Formal Verification of Loop Bound Estimation for WCET Analysis *

Sandrine Blazy[1], Andre Maroneze[1], and David Pichardie[2]

[1] IRISA - Université Rennes 1
[2] Harvard University / INRIA

Abstract. Worst-case execution time (WCET) estimation tools are complex pieces of software performing tasks such as computation on control flow graphs (CFGs) and bound calculation. In this paper, we present a formal verification (in Coq) of a loop bound estimation. It relies on program slicing and bound calculation.
The work has been integrated into the CompCert verified C compiler. Our verified analyses directly operate on non-structured CFGs. We extend the CompCert RTL intermediate language with a notion of loop nesting (a.k.a. weak topological ordering on CFGs) that is useful for reasoning on CFGs. The automatic extraction of our loop bound estimation into OCaml yields a program with competitive results, obtained from experiments on a reference benchmark for WCET bound estimation tools.

1 Introduction

Avionics embedded software is developed according to international regulations. Among them is the DO-178C, that has been published in 2012, thirty years after its previous version DO-178B [19]. The DO-178C promotes the use of formal methods for developing real-time safety-critical software rigorously. Airplane manufacturers also follow their own development standards, and formal methods were already used by the Airbus airplane manufacturer for developing safety-critical software during DO-178B.

In this context, Airbus conducted experiments (see [5]) in order to compile in a realistic environment an up-to-date flight control software with CompCert, a formally verified compiler [16]. The CompCert compiler is a formally verified optimizing compiler for the C language that has been specified, implemented and proved using the Coq proof assistant. The compiler is exempt from miscompilation issues: it is equipped with a proof of semantic preservation. This proof is done once for all in Coq; it states that every compiled program behaves as prescribed by the semantics of its source program.

Even if formal methods are promoted by avionics standards, adopting a formally verified compiler is not self-evident in an industrial context. The quality of

* This work was supported by Agence Nationale de la Recherche, grant number ANR-11-INSE-003 Verasco.

the code generated by the compiler is as important as the formal guarantees. For real-time safety-critical software, a common practice is to measure this quality by counting the size of the compiled code and by estimating its worst-case execution time (WCET) [5]. Estimating WCET is a crucial step when developing real-time software. It ensures that no run of a program will exceed its allowed execution time. Computing the exact WCET of any program is not always possible and simulations or static analyses are required to estimate it.

WCET estimation tools are complex pieces of software performing three main tasks related to 1) control flow facts, 2) hardware features (e.g. cache misses) and 3) estimate calculation; see [23] for a survey of techniques and tools. Sound estimate calculation computes an upper bound of all execution times of a whole program (i.e. a global bound) from the flow and timing information obtained by the first two tasks (i.e. from local bounds). This paper focuses on the first and third tasks: control flow facts that are useful for estimating loop bounds. A loop bound is a static over-approximation of the number of times a loop is executed during any execution of a given program. Estimating the execution time of instructions on a given hardware is still an active field of research in the WCET community and is out of the scope of this paper.

There are many studies on loop bound estimation in the literature. The techniques range from pattern-matching (for identifying simple loop patterns), modeling computations using affine equalities and inequalities (that are solved by a decision procedure for Presburger arithmetic), data flow analysis, symbolic execution to abstract interpretation. Basic techniques handle only simple loops; advanced techniques handle various forms of nested loops. Some of these static analysis techniques are well understood for several years now but their implementations in a real toolchain are still error prone, because these implementations operate directly on unstructured CFGs originating from C programs. We focus here on the SWEET loop bound analysis technique [9] that demonstrated a good precision in the context of WCET analysis.

Compiling Airbus flight control software with CompCert has shown that the quality of the compiled code is better than the quality of the compiler currently used at Airbus [5]. The next step towards an industrial use of CompCert is to qualify it according to DO-178C, and to strengthen the confidence in the results of tools such as WCET estimation tools. In that perspective, combining the CompCert verified compiler with a formal verification of a loop bound analysis estimation for WCET analysis is valuable.

Our work is significant for many reasons.

- It constitutes the first machine-checked proof of a nontrivial loop bound estimation algorithm operating over an intermediate language having the same expressiveness as C. This proof combines two proof techniques, whole formal verification using the Coq proof assistant and formal verification of untrusted checkers.
- It provides a reference implementation of a tool combining independent techniques: loop reconstruction in an unstructured CFG, program slicing and loop bound calculation. Program slicing is required to improve the precision

of the analysis, by removing irrelevant variables that do not impact on the number of iterations of a loop.
– A tool has been generated automatically from our formalization. Its performances are close to those of reference tools for estimating loop bounds. In this paper, we compare our tool with a reference tool called SWEET that also relies on program slicing [13]. Our tool has been integrated into the CompCert compiler, thus enabling the transmission of loop bound annotations to other WCET tools.

All results presented in this paper have been mechanically verified using the Coq proof assistant. The complete Coq development is available online at the following URL: `http://www.irisa.fr/celtique/ext/loopbound`. Consequently, the paper only sketches the proofs of some of its results; the reader is referred to the Coq development for the full proofs.

The remainder of this paper is organized as follows. First, Section 2 introduces our loop bound estimation. Then, Section 3 defines an abstract notion of loop nesting. Section 4 explains the formal verification of program slicing. Section 5 is devoted to the formal verification of the loop bound calculation. Section 6 describes the experimental evaluation of our implementation. Related work is discussed in Section 7, followed by concluding remarks.

2 A Loop Bound Estimation for WCET Analysis

First, this section gives an overview of our loop bound estimation and explains informally its key features. The loop bound estimation operates over a language that is introduced in the second part of this section. Then, the main theorems stating the soundness of our loop bound estimation are explained.

2.1 Overview

Fig. 1 shows the user's view of our analysis. The CompCert compiler consists in many intermediate languages and passes. It provides a general mechanism to attach annotations to program points. Annotations are transported throughout compilation, all the way to the generated assembly code [16]. Our loop bound analysis computes bounds on the RTL intermediate representation and attaches them to these annotations. Moreover, thanks to the semantic preservation of the CompCert compiler, we obtain semantic guarantees about these bounds in terms of the semantics of the assembly code generated by the compiler: each annotation in the assembly code is attached with a provably correct bound.

Classically, bounding a loop consists in estimating by static analysis how many times at most the loop will be executed. In our setting, the estimation of the loop bound is calculated by approximating the variation of the sizes of the domains of some selected variables (we call them *interesting* variables), that influence the loop bound estimation. If the loop is not nested into another loop, the estimation of the loop bound is the product of all the sizes of the domains

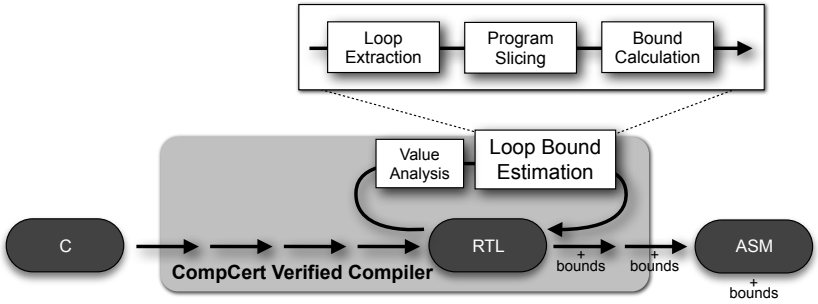

Fig. 1. Main architecture of our loop bound analysis.

related to interesting variables. If all variables are considered as interesting, then we may obtain a bad over-approximation of the loop bound. If some interesting variables are forgotten, then we obtain an incorrect approximation. When the loop is nested, the local bounds of all the loop bounds involved in the nesting are estimated separately as if there was no nesting, and the global estimation of the innermost loop combines in a product the estimations of local bounds.

There are several challenges for estimating a loop bound.

1. The loop structure of the program must be reconstructed from the unstructured graph representation of the program. Efficient loop extraction algorithms have been developed for graphs but directly reasoning on them in a semantic proof is challenging.
2. An analysis is required to select the interesting variables. It is performed in two steps: program slicing and computation of locally modified variables in loop bodies. First, the program is sliced w.r.t. each loop condition, as described in [9]. There are as many slices as loops and each slice is an executable program. Secondly, interesting variables are selected among the variables belonging to the slice. Due to nested loops, a computation is performed to select the interesting variables of the current nested loop. Given such a loop L, the interesting variables of L are live variables at the entry of L that may be both modified and used in the body of L. This computation is simpler than program slicing, but complementary to program slicing. It can be seen as a slicing of the program restricted to one of its nested loops.
3. A final calculation is required to take into account nested loops and collect all the local estimations of bounds involved in nestings.

A last challenge is related to the value analysis that is required to estimate at any program point the valuation of all program variables. A value analysis is usually based on abstract interpretation and uses widening and narrowing operators to speed up fixpoint resolution. The formal verification of a value analysis based on abstract interpretation and operating over a real-world language raises many challenging verification problems that are detailed in [7].

For the purpose of illustrating our approach, a succinct example program `P` is presented in Fig. 2, extracted from a LU decomposition algorithm. In `P`, there are 2 annotations, written as 2 calls to a specific built-in function and used here to mark loop entries.[3] They are attached to program points 4 and 6 and will be transported throughout compilation. The CompCert compiler will place the comment "loop1" (resp. "loop2") at the exact program point corresponding to the program point 4 (resp. 6) in the assembly code. The right part of the figure shows the CFG of the program and its loop nestings that will be presented in Section 3. First, the program `P` at the left of the figure is sliced twice as there are two loops in `P`. The third (resp. fourth) column shows the slice w.r.t. the first (resp. second) loop of `P`.

The first slice consists of the statements that contribute to the number of executions reaching program point 4. It includes the variables used in the loop exit condition (i.e. at program point 15). Intuitively, we slice w.r.t. a loop condition, but we could also slice w.r.t. any other program point of the slice. To facilitate our proofs (i.e. the reasoning on graphs), we choose to slice w.r.t. loop headers (i.e. loop entries, see Section 3) and show that this amounts to slicing w.r.t. loop conditions. It is then easy to bound the loop of the first slice. At program point 4, the value analysis states that the values of `n` and `i` belong to respectively $[5; 5]$ (size 1) and $[0; 5]$ (size 6). Thus, the condition of this loop is evaluated $1 * 6 = 6$ times and the bound of this loop is estimated to 6. This result is written as a comment in the corresponding loop, for illustration purposes.

The second slice of Fig. 2 is related to the loop entry at program point 6 (i.e. the second loop of P) and includes variables `j` and `n` used in the loop exit condition at program point 12. Because `j` is defined at program point 5, in the slice, the second loop is still nested in the first one. Among the variables in the second slice, only `j` is an interesting variable (only `j` is modified in the loop body). The local bound of the second loop is 6, that is estimated from the second slice as the size of the domain of `j` (that is the size of $[0; 5]$, the interval estimated by the value analysis). Note that if the value of `i` was modified in the second loop, then the local bound would have been estimated as the product between 6 and the size of the domain of `i`.

The last step is the calculation of the global bound of the innermost loop, from the local bounds. The most widely used technique consists in translating the CFG (and some extra information about the control flow) into an ILP (i.e. integer linear program) [23]. The goal function of the ILP solver expresses the total execution time to be maximized. Here, we do not rely on an ILP solver but we simply compute the product of both previous local bounds which may over-approximate the exact bound in some cases. In Fig. 2, the global bound is estimated to the exact bound $6 * 6 = 36$.

This example shows that program slicing is complementary to the computation of modified variables. For instance, only the program slicing can eliminate all statements related to `w`, since `w` would have been considered as an interesting variable if it had not been sliced. Moreover, the computation of modified vari-

[3] Annotations can be written manually by the user or generated by an untrusted tool.

	Program P	Slice of P w.r.t. 1st loop (at 4)	Slice of P w.r.t. 2nd loop (at 6)
1	`n = 5;`	`n = 5;`	`n = 5;`
2	`i = 0;`	`i = 0;`	`i = 0;`
3	`w = 0.0;`		
4	`do { _annot("loop1");`	`do { _annot("loop1");`	`do { _annot("loop1");`
5	`  j = 0;`	`  /* bound=6 */`	`  j = 0;`
6	`  do { _annot("loop2");`		`  do { _annot("loop2");`
7	`    a[i][j] = i+1+j+1;`		`    /* bound=6 */`
8	`    if (i == j)`		
9	`      a[i][j] *= 5.0;`		
10	`    w += a[i][j];`		
11	`    j++;`		`    j++;`
12	`  } while (j <= n);`		`  } while (j <= n);`
13	`  b[i] = w;`		
14	`  i++;`	`  i++;`	`  i++;`
15	`} while (i <= n);`	`} while (i <= n);`	`} while (i <= n);`

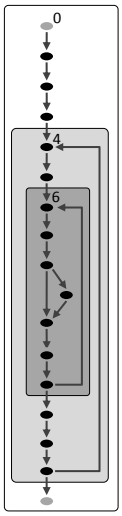

Fig. 2. An example program, its computed slices and its loop nestings.

ables eliminates non-interesting variables belonging to a slice (e.g. the variable `i`) that would make the bound estimation less precise.

2.2 RTL Semantics with Counters

Instead of reasoning at the assembly level, our loop bound estimation operates on the RTL intermediate language, mainly because RTL programs are represented by their control flow graph (CFG), with explicit program points. RTL stands for "Register Transfer Language". Among the intermediate languages of CompCert, RTL is the most adapted for representing gotos and CFGs. Moreover, the compiler optimizations are also performed at the RTL level and we can benefit from them (e.g. common subexpression elimination). Thus, our loop bound estimation operates at the RTL level and extends the RTL representation of programs with a notion of loop nesting [10]. RTL is just an intermediate representation in our tool: our final theorem is related to assembly code thanks to the correctness of the CompCert compiler, that states that any RTL program behaves as its corresponding assembly program.

Real-time systems only use a restricted form of programming, where each program consists in a main reacting loop triggering tasks that always terminate and where recursion is not allowed [23]. Hence, in our theorems, we consider only finite executions of programs, even if the CompCert semantics model diverging executions. In the same way, functions are inlined before bounding loops. This is how WCET estimation tools proceed to perform interprocedural analyses.

The semantic preservation theorem of the CompCert compiler requires the definition of formal semantics for all the languages of the compiler. Each of these operational semantics is defined in small-step style as a transition relation between execution states. We use σ to denote execution states in the RTL semantics. Among the components of a tuple σ are the current program point l (i.e. a CFG vertex) and an environment E mapping program variables to values. We have instrumented the RTL semantics by counting the number of times each program point is reached. We have thus added counters (i.e. mapping program points to natural numbers) in execution states. We need two kinds of counters: a global counter c_{glob} such that $c_{glob}(l)$ is incremented each time the program point l is reached during program execution, and a local counter c_{loc} modeling the execution of nested loops. We slice n nested loops into n separate loops, and we need local counters to count for each sliced loop how many times each vertex of the loop is reached. Thus, local counters are incremented as global counters, except at loop exits where they are reset to zero. Loop exits depend on loop nestings and are defined in Section 3.

We use $\sigma.l$, $\sigma.E$, $\sigma.c_{glob}$ and $\sigma.c_{loc}$ to denote label, environment and counters of a program state σ, respectively. We use $\texttt{dom}(\sigma.E)$ to denote the domain of the environment $\sigma.E$ (i.e. the set of its variables). We write $P \Downarrow c_{glob}$ to express that the execution of program P terminates with the final counters c_{glob}. In this paper, we omit the value returned by the main function of the program, even if it is part of the program behavior in our development. We use $\texttt{reach}(P)$ to denote the set of states belonging to the execution trace of P.

2.3 Soundness of Loop Bound Estimation

We prove two soundness theorems for our analysis. The first main theorem states the soundness of the loop bound estimation at the RTL level. For any RTL program P and any program point l of P, the bound estimation at l is a correct estimation of the counter computed by the instrumented semantics at l.

Theorem 1 (Main theorem). *Let P be a RTL program such that $P \Downarrow c_{glob}$ and l a program point of P. Then, we have $c_{glob}(l) \leq \texttt{bound}(P)(l)$.*

[Coq Proof][4]

Note that this theorem (and the following one) only gives estimations on finite executions. This limitation is inherited from the SWEET methodology we formalize here. A termination analysis (e.g. see [8]) may be required here but formally verifying it is out of the scope of this paper.

The second main theorem states the start-to-end (i.e. from C to assembly) property of our enhanced compiler, that generates an executable code as well as a table of bounds for every program point where an annotation is attached. The CompCert semantics emit a special event each time such a point is reached during program execution. Then, we characterize bounds as an over-approximation of the number of occurrences of such an event in the execution trace of assembly

[4] This is a direct link to the web page showing the corresponding Coq theorem.

programs. In other words, this theorem states that the number of executions we estimate for a given program point at the RTL level is still true at the assembly level.

Theorem 2 (Start-to-end correctness). *Let P_C be a source C program, free of runtime errors. Let P_{Asm} and* `bound_table` *be the result of the compilation of P_C. Then, for any finite execution P_{Asm} that produces a trace of events* `tr` *and any annotation label* `al`*, we have $\#\texttt{tr}_{\downarrow\texttt{al}} \leq \texttt{bound\_table}[\texttt{al}]$, where $\#\texttt{tr}_{\downarrow\texttt{al}}$ represents the number of occurrences of the event attached to* `al` *in* `tr`*.*

[Coq Proof]

This theorem is a consequence of the main theorem and the CompCert theorems about preservation of annotation events trough compilation. As our loop bound estimation relies on three main tasks (loop reconstruction, program slicing and local bound calculation), the proof of the first main theorem follows from the proof of each of these tasks, that are detailed in the three following sections.

Example 1. In program `P` of Fig. 2, our enhanced compiler will generate a table that associates the string "loop1" (resp. "loop2") to the bound 6 (resp. 36).

Our proofs follow the methodology chosen to formally verify the CompCert compiler [16]. Most of the compiler passes are written and proved in Coq. Other passes of the compiler (e.g. the register allocation and some optimizations such as software pipelining) are not written in Coq but validated *a posteriori*. We have implemented efficiently in OCaml some algorithms and we have formally verified (in Coq) a checker that validates *a posteriori* the untrusted results of the OCaml program. More precisely, we have validated *a posteriori* two algorithms, an efficient algorithm for computing loop nestings from a CFG, and the control and data dependence analysis of the slicer.

3 Loop Nestings

Reasoning about loops on a CFG may require complex proofs in graph theory. The 3 tasks of our tool manipulate CFGs that are equipped with *loop nestings*. Loop nestings represent a hierarchical view of the CFG loops. First, this section specifies loop nestings. Then, it explains how they are built. In Section 2.1, we mentioned that the user provides marks (e.g. see the program `P` in Fig. 2) to indicate the program points that are annotated in the final assembly program. The information we compute in Section 3 does not use these marks at all.

3.1 Axiomatization of Loop Nestings

Our axiomatization of loop nestings (that we call nestings in the sequel of this paper) is given in Fig. 3, where the abstract type for nestings is called `t`. The right part of the previous example given in Fig. 2 shows the three nested nestings associated with program `P`. Given a nesting s, `vertices`(s) denotes the list of

its vertices. A vertex v belongs to a nesting s (notation $v \in s$) if it belongs to the list of its vertices. In the same way, we define an inclusion relation $\subseteq$ between nestings as a set inclusion between the sets of vertices of the nestings.

Each RTL function f must be equipped with a *family* of nestings. The type called $\texttt{family}(f)$ describes in a Coq record the elements of such a family. It contains four functions `nesting`, `header`, `parent` and `elements` and eleven properties about these functions. The record type is itself parameterized by the function f because some properties directly mention it.

Each vertex v of the CFG belongs to its nesting $\texttt{nesting}(v)$ (P_1) that is the least nesting containing v (P_2). Each nesting s is given a header vertex $\texttt{header}(s)$ in f (P_3) such that its nesting is s itself (P_4). It implies that $\texttt{header}(s) \in s$. The header of a nesting represents the loop entry. For instance, in Fig. 2, $\texttt{header}(11) = 6$. The hierarchy of nestings is described by a map called `parent` returning the parent nesting of a nesting. The parent nesting $\texttt{parent}(s)$ of a nesting s contains s itself (P_5) and is included in any (strict) sub-nesting of s (P_6).[5] At last, the family contains a list called `elements` of all its nestings (P_7).

Only three properties relate nestings and CFG edges. (P_8) ensures that $\texttt{header}(s)$ is the unique entry of s and that the only incoming edges start from $\texttt{parent}(s)$. (P_9) ensures that each CFG cycle is cut by a header, except for loops starting at headers which are either totally included in their nesting or that are cut by the header of the parent nesting (P_{10}).

The last property (P_{11}) describes the specific role of the CFG entry point.

In our semantics, local counters are reset at loop exits. We use nestings to define precisely loop exits in the semantics. Exiting the loop of a vertex n_0 means traversing an edge $n \mapsto n'$ such that $n \in \texttt{nesting}(n_0)$ but $n' \notin \texttt{nesting}(n_0)$.

3.2 Computation of Loop Nestings

Various algorithms exist in the literature to compute nestings. We follow the Bourdoncle algorithm [10], a variation of the famous Tarjan algorithm for computing strongly connected components. We chose this algorithm because it is also useful for our value analysis. The worst-case complexity of this algorithm is $D \times E$ where D is the maximum depth of the graph vertices and E is the number of edges. The algorithm gives a weak topological ordering of the CFG.

We have implemented in OCaml our algorithm, and we have formally verified a checker that validates *a posteriori* the untrusted results of the algorithm. We use the nesting ordering to efficiently check the properties (P_9) and (P_{10}) about cycles. Our verified checker takes as input a nesting of the following type.

$$\textbf{Inductive}\ \texttt{nesting} := \texttt{I}(v : \texttt{vertex}) \mid \texttt{L}(h : \texttt{vertex})(l : \texttt{list nesting})$$

An element of type `nesting` is either a single vertex ($\texttt{I}\ v$) that directly belongs to the current nesting or a new nesting ($\texttt{L}\ h\ l$) with h a header vertex and l a list of sub-elements. The verified checker outputs a record of type ($\texttt{family}\ f$) or aborts if the verification fails. Let us note that our checker could also validate any other algorithm (e.g. [17]) for computing loop nestings.

[5] The functions `header`, `nesting` and `parent` will be used in the lemmas of Section 5.

Parameter t : **Type**
Parameter $\texttt{vertices} : t \to \texttt{list node}$
Notation $v \in s := v \in_{list} \texttt{vertices}(s)$
Notation $s_1 \subseteq s_2 := \texttt{vertices}(s_1) \subseteq_{set} \texttt{vertices}(s_2)$

Record $\texttt{family}(f : \texttt{function}) := \{$
(f_1) $\texttt{nesting} : \texttt{vertex} \to t$
(f_2) $\texttt{header} : t \to \texttt{vertex}$
(f_3) $\texttt{parent} : t \to t$
(f_4) $\texttt{elements} : \texttt{list}\ t$
(P_1) $\texttt{in\_nesting} : \forall v, \texttt{f\_In}(v, f) \Rightarrow v \in \texttt{nesting}(v)$
(P_2) $\texttt{nesting\_least} : \forall s \in_{list} \texttt{elements}, \forall v \in s, \texttt{nesting}(v) \subseteq s$
(P_3) $\texttt{header\_f\_In} : \forall s \in_{list} \texttt{elements}, \texttt{f\_In}(\texttt{header}(s), f)$
(P_4) $\texttt{nesting\_header} : \forall s \in_{list} \texttt{elements}, \texttt{nesting}(\texttt{header}(s)) = s$
(P_5) $\texttt{incl\_in\_parent} : \forall s \in_{list} \texttt{elements}, s \subseteq \texttt{parent}(s)$
(P_6) $\texttt{parent\_least} : \forall s\, s' \in_{list} \texttt{elements}, s \subseteq s' \Rightarrow s = s' \ \vee\ \texttt{parent}(s) \subseteq s'$
(P_7) $\texttt{nesting\_in\_elements} : \forall v, \texttt{nesting}(v) \in_{list} \texttt{elements}$
(P_8) $\texttt{enter\_in\_nesting\_at\_header\_only} : \forall v\, v', \texttt{is\_succ\_vertex}(f, v, v') \Rightarrow$
$v \notin \texttt{nesting}(v') \Rightarrow v' = \texttt{header}(\texttt{nesting}(v'))\ \wedge\ \texttt{parent}(\texttt{nesting}(v')) = \texttt{nesting}(v)$
(P_9) $\texttt{cycle\_at\_not\_header} : \forall l \neq nil, \forall v, v \neq \texttt{header}(\texttt{nesting}(v)) \Rightarrow$
$\texttt{path}(f, v, l, v) \Rightarrow \texttt{header}(\texttt{nesting}(v)) \in_{list} l$
(P_{10}) $\texttt{cycle\_at\_header} : \forall l \neq nil, \forall v, v = \texttt{header}(\texttt{nesting}(v)) \Rightarrow \texttt{path}(f, v, l, v) \Rightarrow$
$\texttt{header}(\texttt{parent}(\texttt{nesting}(v))) \in_{list} l\ \vee\ (\forall v' \in_{list} l, v' \in \texttt{nesting}(v))$
(P_{11}) $\texttt{in\_nesting\_root} : \forall s \in_{list} \texttt{elements}, \texttt{fn\_entrypoint}(f) \in s \Rightarrow$
$\texttt{fn\_entrypoint}(f) = \texttt{header}(s)\}$.

Fig. 3. Axiomatization of loop nestings

4 Program Slicing

As shown previously in Fig.2, each local bound is estimated from a slice of the program. Precise slicing is an important step in this methodology because it reduces the number of variables we have to consider when estimating the sizes of the domains of the variables that are used in a loop. First, this section presents the two soundness theorems we proved on our program slicer. Secondly, it describes the *a posteriori* validation of our program slicing. Then, it explains the matching we define between execution states in order to prove the soundness.

4.1 Soundness Theorems

Given a program point l_s of a program P, slicing P w.r.t. the slicing criterion l_s means slicing P w.r.t. all the variables that are used at l_s. Two theorems state the soundness of program slicing. The first one is the soundness of program slicing w.r.t. the local counters[6]. It states that for any terminating program P and slicing criterion l_s, a bound of the local counter at l_s of a sliced program P'

[6] As explained in Section 2, only local counters are considered in theorems related to sliced programs.

is also a bound of the local counter at l_s of the original program P. As we will show in Section 5, this is the key property we use to estimate local bounds on P' instead of P.

Theorem 3. *Let P be a program and l_s a program point of P. Let P' be the sliced program w.r.t. the slicing criterion l_s. If M is a bound of every reachable local counter at l_s in P': $\forall\, \sigma \in \texttt{reach}(P'),\ \sigma.c_{loc}(l_s) \leq M$ then M is also a bound of every reachable local counter at l_s in P: $\forall \sigma \in \texttt{reach}(P),\ \sigma.c_{loc}(l_s) \leq M$.* [Coq Proof]

The second theorem states that if a program P terminates, then its sliced program P' also terminates. This theorem is needed to prove our main theorem related to bound calculation (see Section 5.3). Let us note that this property is not obvious. There are slicing algorithms [20] that transform terminating programs into diverging programs, thus program slicing does not always preserve the termination of programs.

Theorem 4. *Let P be a program and l_s a program point of P. Let P' be the sliced program w.r.t. the slicing criterion l_s. If P terminates, then P' terminates.* [Coq Proof]

The standard approach to prove both theorems is to formalize each component of the slicer: data dependencies, control dependencies and post-dominators. Moreover, we need an executable program slicer relying on efficient data structures such as postdominator trees and program dependence graphs. In order to facilitate the proof and avoid intensive reasoning on these data structures, we formally verify a checker that validates *a posteriori* the untrusted results of a slicer written in OCaml. Another advantage of this approach is that our checker can be reused to verify other program slicers.

4.2 *A Posteriori* Validation of Program Slicing

We implement an untrusted program slicer that, given a program P and a slicing criterion l_s yields a slice $\texttt{SL}(l_s)$ giving the set of vertices preserved by the slicing of P w.r.t. l_s. For any vertex outside this set we transform[7] the corresponding statement (resp. condition) into a skip statement (resp. a constant condition).

Alone, this set $\texttt{SL}(l_s)$ is not enough for an efficient *a posteriori* validation. Because we need to find information that can guide the validator, we reuse the notion of relevant variables and next observable vertices that are used in paper-and-pencil proofs of program slicing [18]. A set $\texttt{RV}(l)$ of relevant variables at program point l contains the variables whose values are preserved by the slicing and influence the computation in $\texttt{SL}(l_s)$. Given a vertex l in $\texttt{SL}(l_s)$, $\texttt{NObs}(l)$ is defined as the closest vertex (i.e. when following a path of CFG edges) to l belonging to $\texttt{SL}(l_s)$; $\texttt{DObs}(l)$ is the distance (i.e. the number of edges of the

[7] Slicing is often described as a program transformation that removes statements, but for the purpose of our soundness proof we need to preserve the CFG structure.

SL	Program P (sliced at 6)	Relevant Variables (RV)	Next Observable (NObs)	Distance to Next Observable (DObs)
1	`n = 5;`	∅	1	0
2	`i = 0;`	{n}	2	0
3	`skip;`	{n, i}	4	1
4	`do { __annot("loop1");`	{n, i}	4	0
5	`  j = 0;`	{n, i}	5	0
6	`  do { __annot("loop2");`	{n, i, j}	6	0
7	`    skip;`	{n, i, j}	11	3
8	`    if (false)`	{n, i, j}	11	2
9	`      skip;`	{n, i, j}	11	2
10	`    skip;`	{n, i, j}	11	1
11	`    j++;`	{n, i, j}	11	0
12	`  } while (j <= n);`	{n, i, j}	12	0
13	`  skip;`	{n, i}	14	1
14	`  i++;`	{n, i}	14	0
15	`} while (i <= n);`	{n, i}	15	0

Fig. 4. Relevant variables and next observable vertices for the program P in Fig. 2, sliced at vertex 6, shown with skip statements and constant conditions at sliced vertices.

shortest path) from l to $\texttt{NObs}(l)$. This distance is used in the proof we detail in Appendix A. It is used to follow the shortest path in the sliced program, and thus select the next statement to execute while avoiding possibly infinite loops.

Fig. 4 shows these sets for each program point of the second slice of the example program of Fig. 2. This slice is written in grey; it is defined as the set $\texttt{SL}(6) = \{1; 2; 4; 5; 6; 11; 12; 14; 15\}$ consisting of the program points without skip statement or constant condition. As the variable `j` is initialized at program point 5, and its last use is at program point 12, `j` is relevant in program points 6 to 12. Vertices 7 and 8 do not belong to the slice; $\texttt{NObs}(7)$ (resp. $\texttt{NObs}(8)$) gives the closest vertex of 7 (resp. 8) that belongs to $\texttt{SL}(6)$. Thus, $\texttt{NObs}(7) = \texttt{NObs}(8) = 11$. $\texttt{DObs}(7)$ is 3, the length of the shortest path from 7 to 11; $\texttt{DObs}(8)$ is 2.

We implement a checker taking as input the results of an untrusted slicer and performing some coherence checks to ensure mainly the properties that are described in Fig. 5. They axiomatize the notions of slice, relevant variables and observable vertices. They are checked all at once. The figure shows only the main properties; similar properties taking into account memory accesses are ensured in our Coq development. In Fig. 5, l_s denotes a vertex that is a slicing criterion. We use $n \mapsto s$ to denote a vertex n and its successor s. We use $\texttt{def}(n)$ (resp. $\texttt{use}(n)$) to denote the set of defined (resp. used) variables for a program point n. Property (C_1) states that a slice criterion belongs to its slice.

Fig. 5 shows that $\texttt{RV}(l)$ and $\texttt{SL}(l_s)$ are mutually dependent sets: $\texttt{RV}(l)$ contains the variables that are defined in $\texttt{SL}(l_s)$ and whose value may affect the execution of the statements in $\texttt{SL}(l_s)$, while $\texttt{SL}(l_s)$ contains every statement assigning variables in $\texttt{RV}(l)$. This is expressed by properties (C_2) to (C_4) that characterize a backward data-flow algorithm. Property (C_2) states that any variable that is

(C_1) $l_s \in \texttt{SL}(l_s)$
(C_2) If $n \in \texttt{SL}(l_s)$, then $\texttt{use}(n) \subseteq \texttt{RV}(n)$
(C_3) If $n \mapsto s$, then $\texttt{RV}(s) \backslash \texttt{def}(n) \subseteq \texttt{RV}(n)$
(C_4) If $\texttt{def}(n) \cap \texttt{RV}(n) \neq \emptyset$, then $n \in \texttt{SL}(l_s)$
(C_5) $n \in \texttt{SL}(l_s) \iff \texttt{NObs}(n) = n$
(C_6) If $n \notin \texttt{SL}(l_s) \wedge \texttt{NObs}(n) = o$, then $\forall s, n \mapsto s \Rightarrow \texttt{NObs}(s) = o$
(C_7) If $n \notin \texttt{dom}(\texttt{NObs}) \wedge n \mapsto s$, then $s \notin \texttt{dom}(\texttt{NObs})$
(C_8) If $n \notin \texttt{SL}(l_s) \wedge \texttt{DObs}(n) = d$, then $\forall s, n \mapsto s \Rightarrow \texttt{DObs}(s) \geq d - 1$
(C_9) If $n \notin \texttt{SL}(l_s) \wedge \texttt{DObs}(n) = d$, then $\exists s, n \mapsto s \wedge \texttt{DObs}(s) = d - 1$

Fig. 5. Main formally verified properties related to slices, relevant variables, next observable vertices and distances

used in a slice must be a relevant variable. Property (C_3) expresses the backward propagation from s to n of relevant variables that are not defined at n. The backward propagation ends at vertices where variables are defined. (C_4) states that any vertex n defining a relevant variable belongs to the slice.

The following properties axiomatize next observable vertices and their distance. Property (C_5) states that any vertex of a slice is its own observable vertex. Property (C_6) states that the observable vertex o of a vertex n that is not in the slice is the same for all successors of n. The companion property (C_7) is related to vertices having no next observable vertex: none of their successors has a next observable vertex. Properties (C_8) and (C_9) are related to the distance of next observable vertices. Given a vertex n that is not in the slice such that $\texttt{DObs}(n) = d$, they state that at least one of the successors of n has a distance equal to $d - 1$; some successors may have a greater distance.

Our checker is efficient and verifies the whole properties in a single CFG traversal. Indeed, while [18] introduce relevant variables and sets of observable vertices for the purpose of their paper-and-pencil proof, they are not concerned with computation on this information and state them in terms of paths in the CFG. We have adapted these properties by rewriting them into local properties enabling an efficient checker. Our local properties can be checked just by looking at each vertex and its immediate successors. Moreover, our checker is complete: Ranganath et al. [18] show that standard slicing algorithms based on control and data dependencies always satisfy constraints (C_1) to (C_9).

4.3 Proof by Simulation

To prove the soundness of program slicing, the major difficulty is to relate states occurring during the execution of an initial program P and that of each of its slices P'. To account for these differences between the initial program and each of its slices, we define a matching relation between execution states, written $\sigma \sim \sigma'$ and defined in Fig. 6. To simplify the figure, execution states are considered as triples (program point l, environment E, counters c)[8]; other state components

[8] c denotes either a local or a global counter.

$$\frac{l \in SL(l_s) \quad E \simeq_{RV(l)} E' \quad c(l_s) = c'(l_s)}{(l, E, c) \sim (l, E', c')} \ (R_1)$$

$$\frac{l \notin SL(l_s) \quad l' \notin SL(l_s) \quad \texttt{NObs}(l) = \texttt{NObs}(l') \quad E \simeq_{RV(l)} E' \quad c(l_s) = c'(l_s)}{(l, E, c) \sim (l', E', c')} \ (R_2)$$

$$\frac{l \notin dom(NObs) \quad c(l_s) = c'(l_s)}{(l, E, c) \sim (l_{\text{exit}}, E', c')} \ (R_3)$$

Fig. 6. Matching between execution states of a program and one of its slices $SL(l_s)$

are omitted. Given a program point l, we use $\simeq_{RV(l)}$ to denote the equivalence relation between two environments restricted to relevant variables at l. We use l_{exit} to denote the (unique) exit vertex of the program.

All the rules express that the counters at the slicing criterion must be the same. More constraints on the states are expressed in the rules. The first rule matches intuitively an execution state of the initial program with an execution state of the sliced program when the program point l is the same in both states and it belongs to the slice $SL(l_s)$: both states match when the relevant variables have the same values in both environments E and E'.

The second rule matches two states such that neither of their program points l and l' belong to the slice $SL(l_s)$, but some of their successors belong to $SL(l_s)$. These successors are precisely identified using next observable vertices. Both states match when the next observable vertex at l and l' is the same and, as in the first rule, the relevant variables have the same values in both environments E and E'. The third rule is required to ensure the termination of the sliced program. It matches any state of the initial program such that its program point l exited from the slice (i.e. there is no next observable at l) with the state of the sliced program at program point l_{exit}.

These rules allow us to prove Lemma 1, which states that assuming the constraints of Fig. 5, the sliced program executes in ways that simulate the execution of the corresponding initial program. The proof by simulation of this lemma is detailed in Appendix A. We use $\rightarrow$ to denote a single execution step, and $\rightarrow^*$ to denote the reflexive transitive closure of $\rightarrow$.

Lemma 1. *Let P be a program, l_s a program point of P, and let the result of slicing P w.r.t. l_s be $(P', \texttt{SL}(l_s), \texttt{RV}, \texttt{NObs}, \texttt{DObs})$. Assume $(\texttt{SL}(l_s), \texttt{RV}, \texttt{NObs}, \texttt{DObs})$ satisfy the constraints (C_1) to (C_9). $\forall \sigma_1, \sigma_2 \in \texttt{reach}(P), \sigma_1' \in \texttt{reach}(P')$, if $\sigma_1 \rightarrow \sigma_2$ and $\sigma_1 \sim \sigma_1'$, there exists σ_2' such that $\sigma_1' \rightarrow^* \sigma_2'$ and $\sigma_2 \sim \sigma_2'$.*

[Coq Proof]

5 Bound Calculation

This section explains how we combine program slicing, value analysis and loop nestings to build a safe over-approximation of program counters. This calculation called `bound` is based on the 3 steps we described previously. Each step

is proved by a lemma that is explained in this section. Each proof of a lemma requires to strengthen the lemma into a non-trivial inductive property. We give in Appendix B an account of the formal arguments we have machine-checked.

5.1 The header counter dominates the other counters in the nesting

The nesting header plays an important role for bound calculation since its counter dominates the counters of the other program points in the nesting (i.e. every path from the start to these program points must go through the nesting header). This property is expressed by the following lemma.

Lemma 2. *For any reachable state* $\sigma \in \texttt{reach}(P)$ *and any vertex* l *of* P*, we have:* $\sigma.c_{glob}(l) \leq \sigma.c_{glob}(\texttt{header}(\texttt{nesting}(l)))$. [Coq Proof]

We have proved a similar property for the local counter c_{loc}. Thanks to this lemma, the bounds of a vertex l can be computed by simply computing a bound for its header: $\texttt{bound}(P)(l) = \texttt{bound}(P)(\texttt{header}(\texttt{nesting}(l)))$.

5.2 Relating global and local counters

To compute a bound for the global counter of a nesting header l_h, we need two bounds: a global bound of the global counter of the parent nesting and a local bound of the local counter of the current header. The following lemma states how the local and global counters at l_h are related. We assume the current header differs from the entry point of the program. The latter is executed only once (after a normalization of RTL control flow graphs).

Lemma 3. *Let* l_h *be a nesting header and* l_p *the header of its parent nesting, i.e.* $l_p = \texttt{header}(\texttt{parent}(\texttt{nesting}(l_h)))$*. Let* M *be a bound for the local counter of* l_h*:* $\forall \sigma \in \texttt{reach}(P), \sigma.c_{loc}(l_h) \leq M$*. Then, we have:* $\forall \sigma \in \texttt{reach}(P), \sigma.c_{glob}(l_h) \leq M \times \sigma.c_{glob}(l_p)$

[Coq Proof]

This lemma allows us to program the bound computation of l_h by a recursive call to the bound of its parent followed by a multiplication by the estimation of the local counter in l_h. This local counter is called $\texttt{loc\_bound}(P, l_h)$ and defined in the next subsection.

$$\texttt{bound}(P)(l_h) = \texttt{bound}(P)(\texttt{header}(\texttt{parent}(\texttt{nesting}(l_h)))) \times \texttt{loc\_bound}(P, l_h)$$

5.3 Bounding local counters

Our value analysis (called `value`) computes, at each program point of a program, an over-approximation of the domain size of each variable, i.e. the estimated values (represented by an interval) of the program variables. Thus, given a program P and a vertex l, $\texttt{value}(P)(l)$ yields a map such that for any variable x, $\texttt{value}(P)(l)(x)$ is an interval $[a, b]$ representing a conservative range of the

possible values of x at l. We use $|[a,b]| = b - a + 1$ to denote the size of the interval $[a,b]$. Our formally verified value analysis is detailed in [7].

The value analysis could be used directly to estimate local bounds. We could compute the size of each interval and estimate a local bound as the product of all the sizes that were computed at the loop header. Since we assume that programs terminate, a value in this domain is never reached twice. Thus, we have: $\texttt{loc\_bound}(P, l_h) \leq \prod_{x \in \texttt{vars}(P)} |\texttt{value}(P)(l_h)(x)|$, where $\texttt{vars}(P)$ is the set of all program variables. While intuitive, this inequality requires a good amount of formal details to be proved in a proof assistant (see Appendix B).

Example 2. In the following program, our value analysis will infer the loop invariant $\texttt{i} \in [0,9] \wedge \texttt{j} \in [0,1]$. As a consequence, we bound the local counter of the loop header by $2 \times 10 = 20$.

$$\texttt{j = 0; i = 0; } \textbf{while } \texttt{(i < 9) \{ j = 1 - j; if (j) i++; \}}$$

In order to increase the precision of the local bound estimation, it is important to restrict the set of variables involved in this product. This set is modified as follows. First, we slice P w.r.t. program point l_h and only compute the local bound of l_h on P: $\texttt{loc\_bound}(P, l_h) = \texttt{loc\_bound\_after\_slice}(\texttt{slicing}(P, l_h), l_h)$. Second, in the sliced program $P' = \texttt{slicing}(P, l_h)$, we only consider the interesting variables that are live at l_h and used in a statement belonging to the nesting S of l_h (thus $S = \texttt{nesting}(l_h)$) and also defined in any (possibly different) statement of S.

$$\texttt{loc\_bound\_after\_slice}(P', l_h) = \prod_{x \in \texttt{live}(l_h) \cap \texttt{use}(S) \cap \texttt{def}(S)} |\texttt{value}(P')(l_h)(x)|$$

This last part of the bound computation is proved correct using the following lemma stating that the previous computation over-estimates the local counters.

Lemma 4. *For any reachable state $\sigma \in \texttt{reach}(P')$, we have*

$$\sigma.c_{loc}(l_h) \leq \prod_{x \in \texttt{live}(l_h) \cap \texttt{use}(\texttt{nesting}(l_h)) \cap \texttt{def}(\texttt{nesting}(l_h))} |\texttt{value}(P')(l_h)(x)|$$

[Coq Proof]

By combining lemmas 4, 3 and 2 we obtain the proof of our main Theorem 1.

6 Experimental Evaluation

We have integrated our loop bound estimation in the CompCert 1.11 compiler. Our formal development comprises about 15,000 lines of Coq code (consisting of 8,000 lines of Coq functions and definitions and 7,000 lines of Coq statements and proof scripts) and 1,000 lines of OCaml. Our formalization has been translated into an executable OCaml code using Coq's extraction facility.

Our implementation has been compared to the SWEET reference tool [13] against the Mälardalen WCET benchmark [12], a reference benchmark for

Program	#L	Our tool		SWEET			Our tool		SWEET	
		#LE	%LE	#LE	%LE		#GB	%GB	#GB	%GB
1 adpcm	27	13	48%	22	81%		16	59%	18	67%
2 cnt	4	4	100%	4	100%		4	100%	4	100%
3 cover	3	3	100%	3	100%		3	100%	3	100%
4 crc	6	4	67%	6	100%		6	100%	6	100%
5 edn	12	9	75%	11	92%		12	100%	12	100%
6 expint	2	2	100%	2	100%		2	100%	2	100%
7 fdct	2	2	100%	2	100%		2	100%	2	100%
8 fft1	29	3	10%	6	21%		7	24%	7	24%
9 fibcall	1	1	100%	1	100%		1	100%	1	100%
10 fir	2	1	50%	1	50%		1	50%	2	100%
11 insertsort	2	1	50%	1	50%		1	50%	1	50%
12 jfdctint	3	3	100%	3	100%		3	100%	3	100%
13 lcdnum	1	1	100%	1	100%		1	100%	1	100%
14 ludcmp	11	6	55%	6	55%		6	55%	6	55%
15 matmult	7	7	100%	7	100%		7	100%	7	100%
16 ndes	12	12	100%	12	100%		12	100%	12	100%
17 ns	4	4	100%	4	100%		4	100%	4	100%
18 qurt	3	2	67%	3	100%		3	100%	3	100%
19 ud	11	11	100%	11	100%		11	100%	11	100%
Geometric mean			73%		82%			81%		85%

Fig. 7. Exact local bounds and meaningful global bounds of the benchmark. The numbers of loop bounds are given relative to the total number of loops.

WCET estimation tools. This benchmark provides a set of programs with representative loops, mainly used by WCET tools but also by static analyzers [14]. Its focus on flow analysis makes it a reference on WCET-related loop bound estimations. It is especially suited for interval-based analyses, currently the state-of-the-art on industrial WCET tools. Results for both methods are given in Fig. 7. The programs considered are those analyzed in [9] for which SWEET could estimate at least one bound, excluding 2 of them that CompCert cannot compile (i.e. one program with a `longjmp` statement and another one with an unstructured `switch` statement such as in Duff's device).

The number `#L of loops` of each program is given in the second column of Fig. 7. The third column of Fig. 7 shows the accuracy of our estimation of local bounds: it gives the number `#LE` of estimations of local loop bounds (and their percentage) that are exact bounds. Unfortunately, this column is not given in [9], but we have estimated it from the results of our tool and our manual analysis to infer which loops are estimated by SWEET.

Our results are close to those obtained by SWEET. On average, 73% of the loops are exactly estimated by our method, while 82% of the loops are exactly bounded by SWEET. The histogram in Fig. 7 shows for each program, the number of exact local bounds for our tool (in dark grey) and for SWEET (in black)

relatively to the total number of loops (baseline in light grey). Differences in precision come from our value analysis, that is slightly less precise than SWEET's. As our value analysis does neither handle floating-point values nor global variables, nor performs a pointer analysis, 17 loops are bounded by SWEET and not by our method.

The last two columns of Fig. 7 give the number `#GB` of meaningful estimations (i.e. realistic estimations, that differ from `MAX_INT` for instance) of global loop bounds (and their percentages). On average, our tool estimates almost as many global bounds as SWEET. Indeed, 81% of global bounds are estimated by our tool, and 85% of global loops are estimated by SWEET.

Concerning the analysis time, hardware differences make it difficult to compare them with SWEET's. Nevertheless, we could verify that the use of checkers does not incur a significant overhead in our analysis. Benchmarking the programs in Fig. 7 using a current personal computer takes less than a minute.

7 Related Work

Ranganath, Amtoft et al. [18,20] developed paper-and-pencil proofs of program slicing, introducing the notion of observable vertices. Their main concern is to deal with generalized programs, having several or no end nodes. Ranganath et al. prove slicing soundness by weak bisimulation, dealing with infinite behaviors. Amtoft extends the proof, obtaining a smaller slice by using a weak simulation at the cost of not preserving termination. Based on their work, a formal verification of program slicing in Isabelle is given in [22], where program slicing is used for detecting non-interference of information flow. This formalization of program slicing is relational and generic; it has been instantiated on Java programs in the Jinja framework but it is not executable, contrary to our work.

To the best of our knowledge, the only work related to formal verification of loop bound calculation is [3], where the formal verification consists in using Hoare logic to verify that a program satisfies its specification including a cost annotation (expressed by an equality of the form *global cost = constant value*). A Frama-C plugin has been developed in order to experiment the approach on simple programs without nested loops. Contrary to this work where loops are handled syntactically, our work relies on an abstract interpreter with widening capabilities that has been formally verified in Coq. As far as we know, their Hoare logic is neither formalized nor proved sound.

Many papers have been published on resource analysis [1,2] and loop bound is just one example of resource. The associated algorithms generally target more difficult loop bounds that WCET tools like SWEET or our own tool. It is unclear if they provide significant precision gains on representative WCET benchmarks. Resource analysis tools are not formally verified using a proof assistant. One exception is [2] where a shallow embedding of a separation logic in Coq is mentioned. The only mechanized proof is the soundness of the core logic presented in the research paper. This should not be confused with the kind of formalization effort we provide in order to formally verify a tool for C programs. Advanced

ressource analyses such as [11,24] are able to infer symbolic loop bounds that are out of reach for WCET tools like SWEET. This kind of static analysis relies on SMT solvers, hence their formal verification would require the *a priori* verification of a SMT solver.

Checkers are powerful tools for verifying the soundness of program transformations. Several formally verified checkers have been developed for compiler passes of CompCert (e.g. [21,4]). Even if all these checkers are specific tools devoted to a specific compiler pass, previous work and ours has shown that this alternative formal verification technique is worthwhile when the formalization requires to reason on sophisticated imperative data structures and algorithms.

Our long-term goal is to complement the CompCert compiler with WCET guarantees about the code it generates. The formally verified operating system kernel seL4 [15] is faced to similar challenges. Blackham et al. [6] apply traditional WCET estimation techniques on the seL4 kernel and provide conservative upper bounds about its worst-case interrupt response time. Their WCET tool is neither verified nor formalized.

8 Conclusion

We have presented, formalized and implemented a loop bound estimation for WCET analysis. Its design follows closely the techniques used by the reference tool SWEET and our experiments show that it is competitive with it in terms of precision of the estimated bounds. The work strengthens the CompCert framework. It provides bound estimations on the assembly programs generated by the compiler and it increases the CompCert toolchain with non trivial components that could be reused in different contexts, e.g. for developing new optimizations of the compiler: a loop reconstruction for RTL and a program slicer. Moreover, the bound calculation theorem makes an important formal link between the estimated loop bounds and the size of variable ranges.

Our loop bound estimation can be improved in several directions. One is to improve the bound calculation by formalizing ILP solvers to relate precisely local and global bounds. These solvers contain probably too much highly engineered heuristics to be directly formalized and we would like to develop efficient validation checkers for them. Another direction is to increase CompCert with a precise hardware cost model and link abstract counters and realistic costs.

References

1. E. Albert, R. Bubel, S. Genaim, R. Hähnle, and al. Verified resource guarantees using COSTA and KeY. In *PEPM '11*, pages 73–76. ACM, 2011.
2. R. Atkey. Amortised resource analysis with separation logic. *Logical Methods in Computer Science*, 7(2), 2011.
3. N. Ayache, R. M. Amadio, and Y. Régis-Gianas. Certifying and reasoning on cost annotations in C programs. In *Proc. of FMICS*, volume 7437 of *LNCS*, pages 32–46. Springer, 2012.

4. G. Barthe, D. Demange, and D. Pichardie. A formally verified SSA-based middle-end - Static Single Assignment meets CompCert. In *Proc. of ESOP*, volume 7211 of *LNCS*, pages 47–66. Springer, 2012.
5. R. Bedin França, S. Blazy, D. Favre-Felix, X. Leroy, and al. Formally verified optimizing compilation in ACG-based flight control software. In *ERTS*, 2012.
6. B. Blackham, Y.Shi, and G. Heiser. Improving interrupt response time in a verifiable protected microkernel. In *Proc. of EuroSys*, pages 323–336. ACM, 2012.
7. S. Blazy, V. Laporte, A. Maroneze, and D. Pichardie. Formal verification of a C value analysis based on abstract interpretation. In *Proc. of Static Analysis Symposium (SAS)*, 2013. To appear.
8. B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI 2006*, pages 415–426. ACM Press, 2006.
9. 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 *Workshop on WCET Analysis*, 2007.
10. F.Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proc. of FMPA 1993*, volume 735 of *LNCS*, pages 128–141. Springer-Verlag, 1993.
11. S. Gulwani. SPEED: Symbolic complexity bound analysis. In *Proc. of CAV*, volume 5643 of *LNCS*, pages 51–62, 2009.
12. J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET benchmarks: Past, present and future. In *Proc. WCET 2010*, pages 137–147, 2010.
13. J. Gustafsson and A. Ermedahl. Automatic derivation of path and loop annotations in object-oriented real-time programs. *Scalable Computing: Practice and Experience*, 1(2), 1998.
14. N. Halbwachs and J. Henry. When the decreasing sequence fails. In *Proc. of Static Analysis Symposium (SAS)*, LNCS, pages 198–213. Springer, 2012.
15. G. Heiser, T. C. Murray, and G. Klein. It's time for trustworthy systems. *IEEE Security & Privacy*, 10(2):67–70, 2012.
16. X. Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, 2009.
17. G. Ramalingam. On loops, dominators, and dominance frontiers. *ACM TOPLAS*, 24(5):455–490, September 2002.
18. V. P. Ranganath, T. Amtoft, A. Banerjee, J. Hatcliff, and al. A new foundation for control dependence and slicing for modern program structures. *ACM TOPLAS*, 29(5), 2007.
19. RTCA. DO-178C: Software considerations in airborne systems and equipment certification. Radio Technical Commission for Aeronautics Std., 2012.
20. T.Amtoft. Slicing for modern program structures: a theory for eliminating irrelevant loops. *Inf. Process. Lett.*, 106(2):45–51, 2008.
21. J.-B. Tristan and X. Leroy. A simple, verified validator for software pipelining. In *Proc. of POPL*, pages 83–92. ACM Press, 2010.
22. D. Wasserrab and A. Lochbihler. Formalizing a framework for dynamic slicing of program dependence graphs in Isabelle/HOL. In *Proc. of TPHOL*, volume 5170 of *LNCS*, pages 294–309, 2008.
23. R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, and al. The worst-case execution-time problem — overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7:36:1–36:53, May 2008.
24. F. Zuleger, S. Gulwani, M. Sinn, and H. Veith. Bound analysis of imperative programs with the size-change abstraction. In *Proc. of Static Analysis Symposium (SAS)*, volume 6887 of *LNCS*, pages 280–297. Springer, 2011.

A Detailed simulation proof for the program slicing

The simulation between states used in the soundness proof of program slicing is a *weak simulation*, i.e. it does not preserve infinite executions. In particular, a program P may not terminate while a sliced program P' may terminate (some infinite loops may be sliced away).

We consider two matching states, $\sigma_1 \in \texttt{reach}(P)$ and $\sigma'_1 \in \texttt{reach}(P')$: $\sigma_1 \sim \sigma'_1$. An execution step $\sigma_1 \to \sigma_2$ is *observable* if $\sigma_1.l \in \texttt{SL}(l_s)$, and *silent* otherwise. A silent step corresponds to an execution step from a `skip` statement in the sliced program.

Assume the program P satisfies the constraints (C_1) to (C_9). Its entry point is always in the slice, so at the beginning of the execution, the rule (R_1) holds. If $\sigma_1 \to \sigma_2$, then P' follows in lock-step $(\sigma'_1 \to \sigma'_2)$ to the same vertex l_2, which corresponds to the intuitive idea that a statement in the slice is executed in both programs. If $l_2 \in \texttt{SL}(l_s)$, then rule (R_1) still holds. Otherwise, either l_2 has a next observable vertex (and rule (R_2) holds), or it doesn't ($l_2 \notin dom(\texttt{NObs})$) and rule (R_3) holds.

The crux of the simulation happens when the original and sliced programs are desynchronized: rule (R_2) holds and the states have different program points. In this case, whenever $\sigma_1 \to \sigma_2$, there is no corresponding step in the sliced program, until σ_2 is reaching a vertex belonging to the slice. Properties (C_2) to (C_4) ensure that no relevant variable will be modified in P, so the matching relation still holds.

When the execution returns in the slice ($\sigma_2 \in \texttt{SL}(l_s)$), we need to resynchronize the programs. In this case, the sliced program performs one or several (silent) steps ($\sigma'_1 \to^+ \sigma'_2$) until it reaches $\sigma 2'.l = \sigma 2.l$, where $\sigma 2.l$ (resp. $\sigma 2'.l$) is the next observable vertex of $\sigma_1.l$ (resp. $\sigma'_1.l$). This is where the *next observable distance* comes into play: it exhibits a finite number of steps that are required for the resynchronization to happen. After resynchronization, both states match again and rule (R_1) holds.

This alternation between vertices inside and outside the slice can happen several times, until either P reaches the exit node (if it is in the slice) and the simulation ends, or until P reaches a vertex that is after the slice ($\sigma_2.l \notin dom(\texttt{NObs})$). In this case, property (C_7) ensures that we cannot return to the slice anymore. P' then performs an arbitrary number of steps until it reaches the (unique) end vertex l_{exit}. By using an *exit distance* similar to the next observable distance, we know that l_{exit} can always be reached in a finite number of steps. This ensures termination of P'.

Afterwards, states match by rule (R_3), for any further steps performed by P. We do not need to match match relevant variables anymore.

B Detailed proofs for the bound calculation

Proof (of Lemma 2). We establish this property by proving, by induction on finite execution traces, that for any vertex l, distinct from its header $l_h = \texttt{header}(\texttt{nesting}(l))$, and any partial finite execution trace $\xi = \sigma, \sigma_1, ...\sigma_n$, one of three following conditions holds:

- either the expected inequality holds strictly: $\sigma.c_{glob}(l) < \sigma.c_{glob}(l_h)$,
- or l has not been reached yet: $\sigma.c_{glob}(l) = 0$,
- or $\sigma.c_{glob}(l) = \sigma.c_{glob}(l_h)$ but there exists $k \in [0, n-1]$ such that σ_k is at the program point l and all states $\sigma_{k+1}, \ldots, \sigma_{n-1}$ did not reach the header l_h.

The last condition implies that we cannot reach vertex l in σ_n: it would build a cycle from l to l that does not contain l_h and this is forbidden by the property `cycle_at_not_header` (Section 3).

Proof (of Lemma 3). We first consider execution traces of the form $\xi = \xi_0 \cdot \sigma_p \cdot \xi_1 \cdot \sigma$ such that σ_p is a state at point l_p and all states in trace ξ_1 did not reach l_p again. On such traces we show that $\sigma.c_{glob}(l_h) - \sigma_p.c_{glob}(l_h) \leq M$ holds. Unfortunately, this property is not inductive. We strengthen it into a disjunction where:

- either $\sigma.c_{glob}(l_h) = \sigma_p.c_{glob}(l_h)$ and no state in ξ_1 did reach l_h yet,
- or $\sigma.c_{glob}(l_h) = \sigma_p.c_{glob}(l_h) + \sigma.c_{loc}(l_h)$ and the state σ is currently in the nesting of l_h,
- or σ is currently out of the nesting of l_h, $\sigma.c_{glob}(l_h) - \sigma_p.c_{glob}(l_h) \leq M$ and l_h has been reached during ξ_1.

We prove this disjunction by induction on the execution trace ξ_1.

To conclude this proof, we consider an arbitrary trace $\xi = \sigma_0 \cdots \sigma$ and we divide it into $K + 1 = 1 + \sigma.c_{glob}(l_p)$ subtraces $\xi = \xi_0 \cdot \xi_1 \cdots \xi_K$ such that $\forall i \in [1, K], \xi_i$ starts with a state σ_i at point l_p and then never reaches it again. We note $\sigma_{K+1} = \sigma$. We then express $\sigma.c_{glob}(l_h)$ as

$$\sigma.c_{glob}(l_h) = \sigma_0.c_{glob}(l_h) + \sum_{k=0}^{K-1} (\sigma_{k+1}.c_{glob}(l_h) - \sigma_k.c_{glob}(l_h))$$

In the initial state σ_0, every counter is null and each element in the sum is bounded by M. Thus, we conclude that $\sigma.c_{loc}(l_h) \leq K \times M$ and finish the proof since $K = \sigma.c_{glob}(l_p)$.

Proof (of Lemma 4). Given a vertex l, we use $I(l)$ to denote the set $\texttt{live}(l) \cap \texttt{use}(\texttt{nesting}(l_h)) \cap \texttt{def}(\texttt{nesting}(l_h))$ of interesting variables at l. We first prove that, if there exists an execution trace $\xi = \xi_1 \cdot \sigma_1 \cdot \xi_2 \cdot \sigma_2$ such that $\sigma_1.l = \sigma_2.l$ and both states σ_1 and σ_2 match pointwise on each variable of $I(l)$, then we can build a valid execution trace of arbitrary large size $\xi_1 \cdot \sigma_1 \cdot (\xi_2 \cdot \sigma_2)^N$. Since we assume that P terminates, we obtain a contradiction.

Now, any execution trace ξ reaching l_h at least once can be divided into $\xi = \xi_1 \cdot \sigma_1 . \xi_2 . \sigma_2$ where σ_1 is the last state in the execution that enters in the nesting of l_h. The counter $\sigma_2.c_{loc}(l_h)$ is equal to the length of the sub-trace ξ^h

that we obtain by projecting $\sigma_1.\xi_2.\sigma_2$ on the states that are at vertex l_h. Each state in ξ^h can be turned into a n-tuple, where $n = |I(l_h)|$ contains the value of each variable of $I(l_h)$ in this state. Mapping this transformation on ξ^h, we obtain a list of size $\sigma_2.c_{loc}(l_h)$. This list contains distinct n-tuples thanks to the *Reductio ad absurdum* we made early in this proof. By soundness of the value analysis, each n-tuple belongs to the direct product of the interval $\texttt{value}(P')(l_h)$. We prove that there exists a list of size $\prod_{x \in I} |\texttt{value}(P')(l_h)(x)|$ containing all the possible n-uples of this direct product and conclude our proof by a pigeon hole argument.