bit linear-feedback shift register	47	3
`minus.c`	L	Two modes button	193	8
`mmb.c`	C	Modular multiplication-based block cipher	124	6
`parity.lus`	L	Parity bit of a boolean array	359	12
`random.c`	C	Random number generator	146	3
S: standard algorithm C: cryptographic function L: C generated from a Lustre file				

Fig. 3. Experiments on CerCo and the Cost plug-in

compiler include arrays and for-loops. Table 3 provides a list of concrete programs and describes their type, functionality, the number of lines of the source code, and the number of VCs generated. In each case, the Cost plug-in computes a WCET and AltErgo is able to discharge all VCs. Obviously the generation of synthetic costs is an undecidable and open-ended problem. Our experience just shows that there are classes of C programs which are relevant for embedded applications and for which the synthesis and verification tasks can be completely automatized.

Acknowledgement. The master students Kayvan Memarian and Ronan Saillard contributed both to the Coq proofs and the CerCo compiler in the early stages of their development.

References

1. Amadio, R.M., Ayache, N., Memarian, K., Saillard, R., Régis-Gianas, Y.: Compiler Design and Intermediate Languages. Deliverable 2.1 of [4]
2. Ayache, N., Amadio, R.M., Régis-Gianas, Y.: Certifying and reasoning on cost annotations of C programs. Research Report 00702665 (June 2012)
3. Amadio, R.M., Régis-Gianas, Y.: Certifying and Reasoning on Cost Annotations of Functional Programs. In: Peña, R., van Eekelen, M., Shkaravska, O. (eds.) FOPARA 2011. LNCS, vol. 7177, pp. 72–89. Springer, Heidelberg (2012)
4. Certified complexity (Project description). ICT-2007.8.0 FET Open, Grant 243881
5. Correnson, L., Cuoq, P., Kirchner, F., Prevosto, V., Puccetti, A., Signoles, J., Yakobowski, B.: Frama-C user manual. CEA-LIST, Software Safety Laboratory, Saclay, F-91191, `http://frama-c.com/`
6. Cousot, P., Cousot, R.: Abstract Interpretation Frameworks. Jou. of Logic and Computation 2(4), 511–547 (1992)
7. Ferdinand, C., Heckmann, R., Le Sergent, T., Lopes, D., Martin, B., Fornari, X., Martin, F.: Combining a high-level design tool for safety-critical systems with a tool for WCET analysis of executables. In: Embedded Real Time Software (2008)
8. Fornari, X.: Understanding how SCADE suite KCG generates safe C code. White paper, Esterel Technologies (2010)
9. Larus, J.: Assemblers, linkers, and the SPIM simulator. Appendix of Computer Organization and Design: the hw/sw interface. Hennessy and Patterson (2005)
10. MCS 51 Microcontroller Family User's Manual. Publication number 121517. Intel Corporation (1994)
11. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM 52(7), 107–115 (2009)
12. Leroy, X.: Mechanized semantics, with applications to program proof and compiler verification. In: Marktoberdorf Summer School (2009)
13. Necula, G., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002)
14. Wilhelm, R., et al.: The worst-case execution-time problem - overview of methods and survey of tools. ACM Trans. Embedded Comput. Syst. 7(3) (2008)