Abstract

The CerCo project aims at building a certified compiler for the C language that can lift in a provably correct way information on the execution cost of the object code to cost annotations on the source code. These annotations are added at specific program points (e.g. inside loops). In this article, we describe a plug-in of the Frama – C plateform that synthesizes the CerCo's cost annotations of a C function into an assertion of the WCET of the function. Then, we report our experimentations on C code generated from Lustre files.

1 Introduction

Estimating the worst case execution time (WCET) of an embedded software is an important task, especially in a critical system. The micro-controller running the system must be efficiently used: money and reaction time depend on it. However, computing the WCET of a program is undecidable in the general case, and static analysis tools dedicated to this task often fail when the program involves complicated loops, living few hopes for the user to obtain a result. In this article, we present experiments that validate the new approach introduced by the CerCo project[1] for WCET prediction. With CerCo, the user is provided raw and certified cost annotations. We designed a tool that uses these annotations to generate WCET bounds. When the tool fails, we show how the user can complete the required information, so that he is never stuck. The tool already provides good results, but it is yet one of the many possibilities on how to use CerCo's annotations.

CerCo. The goal of the CerCo project is to build a certified compiler for the C language that lifts in a provably correct way information on the execution cost of the object code to cost annotations on the source code. An untrusted compiler has been developed [2] that targets the 8051, a popular micro-controller typically used in embedded systems. The compiler relies on the *labelling approach* to compute the cost annotations: at the C level, specific program points — called *cost labels* — are identified in the control flow of the program. Each cost label is a symbolic value that represents the cost of the instructions following the label and before the next one. Then, the compilation keeps track of the association between program points and cost labels. In the end, a concrete cost is computed for each cost label from the object code, and the information is sent up to the C level for instrumentation. Figure 1a shows a C code, and figure 1b presents its transformation through CerCo.

As one notices, the result of CerCo is an instrumentation of the input C program:

- a global variable called `_cost` is added. Its role is to hold the cost information during execution;
- a `_cost_incr` function is defined; it will be used to update the cost information;
- finally, update instructions are inserted inside the functions of the program: those are the cost annotations. In the current state of the compiler, they represent the number of processor's cycles that will be spent executing the following instructions before the next annotation. But other kind of information could be computed using the labelling approach, such as stack size for instance.

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

```
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) before CerCo

```
int _cost = 0;

void _cost_incr (int incr) { _cost = _cost + incr; }

int is_sorted (int *tab, int size) {
  int i, res = 1;

  _cost_incr(97);

  for (i = 0; i < size-1; i++) {
    _cost_incr(91);
    if (tab[i] > tab[i+1]) { _cost_incr(104); res = 0; }
    else _cost_incr(84);
  }

  _cost_incr(4);
  return res;
}
```

(b) after CerCo

Figure 1: An example of CerCo's action

Frama − C. In order to deduce an upper bound of the WCET of a C function, we need a tool that can analyse C programs and relate the value of the `_cost` variable before and after the function is executed. We chose to use the Frama − C verification tool[2] for the following reasons:

- the platform allows all sorts of analyses in a modular and collaborative way: each analysis is a plug-in that can reuse the results of existing ones. The authors of Frama − C provide a development guide for writing new plug-ins. Thus, if existing plug-ins experience difficulties in synthesizing the WCET of C functions annotated with CerCo, we can define a new analysis dedicated to this task;
- it supports ACSL, an expressive specification language à la Hoare logic as C comments. Expressing WCET specification using ACSL is very easy;

[2] http://frama-c.com/

- the Jessie plug-in builds verification conditions (VCs) from a C program with ACSL annotations. The VCs can be sent to various provers, be they automatic or interactive. When they are discharged, the program is guaranteed to respect its specification.

Figure 2 shows the program of figure 1b with ACSL annotations added manually. The most important is the post-condition attached to the `is_sorted` function:

```
ensures _cost <= \old(_cost) + 101 + (size-1)*195;
```

It means that executing the function yields the value of the `_cost` variable to be incremented by at most `101 + (size-1)*195`: this is the WCET specification of the function. Running the Jessie plug-in on this program creates 8 VCs that an automatic prover such as Alt − Ergo is able to fully discharged, which proves that the WCET specification is indeed correct.

```
int _cost = 0;

/*@ ensures _cost == \old(_cost) + incr; */
void _cost_incr (int incr) { _cost = _cost + incr; }

/*@ requires size >= 1;
  @ ensures _cost <= \old(_cost) + 101 + (size-1)*195; */
int is_sorted (int *tab, int size) {
  int i, res = 1;

  _cost_incr(97);

  /*@ loop invariant i < size;
    @ loop invariant _cost <= \at(_cost, Pre) + 97 + i*195;
    @ loop variant size-i; */
  for (i = 0; i < size-1; i++) {
    _cost_incr(91);
    if (tab[i] > tab[i+1]) { _cost_incr(104); res = 0; }
    else _cost_incr(84);
  }

  _cost_incr(4);
  return res;
}
```

Figure 2: Annotations with ACSL

Contributions. **(*** TODO ***)** Validate the benefits of CerCo.

Outline. In the remaining of this article, we present a Frama − C plug-in called Cost that adds a WCET specification to the functions of a CerCo-annotated C program. Section 2 briefly details the inner workings of the plug-in and discusses its soundness. Section 3 compares our approach with other WCET tools. Section 4 presents a case study for the plug-in on the Lustre synchronous language. Section 5 shows some benchmarks on typical embedded C programs and C programs originated from Lustre files. Finally, section 6 concludes.

2 The Cost plug-in

The Cost plug-in for the Frama − C platform has been developed in order to automatically synthesize the cost annotations added by the CerCo compiler on a C source program into assertions of the WCET of the functions in the program. We start our description of the plug-in by discussing the soundness of the framework, because, as we will see, the action of the plug-in is not involved in this issue. Then, the details of the plug-in will be presented.

2.1 Soundness

The architecture of the plug-in is depicted in figure 3. The plug-in has a C source file for parameter and creates a new C file that is the former with additional cost annotations (C code) and WCET assertions (ACSL annotations). First, the input file is fed to Frama − C that will in turn send it to the Cost plug-in. The action of the plug-in is:

1. apply the CerCo compiler to the source file;
2. synthesize an upper bound of the WCET of each function of the source program by reading the cost annotations added by CerCo;
3. add the results in the form of post-conditions in ACSL format, relating the cost of the function before and after its execution.

Then, the user can either *trust* the results (the WCET of the functions), or want to *verify* them, in which case he can call Jessie.

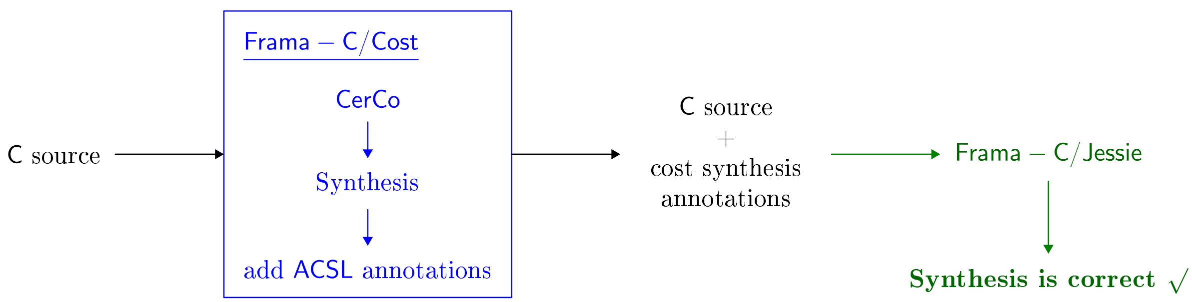

Figure 3: the Cost plug-in

Since the Cost plug-in adds annotations in ACSL format, Jessie (or other verification plug-ins for Frama − C) can be used to verify these annotations. Thus, even if the added annotations are incorrect, the process in its globality is still correct: indeed, Jessie will not validate incorrect annotations and no conclusion can be made about the WCET of the program in this case. This means that the Cost plug-in can add *any* annotation for the WCET of a function, the framework will still be correct. But in order to be able to actually prove a WCET of a function, we need to add correct annotations in a way that Jessie and subsequent automatic provers have enough information to deduce validity.

2.2 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. This raises two main issues: indecidability caused by loop constructs, and function calls. Indeed, a side effect of function calls is to change the value of the cost variable. When a function calls another one, the cost of the callee is part of the cost of the caller. This means that the computation of a WCET of each function of a C program is subject to the calling dependencies. To cope with the issues of loops and function calls, the Cost plug-in proceeds as follows:

- each function is independently processed and associated a WCET that may depend on the cost of the other functions. This is done with a mix between abstract interpretation [4] and syntactic recognition of specific loops for which we can decide the number of iterations. The abstract domain used is made of expressions whose variables can only be formal parameters of the function;

- a system of inequations is built from the result of the previous step, and is tried to be solved with a fixpoint. At each iteration, the fixpoint 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;

- ACSL annotations are added to the program according to the results of the above fixpoint. Note that the two previous steps may fail in finding a concrete WCET for some functions, because of imprecision inherent to abstract interpretation, and recursion 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.

Figure 4 shows the result of the Cost plug-in when fed the program in figure 1a. There are several differences from the manually annotated program, the most noticeable being:

- the manually annotated program had a pre-condition that the `size` formal parameter needed to be greater or equal to 1. The Cost plug-in does not make such an assumption, but instead considers both the case where `size` is greater or equal to 1, and the case where it is not. This results in a ternary expression inside the WCET specification (the post-condition or `ensures` clause), and some new loop invariants;

- the loop invariant specifying the value of the cost variable depending on the iteration number refers to a new local variable named `_cost_tmp0`. It represents the value of the cost variable right before the loop is executed. It allows to express the cost inside the loop with regards to the cost before the loop, instead of the cost at the beginning of the function; it often makes the expression a lot shorter and eases the work for nested loops.

Running Jessie on the program generates VCs that are all proved by Alt − Ergo: the WCET computed by the Cost plug-in is correct.

```
int _cost = 0;

/*@ ensures _cost ≡ \old(_cost) + incr; */
void _cost_incr (int incr) { _cost = _cost + incr; }

/*@ 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_incr(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_incr(91);
    if (tab[i] > tab[i+1]) { _cost_incr(104); res = 0; }
    else _cost_incr(84);
  }

  _cost_incr(4);
  return res;
}
```

Figure 4: Result of the Cost plug-in

3 Related work

There exist a lot of tools for WCET analysis. Yet, the framework encompassing the Cost plug-in is the only one, to our knowledge, that enjoys the following features:

- The results of the plug-in have a very high level of trust. First, because the cost annotations added by CerCo are proven correct (this is on-going research in the Matita system). Second, because verification with Jessie is deductive and VCs can be discharged with various provers. The more provers discharge a VC, the more trustful is the result. When automatic provers fail in discharging a VC, the user can still try to verify them manually, with an interactive theorem prover such as Coq[3] that Jessie outputs to.

- While other WCET tools act as black boxes, the Cost plug-in provides the user with as many information as it can. When a WCET tool fails, the user generally have few hopes, if any, of understanding and resolving the issue in order to obtain a result. When the Cost plug-in fails to add an annotation, the user can still try to complete it. And

[3]http://coq.inria.fr/

since the results of CerCo is C code, it is much easier to understand the behavior of the annotations.

- The results of the Cost plug-in being added to the source C file, it allows to easily identify the cost of parts of the code and the cost of the functions of the program. The user can modify parts that are too costly and observe their precise influence on the overall cost.
- The framework is modular: the Cost plug-in is yet one possible synthesis, and Jessie is one possible back-end for verification. We can use other synthesis strategies, and choose for each result the one that seems the most precise. The same goes for Jessie: we can use the WP plug-in of Frama − C instead, and even merge the results of both.

4 Lustre case study

Lustre is a synchronous language where reactive systems are described by flow of values. It comes with a compiler that transforms a Lustre node (any part of or the whole system) into a C *step* function that represents one synchronous cycle of the node. A WCET for the step function is thus a worst case reaction time for the component. The generated C step function neither contains loops nor is recursive, which makes it particularly well suited for a use with the Cost plug-in with a complete support.

We designed a wrapper that has for inputs a Lustre file and a node inside the file, and outputs the cost of the C step function corresponding to the node. Optionally, verification with Jessie or testing can be toggled. The flow of the wrapper is described in figure 5. It simply executes a command line, reads the results, and sends them to the next command.

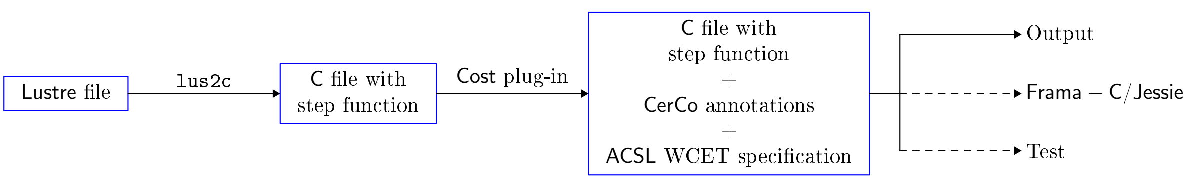

Figure 5: Flow of the Lustre wrapper

A typical run of the wrapper looks as follows (we use the `parity` example from our distribution of Lustre; it computes the parity bit of a boolean array):

```
frama-c_lustre -verify -test parity.lus parity
```

Invoking the above command line produces the following output:

```
WCET of parity_step:  2220+_cost_of_parity_0_parity+_cost_of_parity_0_done
(not verified).
Verifying the result (this may take some time)...
WCET is proven correct.
Testing the result (this may take some time)...
Estimated WCET: 2220
Minimum:  2144
Maximum:  2220
Average:  2151
Estimated WCET is correct for these executions.
```

- All the intermediary results of the wrapper are stored in files. Verbosity can be turned on to show the different commands invoked and the resulting files.

- The step function generated with the Lustre compiler for the node `parity` is called `parity_step`. It might call functions that are not defined but only prototyped, such as `parity_O_parity` or `parity_O_done`. Those are functions that the user of the Lustre compiler can use for debugging, but that are not part of the `parity` system. Therefore, we leave their cost abstract in the expression of the cost of the step function, and we set their cost to 0 when testing (this can be changed by the user).

- Testing consists in adding a `main` function to the C file, that will run the step function on a parameterized number of input states for a parameterized number of cycles. The C file contains information that allows to syntatically distinguish integer variables used as booleans, which helps in generating interesting input states. After each iteration of the step function, the value of the cost variable is fetched in order to compute its overall minimum, maximum and average value for one step. If the maximum were to be greater than the WCET computed by the Cost plug-in, then we could conclude of an error in the plug-in.

5 Experiments

(* TODO ***)** remember: validate the benefits of CerCo. loc

6 Conclusion

We have described a plug-in for Frama − C that relies on the CerCo compiler to automatically synthesize a WCET for C programs. The soundness of the overall process is guaranteed through the Jessie plug-in. Finally, we successfully used the plug-in on C programs generated from Lustre files; the result is an automatically computed and certified reaction time for the Lustre nodes. **(*** TODO ***)**: CerCo's validation.

References

[1] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: Design and implementation of a cost and termination analyzer for java bytecode. In *Formal Methods for Components and Objects, 6th International Symposium, FMCO 2007, Amsterdam, The Netherlands, October 24-26, 2007, Revised Lectures*, volume 5382 of *Lecture Notes in Computer Science*, pages 113–132. Springer, 2008.

[2] R. M. Amadio, N. Ayache, and Y. Régis-Gianas. Deliverable 2.2: Prototype implementation. Technical report, ICT Programme, Feb. 2011. Project FP7-ICT-2009-C-243881 CerCo - Certified Complexity.

[3] R. M. Amadio, N. Ayache, Y. Régis-Gianas, K. Memarian, and R. Saillard. Deliverable 2.1: Compiler design and intermediate languages. Technical report, ICT Programme, July 2010. Project FP7-ICT-2009-C-243881 CerCo - Certified Complexity.

[4] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[5] R. Heckmann, C. Ferdinand, A. Angewandte, and I. Gmbh. Worst-case execution time prediction by static program analysis. In *In 18th International Parallel and Distributed Processing Symposium (IPDPS 2004*, pages 26–30. IEEE Computer Society.