A Formally Verified WCET Estimation Tool

André Maroneze[1], Sandrine Blazy[1], David Pichardie[2], and Isabelle Puaut[1]

1 **IRISA – Université Rennes 1**
Campus Universitaire de Beaulieu, Rennes, France
andre.maroneze@irisa.fr, sandrine.blazy@irisa.fr, isabelle.puaut@irisa.fr

2 **IRISA – ENS Rennes**
Campus Universitaire de Ker Lann, Bruz, France
david.pichardie@irisa.fr

Abstract

The application of formal methods in the development of safety-critical embedded software is recommended in order to provide strong guarantees about the absence of software errors. In this context, WCET estimation tools constitute an important element to be formally verified. We present a formally verified WCET estimation tool, integrated to the formally verified CompCert C compiler. Our tool comes with a machine-checked proof which ensures that its WCET estimates are safe. Our tool operates over C programs and is composed of two main parts, a loop bound estimation and an Implicit Path Enumeration Technique (IPET)-based WCET calculation method. We evaluated the precision of the WCET estimates on a reference benchmark and obtained results which are competitive with state-of-the-art WCET estimation techniques.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases Formal Verification, CompCert C Compiler, WCET Estimation

Digital Object Identifier 10.4230/OASIcs.WCET.2014.11

1 Introduction

In the context of safety-critical embedded software, international regulations such as the DO-178C standard promote the use of formal methods for software development. Among them, formal verification provides guarantees about the specification and the implementation of a program through the use of machine-checked proofs. Instead of relying on a manual verification effort, the use of tools provides stronger guarantees about the absence of errors in the proof. This is especially important for reasoning on real languages such as C.

Safety-critical systems are an instance of real-time systems, where programs must respect given timing constraints. An important measure in real-time systems is the worst-case execution time of a program. Obtaining a *safe* WCET estimate (that is, a value at least as large as the actual WCET) is part of the necessary guarantees for such systems.

Current WCET estimation tools, even when based on sound static analysis techniques, are not verified. This may lead to bugs being accidentally introduced in the implementation. The main contribution of this paper is a formally verified WCET estimation tool operating over C code. It extends previous work on formally verified static analyses ([5] and [4]) by adding our WCET estimation, based on the classic IPET technique. In our approach, the code of our tool is automatically generated from its formal specification. Furthermore, machine-checked proofs ensure the estimated WCET is at least as large as the actual WCET.

Our formally verified WCET estimation tool has been integrated into CompCert [10], a moderately optimizing, formally verified C compiler usable for critical software. This integration provides two major benefits for our tool: first, it allows us to reuse CompCert's formal specifications, including formal semantics for C and assembly (CompCert targets PowerPC, ARM and x86 assembly). Second, it enables the integration of analyses at different intermediate languages (e.g. high-level loop transformations and low-level WCET estimation). It also allows us to benefit from CompCert's optimizations (such as constant propagation) to improve the precision of our WCET estimates.

In our formal development, we verified two major components of our WCET estimation tool: a loop bound estimation technique (inspired by one of SWEET's [7] techniques for loop bound estimation), based on program slicing and a value analysis, and the generation of an integer linear programming (ILP) system for WCET estimation via IPET. Combining these techniques results in a WCET estimation tool together with a machine-checked proof that the produced WCET estimate is safe. We evaluated the precision and efficiency of our implementation on the Mälardalen WCET benchmarks [8]. Our results are competitive with state-of-the-art WCET estimation techniques.

This paper is structured as follows: in Section 2 we present the architecture of our WCET estimation tool, focusing on the techniques and implementation. In Section 3, we detail the proof architecture, presenting a high-level view of the final correctness theorem in our formalization and its main components. In Section 4, we describe the experimental evaluation and its results. We present some related work in Section 5 and then conclude in Section 6.

2 Architecture of our WCET estimation tool

We developed our WCET estimation tool within CompCert, which is equipped with a correctness theorem, that is, a proof relating the behaviors of source and compiled code, which ensures that no bugs are introduced during compilation. CompCert has several intermediate languages, among them RTL (for *Register Transfer Language*), where a program is represented by its control-flow graph (CFG). Our loop bound estimation is performed at this level. The WCET calculation phase is defined at the lowest level (assembly), closer to the executable for more precision. The result of our analyses performed in RTL is transported to assembly thanks to the correctness theorem.

Figure 1 illustrates the architecture of our WCET estimation tool (bottom row) within the CompCert compilation chain (top row). We start by presenting the loop bound estimation (Section 2.1). Due to space considerations, we only present an overview of this technique, referring the reader to [5] for more details.

The other main component in our tool is the WCET calculation step, which generates an ILP system to produce the WCET estimate (Section 2.2). It uses the result of the loop bound estimation. Finally, we present some (optional) loop transformations (Section 2.3), used to improve the precision of our WCET estimate. They are performed on structured loops, during front-end compilation.

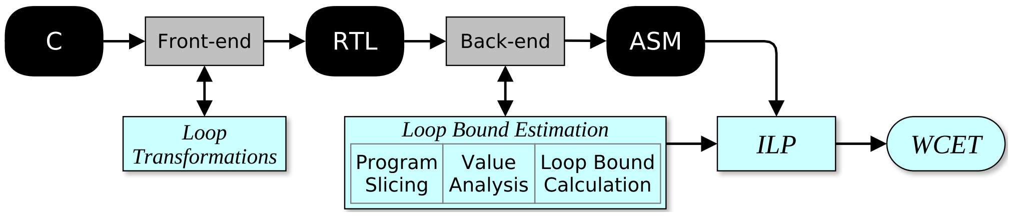

Figure 1 CompCert's compilation chain (top) and our WCET estimation tool (bottom).

2.1 Loop bound estimation

Our loop bound estimation technique, inspired by one of those used in SWEET [7], is composed of three parts: program slicing, value analysis, and loop bound calculation. In a deterministic and terminating program execution, the same program state cannot occur twice, i.e. the values of the program variables are unique at each iteration. Thus, we count the number of different states and use it as an upper bound for the number of loop iterations.

For each loop in the program, the first step consists in performing program slicing by removing statements not related to the loop header, which is our slicing criterion. This improves the precision and speed of the following passes. For instance, in the program below, where the loop header is the loop exit condition (`i < 5`), program slicing removes statements which do not affect the value of variable `i` (considering `f` free of side-effects).

```
i = 1;
a[0] = 0;
while (i < 5) {
    a[i] = a[i-1] + f(i);
    i++;
}
```

```
i = 1;

while (i < 5) {

    i++;
}
```

Then, our value analysis computes, for each program variable at each program point, an interval containing all possible values of that variable, as indicated in the code fragment below. This interval is a safe over-approximation, and therefore when counting the number of different possibilities, we will obtain a safe estimate of all possible values. This analysis is based on abstract interpretation, and it is detailed in [4]. An example program is presented below, with the result of our value analysis in the right column.

`i = 0; j = 0;`	$i \in [0,0]$, $j \in [0,0]$
`while (i < 5)`	$i \in [0,5]$, $j \in [0,2]$
`{`	$i \in [0,4]$, $j \in [0,2]$
`    i++;`	$i \in [1,5]$, $j \in [0,2]$
`    if (i == 5 && j < 2) {`	$i \in [5,5]$, $j \in [0,1]$
`      i = 0;`	$i \in [0,0]$, $j \in [0,1]$
`      j++;`	$i \in [0,0]$, $j \in [1,2]$
`    }`	$i \in [0,5]$, $j \in [0,2]$
`}`	$i \in [5,5]$, $j \in [2,2]$

The final stage consists in computing the product of the size of the domains of the *relevant variables* for each loop – variables which are *live*, *used* and *modified* inside the loop. Only these variables may influence the number of loop iterations.

The computed product is a safe loop bound estimation. For instance, in the previous example there are two relevant variables, `i` and `j`. The number of iterations is safely bounded by the product of the sizes of their intervals at the loop exit condition ($[0,5]$ and $[0,2]$), that is, $6 \times 3 = 18$ iterations (the exact bound here is 15 iterations). This method also works for nested loops, by computing the product of outer and inner loop bounds.

2.2 IPET-based WCET estimation

We apply a classic technique, namely IPET [11], to produce an ILP system representing the program's execution flow. The objective function to maximize, representing the program execution time, is $T = \sum_{i \in code} t_i x_i$. For each program point i in the program, x_i is a static over-approximation of the number of times i is executed, and t_i is the cost coefficient (in cycles) associated to this program point. In our tool, we currently use a simple cost model where $t_i = 1$ for every instruction. Our ILP constraints are obtained from the reconstructed CFG at the assembly level. The ILP also incorporates the loop bounds previously computed, as constraints of the form $x_h \leq N$, where h is a loop header and N is the inferred loop bound. RTL bounds are correctly transported to assembly thanks to CompCert's semantic preservation theorem.

2.3 Loop transformations

To improve the precision of our loop bound estimations, we apply two kinds of loop transformations: *loop inversion* and *loop unrolling*. Loop inversion consists in converting `while` and `for` loops into `do-while` loops. The motivation behind this transformation is the fact that each kind of loop behaves differently with respect to loop iterations (`while` and `for` loops execute the loop exit condition more often than the loop body). A cost-effective way to deal with the variety of C constructs is to reduce them to a few general cases and treat them uniformly, which is done by loop inversion. The code fragment below illustrates the application of loop inversion to a simple `while` loop.

```
while (i < 5) {
    f(i);
    i++;
}
```

```
if (i < 5) {
    do {
        f(i);
        i++;
    } while (i < 5);
}
```

Loop unrolling is used to improve the precision of the WCET estimate for loops containing conditional branches with different execution costs, for instance in a loop where the first iteration performs differently from the others. Without loop unrolling, the cost of the longest branch is considered in each iteration and results in a WCET overestimation. The code below illustrates an example where loop unrolling helps to improve precision. The values inside the `/*...*/` comments indicate statically known values which will be optimized by a constant propagation pass. For instance, the second call to the `init()` function in the unrolled loop below is unreachable, and therefore eliminated after code simplification, as indicated in the last column.

```
i = 0;
do {
    if (i == 0) init();
    i++;
} while (i < 2);
```

```
i = 0;
do {
    if (i /*0*/ == 0) init();
    i++;
    if (i /*1*/ >= 2) break;
    if (i /*1*/ == 0) init();
    i++;
} while (i /*2*/ < 2);
```

```
i = 0;
do {
    init();
    i++;
    //if (1 >= 2) break;
    //if (1 == 0) init();
    i++;
} while (i < 2);
```

To avoid excessive unrolling, we only unroll loops with conditional branches (which can help improve the WCET), and limit the unrolling factor according to the size of the code.

3 Proof of our WCET estimation tool

Our tool has been specified and formally proved correct using the Coq [6] proof assistant. With Coq's functional specification and programming language, we proceeded as follows: first, we specified our functions; then, we defined logical properties about these specifications; afterwards, we proved these properties using Coq's interactive proof mechanism, where we write the proof step-by-step while Coq checks its correctness; finally, we used the automatic code generation mechanism available in Coq to obtain our verified tool directly from its specification. In the end, we obtained an executable software (our WCET estimation tool) plus its *proof of correctness*.

The proof of correctness is a proof of semantic preservation. In this section, we define more precisely our notion of semantic preservation and then we detail the proof architecture of the main components of the tool. We present *what* has been proved correct, with an intuitive notion of the main correctness lemmas, and briefly mention the proof techniques, i.e. *how* it has been proved.

3.1 Correctness theorem of the WCET estimation

To define the correctness of a WCET estimation algorithm, we need to define the WCET itself and then what is a WCET estimate and how to compute it. We do so from a formal semantics of the CompCert assembly language. We present here some notions necessary for understanding our proof sketches.

In our semantics, a *program state* contains the value of each memory variable and machine register at a given point in the program execution. The semantics defines the evolution of program states. A well-formed sequence of program states forms an *execution trace*. We denote $\mathit{Terminates}(P, \mathit{tr})$ as the complete execution of program P, producing the execution trace tr.

We extended the CompCert assembly semantics to take into account the quantitative aspect of the WCET, adding *execution counters* for every program point and program transition (CFG vertex and edge, respectively). These counters correspond to the number of occurrences of the program point (or program transition) in a given execution trace. They represent the *exact* values obtained during execution along that trace.

The execution time T of an execution trace tr is defined as *the sum of the execution counters of every program point* (since local costs are considered as 1). The *worst-case* execution time of a program P is thus defined as the maximum execution time among all possible program executions. More formally, we can define the WCET as follows. Let $\mathit{TR}(P)$ be the set of all possible traces of program P, that is, $\mathit{TR}(P) = \{\mathit{tr} \mid \mathit{Terminates}(P, \mathit{tr})\}$. Then:

$$\mathit{WCET}(P) = \max_{\mathit{tr} \in \mathit{TR}(P)} T(\mathit{tr})$$

Note that this is only a mathematical definition: neither our algorithm nor our proof actually enumerates all program paths.

To perform the WCET estimation, we *over-approximate* the execution counters using the x_i variables of the ILP system. Our WCET estimate, WCET_E, is the sum of all x_i variables. For a WCET estimation tool to be considered *sound*, all estimates it produces must be larger than or equal to the actual WCET. This can be stated as follows: *for any terminating program, every estimate t_E produced by the tool must be an over-approximation of the actual WCET.*

▶ **Theorem 1** (Correctness of the WCET Estimation).

$$\forall P, \forall \mathit{tr}, \mathit{Terminates}(P, \mathit{tr}) \wedge \mathit{WCET}_E(P) = \lfloor t_E \rfloor \implies \mathit{WCET}(P) \leq t_E$$

WCET_E is the actual WCET estimation (partial) function, defined as the composition of all stages of our WCET estimation tool. A successful estimation is denoted by $\lfloor t_E \rfloor$. The executable code for WCET_E is automatically obtained from its formal specification.

3.2 Proof techniques

In formal verification, the standard approach consists in formally specifying and proving every concept and algorithm, once and for all. For instance, this means formally verifying an ILP solver to prove the correctness of the WCET calculation phase, for every program. However, the formal verification of an ILP solver is an endeavor which is out of our approach. In such situations, there is an alternative proof technique which provides strong guarantees about correctness, called *a posteriori validation* [14]. It consists in checking the *result* of a computation (using a *validator*) without proving each step of its construction.

More specifically, we used *verified validation*, which includes a proof of correctness of the validator itself. This proof ensures that any result accepted by the validator is indeed correct. This technique is already used in CompCert, for instance during register allocation.

The major advantages of a posteriori validation are (1) manageable proof effort (especially for algorithms relying on sophisticated heuristics, whose proof might be otherwise too costly) and (2) the possibility to integrate untrusted code (such as an off-the-shelf ILP solver, instead of having to prove the solver itself). The trade-off is that validation may incur some extra computation time during program execution. However, in our case validators were used in situations where their cost was negligible with respect to the computation time of the solution itself (such as during ILP computation).

3.3 Correctness proof of the loop bound estimation

To deal with all kinds of loops (such as unstructured loops created by `goto` statements), the correctness theorem of our loop bound estimation is defined in terms of arbitrary program points. In other words, we prove that the execution counters of a given set of program points are bounded by our estimation technique. In practical terms, these program points correspond to *loop headers*, which entails that their loops are effectively bounded.

Since the loop bound estimation is composed of three parts, we defined a correctness theorem for each of them and combined the proofs using theorems that already exist in CompCert. The idea behind the correctness of each of these intermediate theorems is presented in the following.

Program slicing

Informally, the correctness of program slicing is stated as: *the bounds computed in the sliced program are a safe overestimation of the bounds computed in the original program.* In other words, we can transform a program P, obtaining a program slice P', compute bounds on the latter, and safely transpose these bounds to the original program.

More formally, and as an example of a Coq theorem, we present below a simplified version of the correctness theorem of program slicing. Let `P` be an RTL program and `slice(P,i)` its sliced version with respect to program point `i`. Let `tr` and `tr'` be valid execution traces of `P` and `slice(P,i)`, respectively. Then, any bound `B` which is correct for the counter of `i` in `P'` (that is, `counter(tr', i)` $\leq$ `B`) is also a correct bound in the original program.

```
Theorem slicing_correctness:
   forall (P : program) (i : program_point) (tr tr' : exec_trace),
     Terminates(P, tr) ∧ Terminates(slice(P,i), tr') ⟹
   forall (B : int),
     counter(tr', i) ≤ B
     ⟹ counter(tr, i) ≤ B.
```

Program slicing is a transformation that preserves the semantics with respect to some slicing criterion. This property is classically proved by induction on the execution relation between the original and transformed programs. It amounts to showing that a step-by-step parallel execution of both programs results in the same execution counters for the slicing criterion (i.e. the loop header) in both programs.

To compute a slice efficiently, sophisticated data structures (such as postdominator trees and program dependence graphs) are necessary. To avoid formalizing all of them, we developed an untrusted program slicer and validated its result, obtaining the same guarantees while enabling the adoption of more efficient slicers without having to change the proof. Detailed information about the program slicer and its validation are described in [5].

Value analysis

An intuitive notion of the formal correctness of the value analysis can be stated as follows: *for any interval given by the value analysis, all possible variable values are taken into account.* This is true for each program point and each program variable. To prove it correct, we first show that each individual execution step (given by the assembly formal semantics) is correct in itself. For instance, after a CFG branch merge, an interval union of the values of each branch is a correct over-approximation of the values after the merge.

The major difficulties in proving the value analysis come from issues related to the complexity of the C language (such as having a large number of operators) and the efficiency of the analysis. In the presence of loops, the analysis needs to perform several iterations before it reaches a final (stable) solution. We show that each operator is correctly abstracted into an interval, and then we show that the final solution is stable with respect to loops. Both facts entail that the solution is a correct over-approximation.

Loop bound calculation

The proof of the loop bound calculation relates the sizes of the intervals of variable values to the execution counters in the extended semantics. It is proved by induction on the execution trace: every time an execution step would allow a program point to exceed its bounds, this would lead to an infinite loop. The bounds, defined with respect to the result of the value analysis, contain every possible value for all variables which influence the loop condition. For instance, if variables i and j have bounds $[0, 1]$, and they are the only variables influencing the loop exit condition, then these pairs can appear at most once in the execution trace: $(0, 0)$, $(0, 1)$, $(1, 0)$ and $(1, 1)$. Reaching the loop exit condition a fifth time (e.g. with $(i, j) = (1, 1)$) would imply an infinite repetition of the same sub trace, leading to an unbounded execution time. The same reasoning is extended to handle nested loops.

3.4 Correctness proof of the ILP

As mentioned in Section 3.2, we use *a posteriori* validation to guarantee a correct ILP result. Our verification is decomposed in three parts, as indicated in Figure 2. The first part consists in proving that the ILP generation is correct. The second part is the ILP solving, performed by an external component. The final part is the validation of the solution.

Proving the ILP correct means showing that each ILP variable is an over-approximation of its corresponding execution counter. Using the flow constraints (including loop bounds), the proof of correctness is based on reasoning by induction on the execution trace.

The generated ILP is sent to an external ILP solver (such as lpsolve [2]), which returns an assignment of variable values and the corresponding WCET estimate. We prove two

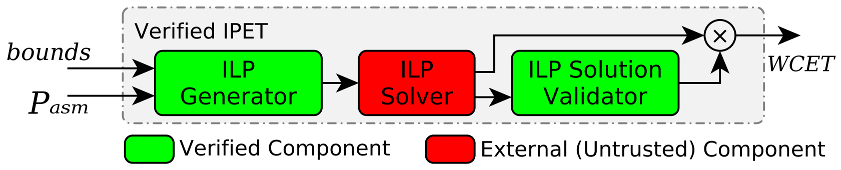

Figure 2 Diagram of the components used for the IPET verification.

properties about the result: that (1) it is a solution of the system (a valid assignment), and that (2) it is the *largest* solution (i.e. the worst-case).

Verifying property (1) only requires substituting variables with their assigned values and checking that all constraints are respected. To verify property (2), we show that any larger solution is infeasible. To do so, we augment the system with the negation of the solution (i.e. we add $\sum x_i > t_E$, where t_E is the solution given by the solver) and then we compute a *Farkas certificate* [3], a set of linear coefficients which can be used to prove the infeasibility of a linear system. This computation amounts to the solution of a system of the same size.

To integrate this technique within the proved framework, we define and prove a verified validator in Coq, whose inputs are the ILP system, its (untrusted) solution and a certificate, and whose output is *true* if the certificate confirms that the solution is valid. The correctness theorem of the validator ensures that, if the inputs pass validation, then the solution is a correct WCET estimate. With this final step, we prove the theorem stated in Section 3.1.

3.5 Feedback on the proof effort

Our proof effort resulted in over 15,000 lines of Coq code (half of them being Coq definitions, and the other half being Coq proofs). The development also contains about 2,000 lines of manually written OCaml code (which includes code such as the program slicer), and about the same size of code automatically generated from the Coq development.

Concerning the development methodology, we followed the standard practice in formal verification, which consists in performing the specification and the proof in parallel. While performing the proof, several details about the specification need to be reformulated, either to improve their clarity, or to enable the proof to go through (e.g. there are several ways to specify a program slicer, but only a few of them lead to efficient proof strategies).

4 Experimental evaluation

We evaluated our WCET estimation tool on the Mälardalen WCET benchmarks. We considered 15 of the 20 programs evaluated in [5]. The other 5 programs (`adpcm`, `fft1`, `fir`, `insertsort` and `ludcmp`) contain loops which were not bounded due to imprecisions in the value analysis, such as loops depending on floating-point variables or on memory contents.

We compiled the code to PowerPC assembly to estimate its WCET. To evaluate the precision of our WCET estimation, we modified the programs having several possible execution paths (e.g. by setting specific values to input variables) to ensure execution of a worst-case path. We executed them using the formal semantics to obtain the exact WCET, and then we compared this value to the estimate obtained on the original program.

Figure 3 presents our evaluation. For each program, we indicate the size of its source code (LoC) and we present the relative WCET overestimation, using 3 different configurations: no loop transformations, loop inversion only, and loop inversion together with loop unrolling. We also present the analysis times of our tool.

		No Loop Transformations		*Loop Inversion*		*Inversion+Unrolling*		
Program	***LoC***	***Overestimation***	***Time (s)***	***Overestimation***	***Time (s)***	***Overestimation***	***Time (s)***	***Class***
cnt	267	18.3%	0.1	2.8%	0.2	3.3%	7.0	OK
cover	640	10.9%	1.0	11.5%	1.0	0.0%	21.8	OK
crc	128	100.2%	0.2	99.5%	0.2	99.2%	1.7	Imprecise
edn	285	141.5%	12.5	110.4%	13.1	110.4%	23.4	Imprecise
expint	157	2601.6%	0.0	2419.7%	0.0	0.0%	8.2	OK
fdct	239	0.0%	0.4	0.0%	0.5	0.0%	0.6	OK
fibcall	72	0.9%	0.0	1.1%	0.0	1.1%	0.0	OK
jfdctint	375	0.0%	0.3	0.0%	0.3	0.0%	0.5	OK
lcdnum	64	50.9%	0.0	55.2%	0.0	11.9%	0.1	OK
matmult	163	11.5%	0.3	0.0%	0.3	0.0%	0.5	OK
ndes	231	12.2%	4.0	3.6%	4.2	3.6%	225.4	OK
ns	535	88.3%	0.1	0.2%	0.1	0.2%	0.2	OK
nsichneu	4,253	106.1%	60.5	106.1%	60.2	106.3%	89.7	Imprecise
qurt	166	168.2%	0.7	165.7%	0.7	215.2%	3.0	Imprecise
ud	161	225.1%	0.6	217.3%	0.6	265.2%	11.3	Imprecise

Figure 3 Experimental results of our WCET tool, given as a relative overestimation w.r.t. the exact WCET, without and then with loop transformations.

We classify the programs in two groups: OK (WCET estimate with no or small overestimation) and Imprecise (significant overestimation). Comparing the different configurations confirms that the loop transformations improved the precision of the WCET estimate, sometimes drastically (e.g. `ns` goes from 88% overestimation down to 0%, thanks to loop inversion, and `expint` goes from 2420% to 0% due to loop unrolling). In a few programs, we see a *relative* increase, which is due to the decrease in the *absolute* WCET of the transformed loops. Overall, the loop optimizations provide a significant benefit in terms of precision.

5 Related work

There are several WCET estimation tools in the literature which perform loop bound estimation, such as aiT [9], Bound-T [16], oRange [13], SWEET [7] and TuBound [15]. Our objective is not to develop a novel technique to estimate the WCET, but to formally specify an existing method and to prove it correct.

Due to its extensive flow analysis, SWEET was the inspiration for our loop bound estimation. SWEET has two loop bound estimation techniques: one similar to the one we formally verified, and another one having a context-sensitive mechanism capable of inferring more precise flow constraints. The trade-off is that, in some cases, it may perform excessive unrolling and not terminate. Unlike our tool, SWEET is not formally verified.

WCC [12] is a C compiler integrating an external WCET estimation tool. WCC performs automatic loop bound estimation based on polyhedral evaluation, but it focuses on hardware-level optimizations, while we focus on the control-flow analysis. WCC is not formally verified and it relies on other tools (such as aiT) to obtain WCET estimates. Our tool, on the other hand, only relies on external tools if they can provide certificates. This ensures that bugs in their implementation do not affect the correctness of our tool.

The CerCo [1] project shares our views about the necessity of formal guarantees in WCET estimation, but it has a different approach and objective. In CerCo, an original technique transports annotations from the assembly to the C source program, and then it relies on a non-verified tool, based on program proof (Frama-C), to produce proof certificates about the correctness of the WCET estimates. Unlike ours, this approach is not entirely automatic: after analyzing a program, some verification conditions may need manual proof. Concerning the hardware model, cost information is based on a simple timing model, like our tool.

6 Conclusion

We presented a formally verified WCET estimation tool whose correctness theorem ensures its WCET estimates are safe. Our tool is built within the CompCert C compiler, providing extra guarantees about the execution time of the code produced by the compiler. There are now two complementary tools which operate over C code for safety-critical embedded systems, and these tools are formally verified, which provides unprecedented guarantees.

Our tool relies on the formal verification of different techniques commonly used by industrial-strength WCET estimation tools: loop bound estimation and IPET. Experimental evaluation of the precision of our tool indicates satisfactory results. Future work includes improving the precision of current analyses and adding a more realistic hardware model.

References

1 R. Amadio, A. Asperti, N. Ayache, B. Campbell, D. P. Mulligan, R. Pollack, Y. Régis-Gianas, C. S. Coen, and I. Stark. Certified complexity. *Procedia CS*, 7:175–177, 2011.

2 M. Berkelaar, K. Eikland, and P. Notebaert. lpsolve : Open source (Mixed-Integer) Linear Programming system. `http://lpsolve.sourceforge.net`.

3 F. Besson, T. Jensen, D. Pichardie, and T. Turpin. Certified result checking for polyhedral analysis of bytecode programs. In *TGC*, pages 253–267. Springer, 2010.

4 S. Blazy, V. Laporte, A. Maroneze, and D. Pichardie. Formal verification of a C value analysis based on abstract interpretation. In *SAS*, LNCS, pages 324–344. Springer, 2013.

5 S. Blazy, A. Maroneze, and D. Pichardie. Formal verification of loop bound estimation for WCET analysis. In *VSTTE 2013*, LNCS. Springer, 2013.

6 Coq development team. The Coq proof assistant. `http://coq.inria.fr`, 1989–2014.

7 A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In *WCET*, 2007.

8 J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET benchmarks: Past, present and future. In *WCET*, pages 137–147, 2010.

9 R. Heckmann and C. Ferdinand. aiT: worst case execution time prediction by static program analysis. In *IFIP Congress Topical Sessions*, pages 377–384, 2004.

10 X. Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, 2009.

11 Y. T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. *IEEE Trans. on CADICS*, 16(12):1477–1487, 1997.

12 P. Lokuciejewski and P. Marwedel. *Worst-Case Execution Time Aware Compilation Techniques for Real-Time Systems*. Springer, 2011.

13 M. de Michiel, A. Bonenfant, H. Cassé, and P. Sainrat. Static loop bound analysis of C programs based on flow analysis and abstract interpretation. In *Proc. of ERTSS*, pages 161–166. IEEE Computer Society, 2008.

14 G. Necula. Translation validation for an optimizing compiler. In *PLDI*, pages 83–94. ACM, 2000.

15 A. Prantl, M. Schordan, and J. Knoop. TuBound – A Conceptually New Tool for Worst-Case Execution Time Analysis. In *WCET*, 2008.

16 Tidorum. Bound-T tool homepage. `http://www.bound-t.com`, 2010.