

Improving WCET Evaluation using Linear Relation Analysis

Pascal Raymond, Claire Maiza, Catherine Parent-Vigouroux, Erwan Jahier, Nicolas Halbwachs, Fabienne Carrier, Mihail Asavoae, Rémy Boutonnet

▶ **To cite this version:**

Pascal Raymond, Claire Maiza, Catherine Parent-Vigouroux, Erwan Jahier, Nicolas Halbwachs, et al.. Improving WCET Evaluation using Linear Relation Analysis. Leibniz Transactions on Embedded Systems, 2019, 6 (1), pp.02:28. 10.4230/LITES-v006-i001-a002 . hal-03013987

HAL Id: hal-03013987
https://hal.science/hal-03013987

Submitted on 19 Nov 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Improving WCET Evaluation using Linear Relation Analysis*

Pascal Raymond
Univ. Grenoble Alpes, CNRS, Grenoble INP (Institute of Engineering Univ. Grenoble Alpes), VERIMAG, Grenoble, France
Pascal.Raymond@univ-grenoble-alpes.fr

Claire Maiza
Univ. Grenoble Alpes, CNRS, Grenoble INP (Institute of Engineering Univ. Grenoble Alpes), VERIMAG, Grenoble, France
Claire.Maiza@univ-grenoble-alpes.fr

Catherine Parent-Vigouroux
Univ. Grenoble Alpes, CNRS, Grenoble INP (Institute of Engineering Univ. Grenoble Alpes), VERIMAG, Grenoble, France
Catherine.Parent-Vigouroux@univ-grenoble-alpes.fr

Erwan Jahier
Univ. Grenoble Alpes, CNRS, Grenoble INP (Institute of Engineering Univ. Grenoble Alpes), VERIMAG, Grenoble, France
Erwan.Jahier@univ-grenoble-alpes.fr

Nicolas Halbwachs
Univ. Grenoble Alpes, CNRS, Grenoble INP (Institute of Engineering Univ. Grenoble Alpes), VERIMAG, Grenoble, France
Nicolas.Halbwachs@univ-grenoble-alpes.fr

Fabienne Carrier
Univ. Grenoble Alpes, CNRS, Grenoble INP (Institute of Engineering Univ. Grenoble Alpes), VERIMAG, Grenoble, France
Fabienne.Carrier@univ-grenoble-alpes.fr

Mihail Asavoae
Univ. Grenoble Alpes, CNRS, Grenoble INP (Institute of Engineering Univ. Grenoble Alpes), VERIMAG, Grenoble, France
Mihail.Asavoae@univ-grenoble-alpes.fr

Rémy Boutonnet
Univ. Grenoble Alpes, CNRS, Grenoble INP (Institute of Engineering Univ. Grenoble Alpes), VERIMAG, Grenoble, France
Rémy.Boutonnet@univ-grenoble-alpes.fr

Abstract

The precision of a worst case execution time (WCET) evaluation tool on a given program is highly dependent on how the tool is able to detect and discard semantically infeasible executions of the program. In this paper, we propose to use the classical abstract interpretation-based method of *linear relation analysis* to discover and exploit relations between execution paths. For this purpose, we add auxiliary variables (counters) to the program to trace its execution paths. The results are easily incorporated in the classical workflow of a WCET evaluator, when the evaluator is based on the popular *implicit path enumeration technique*. We use existing tools – a WCET evaluator and a linear relation analyzer – to build and experiment a prototype implementation of this idea.

* This work is supported by the French research fundation (ANR) as part of the W-SEPT project (ANR-12-INSE-0001)



© Pascal Raymond, Claire Maiza, Catherine Parent-Vigouroux, Erwan Jahier, Nicolas Halbwachs, Fabienne Carrier, Mihail Asavoae, and Rémy Boutonnet;
licensed under Creative Commons Attribution 3.0 Germany (CC BY 3.0 DE)



2012 ACM Subject Classification Software and its engineering → Real-time systems software
Keywords and Phrases Worst Case Execution Time estimation, Infeasible Execution Paths, Abstract Interpretation
Digital Object Identifier 10.4230/LITES.6.1.2
Received 2017-12-12 **Accepted** 2018-11-19 **Published** 2019-02-18

1 Introduction

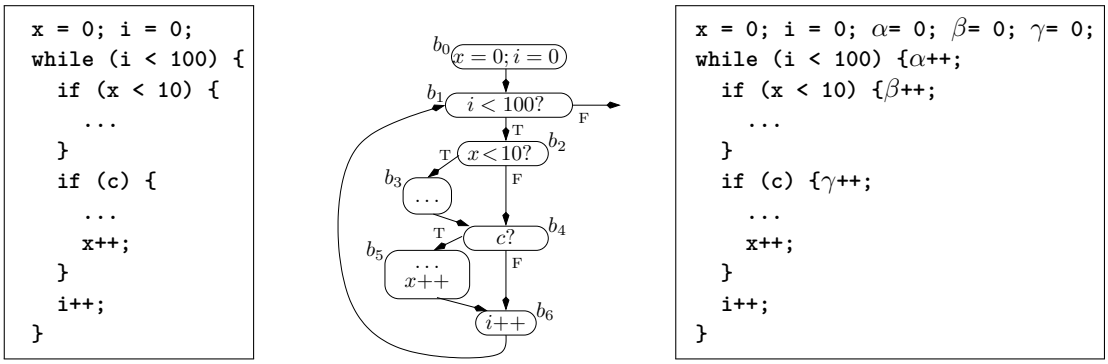
The computation of a precise and safe approximation of the worst case execution time (WCET) of programs on a given architecture is an important step in the design of hard real-time systems [41]. It is part of the validation of the design, and a prerequisite for tasks scheduling. In this computation, over-approximation is mainly due to pessimistic abstraction of (1) complex hardware mechanisms (caches, pipeline) and (2) the program semantics (loop bounds, infeasible executions). Taking into account the target execution platform is, by far, the most difficult problem. It has been largely studied in the literature and remarkable tools exist, both in the academia [5, 27, 29] and in the industry [40].

In this paper, we specifically address the problem of taking into account the program semantics. The objective is to extract semantic properties that make some executions infeasible, and to exploit these properties in an existing WCET evaluator. It is generally admitted that such properties are easier to analyze on high-level code – e.g., C programs – than on binary, even if semantic analysis of executable code has been explored [3, 4, 36]. WCET evaluation is performed on object code in order to be able to take into account the execution architecture. This raises the problem of traceability between the source and the object code.

The most popular approach to evaluate the WCET is called *implicit path enumeration technique* (IPET) [28]. A micro-architectural analysis provides an evaluation of the duration of each basic block of the object-code control-flow graph. The WCET is then expressed as the solution of an integer linear programming problem (ILP) where the variables are the number of times each basic block is traversed during an execution. Relations between these variables come from the control-flow graph (flow equations) and from semantic "*flow facts*", including at least *loop bounds*. Indeed, each loop in the program should have a constant bound to guarantee that the execution time is finite; such bounds may be provided by the user, or discovered by program analysis.

Hence, the IPET-based evaluation takes into account semantic properties expressed as linear constraints on counters. A natural idea is then to combine it with a semantic analysis devoted to the discovery of invariant linear relations. Polyhedra-based abstract interpretation [2, 8, 17, 20], also called *linear relation analysis* (LRA), is such an analysis. It is able to associate with each control point of a sequential program a system of linear inequalities (whose set of solutions is a convex polyhedron) satisfied by the numerical variables at this control point in any execution of the program.

Our proposal consists in applying LRA to a copy of the source program enriched with counter variables, and translate the obtained relations between counters into constraints to be added to the ILP. Let us illustrate this proposal with a simple example.



(a) initial program **(b)** control flow graph **(c)** instrumented program

Figure 1 Instrumenting an example program with counters.

1.1 An example

Consider the program fragment of Figure 1.a with its control-flow graph (Fig. 1.b). Let us add counters α, β, γ to the main basic blocks as done in the instrumented program Figure 1.c. These counters are initialized to 0 and incremented in their corresponding block. An LRA analysis of this instrumented program automatically discovers that the following relations are satisfied at the end of the program:

$$\alpha = \texttt{i} = 100\ ,\ \gamma = \texttt{x}\ ,\ \beta + \gamma \leq 110\ ,\ \gamma \geq 0\ ,\ \beta \geq 0$$

The inequality $\alpha = 100$ gives the exact bound of the loop. More interestingly, $\beta + \gamma \leq 110$ means that there are at most 10 iterations of the loop where both blocks b_3 and b_5 are executed.

Assume the object code has the same control structure as the C program, i.e., the basic blocks of their control flow graphs are in an one-to-one correspondence. The standard WCET evaluation computes pessimistic execution times (say $t_i, i = 0..6$) of the basic blocks $(b_i, i = 0..6)$, and constructs the following ILP, where n_i (resp., $e_{i,j}$) denotes the number of occurrences of the basic block b_i (resp., the edge from b_i to b_j) in an execution of the program:

$$wcet = \max \sum_{i=0}^{6} n_i.t_i \quad , \quad \text{with the constraints} \quad \left\{ \begin{array}{lll} n_0 = 1 & , & e_{0,1} = n_0 \\ n_1 = e_{0,1} + e_{6,1} & , & e_{1,2} + 1 = n_1 \\ n_2 = e_{1,2} & , & e_{2,3} + e_{2,4} = n_2 \\ n_3 = e_{2,3} & , & e_{3,4} = n_3 \\ n_4 = e_{2,4} + e_{3,4} & , & e_{4,5} + e_{4,6} = n_4 \\ n_5 = e_{4,5} & , & e_{5,6} = n_5 \\ n_6 = e_{4,6} + e_{5,6} & , & e_{6,1} = n_6 \end{array} \right\}$$

If we are able to maintain the correspondence between basic blocks in the source and the object code, i.e., to associate our counters α, β, γ with the variables of the ILP (n_2, n_3, n_5 respectively), we can add to the ILP the corresponding constraints: $n_2 = 100, n_3 + n_5 \leq 110$, which is likely to reduce the maximum value of the objective function[1].

[1] In fact, for this simple example, the results can be computed symbolically: concerning the standard evaluation, if the number of iterations in the loop (= 100) is given as a flow fact, the result will be $t_0 + 101t_1 + 100(t_2 + t_3 + t_4 + t_5 + t_6)$. Taking the additional constraint into account, we get $t_0 + 101t_1 + 100(t_2 + t_4 + t_6) + 100\max(t_3, t_5) + 10\min(t_3, t_5)$ thus improving the previous result by $90\min(t_3, t_5)$.

1.2 Contents of the paper

In Section 2, we focus on some available tools, and experiment their semantics awareness on some simple examples. Two recent papers were dedicated to the state of the art related to semantic analyses for WCET estimation and infeasible path detection [1, 10]. Section 2.3 presents some more recent publications.

Our proposal consists in combining existing techniques, namely IPET-based WCET analysis and Linear Relation Analysis, recalled in Section 3, together with the specific tools that we used in our implementation. In Section 4, we explain how the counters are added and related to ILP counters thanks to debugging information provided by the compiler. Our implementation of the method is used to validate the approach on two existing benchmarks. We also investigated the robustness of the approach in presence of compiler optimizations. These experiments are summarized in Section 5. We conclude with the discussion of possible future work.

2 Existing tools

We have experimented with some existing tools, to evaluate their ability to discover and exploit semantic properties. Four tools have been considered, all of which go through similar steps:

1. extracting a control-flow graph from the object code,
2. performing a set of micro-architectural analyses to obtain execution times for each basic blocks,
3. using IPET to compute a safe WCET.

We compare these tools with respect to their capabilities to extract semantic properties to cut infeasible paths.

2.1 The tools

2.1.1 The Chronos Timing Analyzer

Chronos [27] is an academic tool developed at National University of Singapore. It takes as input a C program, performs limited data-flow analysis at C source code level to determine loop bounds, and requests the user to provide this information when it fails. The semantic analysis in Chronos uses a pattern-based method to detect infeasible paths [39]. The so-called two-phase technique addresses infeasibility from a *conflicting pairs* point of view. In the first phase, an analysis detects some conflicts that capture the fact that two branches can not be taken along the same path. In the second phase, each conflicting pair relation is encoded into an ILP constraint.

2.1.2 The Swedish Timing Analyzer

SWEET[2] [29] is a research toolbox developed at Mälardalen Real-Time Research Center (MRTC). The main objective of SWEET is flow analysis, which computes flow-facts, i.e., information about loop bounds and infeasible paths in the program. The main technique to discover flow-facts is abstract execution [16]. Abstract execution is a form of context-sensitive abstract interpretation, because it uses a symbolic execution to produce context information for each loop iteration and function call. Instead of using the fixpoint engine of abstract interpretation, abstract execution executes the program in the abstract domain, merging the execution paths at certain points in the program. SWEET does not support LRA. It currently implements only the abstract domain of intervals.

[2] `www.mrtc.mdh.se/projects/wcet`

2.1.3 AbsInt - The aiT Tool

Developed by AbsInt[3], aiT is the main industrial product for WCET analysis. It consists of a set of binary executables analyzers, which take the intrinsic cache and pipeline behavior into account. Concerning semantic analysis, aiT uses a value analysis based on intervals [13] to compute safe ranges of values for the program variables. aiT uses this information to determine loop bounds and detect infeasible paths. The approach towards computing loop bounds is not general, but it handles loop patterns. In order to gain precision, aiT pre-processes each loop by transforming its body into a function, in order to expose the iteration contexts. The key element in this transformation is to identify the loop index and to set it as a function parameter. Then, an interval analysis computes the ranges for all the loop variables. The loop transformation is based on loop patterns, which depend on the particularities of the architecture (e.g., parameter order) or on the loop structure (e.g., for-loops, triangular-loops, branch conditions). aiT is able to detect infeasible paths using the results of the value analysis, like conditions made infeasible because of the computed intervals.

2.1.4 oRange, the flow fact analyzer of OTAWA

OTAWA [5] is an academic toolbox, developed at IRIT (University of Toulouse), designed as a generic framework to develop static analyses for WCET computation. Although OTAWA implements several approaches to WCET computation, the one based on IPET is the most mature. OTAWA relies on an auxiliary tool, called oRange [9], to compute loop bounds. oRange analyses C code. As a first phase, oRange detects loop indices and constructs a normal form: a symbolic expression of the bound independently of the call context. In a second phase, by an abstract execution, a syntactic tree is built in function of a full or partial call context. It combines loop bounds and conditional expressions as numeric or symbolic expressions. Finally, the tree is computed in the full context in order to produce a file in the specific flow-facts format FFX [42].

2.2 Some experiments

In order to evaluate the capabilities of these tools to detect infeasible paths, we have applied each of them to programs containing various situations of semantic infeasibility. These situations are given in Figure 2:

- Example 1 is a case where simple pattern-based method may fail, since constant propagation is needed.
- Example 2 may be a problem for pattern-based methods for finding iteration numbers, since the apparent index x is modified.
- Example 3 is our introductory example of §1.1.
- Example 4 is a fragment of code generated by the SCADE[4] compiler, from a design manipulating arrays. On one hand, the loops are exited from inside, which complicates the evaluation of iteration numbers. On the other hand, the third loop is unreachable because of some non-trivial arithmetic conditions.

Table 1 summarizes the results of the tools on these examples. On Example 1, Chronos is unable to detect dead code[5]. Example 2 is correctly analyzed only by SWEET, because it unrolls

[3] AbsInt GmbH `www.absint.com/ait/`

[4] `www.esterel-technologies.com/products/scade-suite`

[5] We used the available version of Chronos. Some additional work has been done that complement the infeasible path analysis [6, 37], which is not part of the available version.

```
y = 2 ; x = y ;
if (x>3){// unreachable
  ...
}
```

Example 1

```
x = 0 ;
while (x<10){
  ...
  if (C1) {
    x = ...
    ...
  }
  x++ ;
  // iteration nbr. may be ≠ 10
}
```

Example 2

see example §1.1

Example 3

```
max_i = 3 ; s1 = 0;
for (i=0 ; i<10 ; i++) {
  s1 = s1 + t[i] ;
  // each t[i] assumed to be >5
  x = i+1 ;
  if (!(i<max_i)) break ;
  // intended for 4 iterations
}
int s2 = s1 ;
if (x < 6)
  for (i=0 ; i<10 ; i++){
    s2 = s2+2 ;
    if (!(i<5)) break ;
    // intended for 6 iterations
  }
int f = 0 ;
if (2*s1 < s2)
  // unreachable, since s1>20 and s2=s1+10
  for (i=0 ; i<10 ; i++){
    f = f+2 ;
    if (!(i<4)) break ;
  }
```

Example 4

Figure 2 Various cases of semantic infeasibilities.

Table 1 Results of tools on programs of Figure 2.

	Chronos	SWEET	oRange	aiT
Example 1	-	✓	✓	✓
Example 2	-	✓	-	-
Example 3	-	-	-	-
Example 4				
nbr. 1st loop	-	4	10	4
nbr. 2nd loop	-	5	10	5
dead code	-	-	-	✓

loops. None of the tools is able to find the property of Example 3. On Example 4, Chronos requires manual annotations for loop bounds; oRange estimates that both loops are iterated 10 times; SWEET and aiT find the exact loop bounds; only aiT detects dead code.

In this paper, we propose a method and a tool-chain that is able to discover the infeasible paths of these 4 examples, namely, infeasible paths that depend on a semantic analysis and that may concern distant program points.

2.3 Other approaches

An extended state of the art related to semantic analyses for WCET estimation can be found in [1] and a general survey of infeasible path detection in [10]. We complement them with some more recent publications.

Several recent works make use of SMT solvers [23, 37]. The idea is to ask the solver if the worst-case path obtained by the ILP solver is feasible. Whenever the path is infeasible, a corresponding constraint is added to the ILP. As in our approach, adding constraints does not always mean that the WCET is refined (2 paths may have the same WCET). In [18], the whole

path analysis is done through SMT solving instead of ILP: infeasible path analysis and worst-case path analysis are merged in one step. Path execution time is expressed as an SMT problem, and the question asked is no longer "is this path feasible?", but "is there a feasible path longer than K?", where K is a given constant (which is adjusted, e.g., by binary search). In [32, 35], a similar approach is taken, by asking this question to a bounded model checker.

3 Used techniques and tools

This section presents the existing techniques and tools used in our prototype: OTAWA implements the classical IPET-based WCET evaluation, and PAGAI performs Linear Relation Analyis.

3.1 WCET evaluation with OTAWA

The WCET estimation work-flow (Figure 3) involves a compiler, a Linear Program solver, and two tools from the OTAWA toolbox: oRange and owcet.

- Compilation: the source C code is compiled by a third party tool; for this experiment, we use a cross compiler from the GNU Compiler Collection (arm-elf-gcc 4.4.2), but other compilers can be used, provided that it produces ELF code (Executable and Linkable Format), with debugging information in DWARF format.
- oRange is a *data flow analyzing tool*, dedicated to the discovery of *loop bounds*. Bounds are stored in the OTAWA *flow facts* format (FFX).
- owcet is the OTAWA command dedicated to the WCET evaluation. The main steps of this tool, not detailed in Figure 3, are:
  - the construction of the control-flow graph (CFG) of the object code; during the construction, and thanks to debugging information, basic blocks (BB) are associated (if possible) to lines in the source program; thanks to this correspondence, the loop bounds computed by oRange are translated into control flow constraints in the CFG. The annotated CFG can be dumped in a file, allowing other tools to exploit it.
  - the micro-architectural analysis, which associates a local WCET estimation with each BB of the CFG.
  - the construction of the Integer Linear Programming (ILP) system; as in the introduction example (§1.1) the resulting system gathers (1) structural constraints (CFG structure), (2) loop bounds constraints (from oRange flow facts) (3) the objective function to be maximized (sum of BB counters weighted by their local WCET).
- ILP solver: the ILP system is then solved by a third-party tool; OTAWA integrates and uses LP_SOLVE[6] (any other equivalent tool can be used).

3.2 Linear Relation Analysis with PAGAI

3.2.1 Principles of LRA

Linear Relation Analysis [8] is a classical program analysis, based on abstract interpretation [7]. It is able to discover, at each control point of a sequential program, a conjunction of linear relations (equalities and inequalities) invariantly satisfied by the numerical variables at this point. Classical algorithms are used to propagate linear systems over the statements of the program. Several causes may result in information loss:

[6] `web.mit.edu/lpsolve/doc`

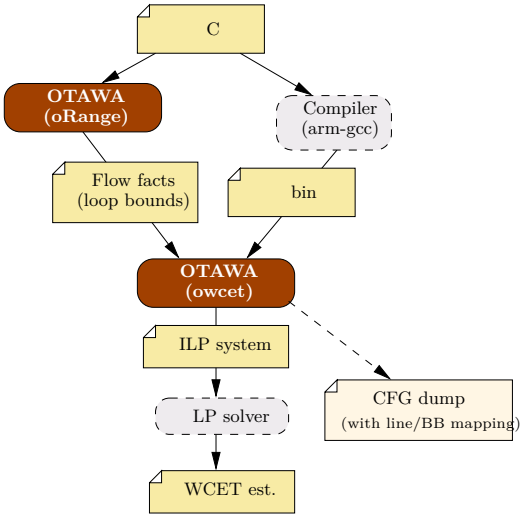


Figure 3 Otawa WCET estimation work-flow.

- the analysis safely ignores non-linear expressions in assignments and tests;
- the analysis performs a *convex hull* at control path junctions, instead of propagating the disjunction of incoming information. It means that the propagated value is the most precise conjunction of linear relations implied by both incoming systems;
- to avoid infinite propagation along loops, the classical *widening-narrowing* method is applied to guess a safe approximation of the limit. Note that, unlike in symbolic execution [23] or SMT methods [18], loops are not unrolled.

3.2.2 Applying LRA to our example

We do not detail further the techniques applied in LRA, and refer the reader to the bibliography. We just show the main steps of the analysis of our example of Figure 1. Let us consider the control point at the entry of the while loop. The first step of the analysis straightforwardly computes the first iterate:

$$x = i = \alpha = \beta = \gamma = 0$$

Its propagation through the loop body provides, with a convex hull at the end of the conditional:

$$i = \alpha = \beta = 1\ ,\ x = \gamma\ ,\ 0 \leq \gamma \leq 1$$

Now a convex hull with the first iterate gives the second iterate at the entry of the loop:

$$i = \alpha = \beta\ ,\ 0 \leq \alpha \leq 1\ ,\ x = \gamma\ ,\ 0 \leq \gamma \leq \alpha$$

Instead of continuing the iterations, a first widening/narrowing step is performed (using "lookahead widening" [14]), which provides:

$$i = \alpha = \beta\ ,\ x = \gamma\ ,\ 0 \leq \gamma \leq \alpha \leq 100\ ,\ \gamma \leq 10$$

Now, the "else" branch of the test `x<10` becomes feasible, and a second widening/narrowing step is performed, providing:

$$i = \alpha\ ,\ x = \gamma\ ,\ \beta + \gamma \leq \alpha + 10\ ,\ \beta \leq \alpha\ ,\ \gamma \leq \alpha \leq 100$$

which is found invariant after one more propagation. Propagated to the end of the program, it becomes:

$$i = \alpha = 100\ ,\ x = \gamma \leq 100\ ,\ \beta + \gamma \leq 110$$

3.2.3 LRA and loop bounds

It may happen, like in the previous example, that LRA discovers a bound to a loop counter, thus providing an essential information for WCET evaluation. However, finding loop bounds is *not* our main goal in this work, as the method is intrinsically unable to discover *non linear* relations, which drastically limits its capability to find loop bounds. As a matter of fact, in presence of nested loops, the number of executions of the body of the innermost loop is not linear in the constants of the program. For instance, in the program fragment "`for(i=0;i<n;i++){for(j=i;j<n;j++){...}}`" the number of executions of the body of the innermost loop is $n(n+1)/2$, which cannot be found by LRA.

The LRA method must then be used together with some other method able to bound nested loops. We can use existing tools such as oRange that comes with OTAWA, or more basically user-given bounds, given as pragmas in the code.

There exist also approaches based on polyhedra manipulation to find loop bounds, such as the one proposed in [38, 30]: it consists in building a polyhedral upper approximation P of the iteration domain, i.e., the set of possible valuations of loop counters (in the previous example, $P = \{(i,j) \mid 0 \leq i \leq j \leq n-1\}$). Under realistic assumptions concerning the determinism of the program, the number of executions of the innermost loop is bounded by the number of integer points in P, and algorithms are available to compute this number. Notice that LRA can be combined with this approach, since it can discover linear invariants reducing the iteration domain, thus improving the precision of the result. Notice also that LRA can deal with parameters (symbolic bounds, like n in our example), an issue specifically addressed by [38].

3.2.4 The PAGAI prototype analyzer

Several tools performing LRA are available ([2, 12, 19, 20] to cite a few). Here, we use the PAGAI prototype analyzer, which implements the basic LRA together with recent improvements like "lookahead widening" [14] and SMT-based "path focusing" [19]. PAGAI analyses LLVM code [24] produced from a C program (thanks to Clang[7]), and is able to return discovered properties at the C level. PAGAI may be used with other abstract domains than general linear systems – like octagons [33] – thanks to the common interface APRON [21].

4 Adding and tracing counters

4.1 The proposed workflow

Figure 4 shows the proposed workflow for the experiment. It involves two existing components: timing analysis with OTAWA (left) and program analysis with PAGAI (right). Two new tools have been developed to complete the workflow: a front-end (top, *Instrumentation*), which produces the input for the analyzers (OTAWA and PAGAI), and a back-end (*ILP translation & merge*), which gathers the results into a more constrained ILP system, and obtains a possibly enhanced WCET estimation.

[7] `http://clang.llvm.org/`

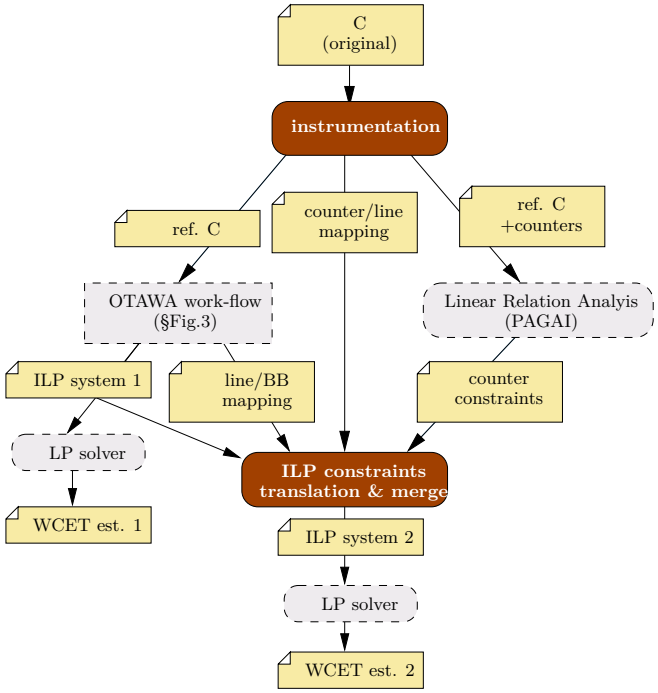


Figure 4 Instrumentation and analysis workflow.

These tools are detailed in this section. We illustrate the successive steps of the method by detailing the processing of an example program, called `lcdnum.c`, extracted from TacleBench programs suite [15]. The main program is given in Figure 5. It calls a function `num_to_lcd`, the execution time of which is taken into account by OTAWA.

4.2 Instrumented program version

The goal of the front-end ("instrumentation", Figure 4, top) is to produce, from the original C code, a reference C program. Some semantics preserving transformations of the source code are necessary or advisable, in order to use properly the analyzers, and trace the information between them.

- Some transformations are purely lexical, and do not change the program structure: because the standard ELF/DWARF traceability mechanism is line-based, line breaks are introduced to isolate each atomic statements on its own line.
- Some transformations that modify the control structure are necessary because of the limitation of the analyzers. For instance, a single-return statement per function is mandatory for exploiting the results of PAGAI: this unique control point is the place where counter invariants actually express properties on the whole execution of the function. Other transformations are required because of the limitation of both OTAWA and PAGAI: the control structure (CFG) must be statically known, which forbids dynamic computation of program pointers. In particular, "switch/case" statements must be rewritten into a static control structure based on "if" and "goto" statements.
- Another transformation is desirable in our case: the current version of PAGAI does not handle inter-procedural analysis. In order to exploit the plain capacity of this tool to find invariants, a light-weight solution is to inline function calls at the source level. This transformation is

indeed hardly admissible in real-life, but it must be seen here as a "trick" to reach our goal (study the ability of LRA to detect infeasible executions).

The front-end produces the reference C code in two flavors.

- The reference C code with counters (Figure 4, right) is instrumented with auxiliary counters, in the same manner as in the introductory example (§ 1.1). The present version introduces a counter for each sequential block in the program control flow. However, some strategy could be used to reduce the number of counters by targeting blocks that are more likely to have an influence [43].
- The reference C code without counters (Figure 4, left) is the same code, where all lines related the counters (declaration, initialization and incrementation) have been commented out. This method ensures a semantic equivalence between the programs analyzed by OTAWA and PAGAI: since they only differ on the side-effect-free local variables, these programs are naturally input/output equivalent. Moreover, at least at the source level, the two programs are also structurally equivalent: a block in the reference C code is executed if and only if the corresponding block (marked with a counter c) is executed in the reference C program with counters. This property becomes false in general at the binary level, since the C compiler may modify the control structure: this well-know problem of traceability is discussed later.
- An auxiliary file is generated, that contains the mapping between each counter and its corresponding source line in the reference C code.

▶ **Example 1.** Applied to our example program (Figure 5), our instrumentation front-end calls the C preprocessor, eliminates the multiple returns and switches (only within **`num_to_lcd`**, not shown), and produces the reference C programs. The first one (without counters) is shown on Figure 6; the second (not shown) is exactly the same with uncommented lines involving counters. An auxiliary file (not shown) simply lists the pairs "counter/line" (e.g., **`(cptr_main_1,144)`**, **`(cptr_main_2, 147)`**).

The first version is provided to OTAWA. Loop bounds computation by oRange is optional, which allows us to check if PAGAI is able to find them on its own. OTAWA calls the gcc compiler (here with -O0 optimization level), builds the CFG of the object code, performs the micro-architectural analysis, and builds the ILP problem.

PAGAI is applied to the second version of the program, and returns the following invariants:

```
-10+cptr_main_2 = 0
-10+cptr_main_4 = 0
5-cptr_main_3 >= 0
```

The first equation finds the exact loop bound (which may also be found by oRange). The second equation is structural (from the shape of the source CFG, cptr_main_2 and cptr_main_4 are equal). The third property is new, and expresses, in particular, that the function **`num_to_lcd`** is called at most 5 times.

4.3 Tracing back the counters

The back-end ("ILP constraints translation & merge", Figure 4, bottom) gathers the information coming form OTAWA and PAGAI:

- Thanks to the counter/C-line mapping provided by the front-end, and the C-line/binary-BB mapping provided by OTAWA (through the ELF/DWARF information), a counter/BB mapping is built. Note that this mapping is partial, and deliberately pessimistic: depending on the compilation process, it may happen that a counter is associated either to zero or to

```
unsigned char num_to_lcd( unsigned char a ) ;

volatile unsigned char IN = 120;
volatile unsigned char OUT;
int main( void ) {
  int i;
  unsigned char a;
  for( i=0; i< 10; i++ ) {
    a = IN;
    if(i<5) {
      a = a &0x0F;
      OUT = num_to_lcd(a);
    }
  }
  return 0;
}
```

Figure 5 The initial `lcdnum.c` program.

```
133 int main(void) {
134   int i ;
135   unsigned char a ;
136   unsigned char tmp ;
137   int __retres4 ;
138   //int cptr_main_1 = 0;
139   //int cptr_main_2 = 0;
140   //int cptr_main_3 = 0;
141   //int cptr_main_4 = 0;
142   //int cptr_main_5 = 0;
143   //cptr_main_1 ++; #line 144
144   i = 0;
145   while (i < 10) {
146     //cptr_main_2 ++; #line 147
147     a = (unsigned char )IN;
148     if (i < 5) {
149       //cptr_main_3 ++; #line 150
150       a = (unsigned char )((int )a & 15);
151       tmp = num_to_lcd(a);
152       OUT = (unsigned char volatile )tmp;
153     }
154     //cptr_main_4 ++; #line 155
155     i ++;
156   }
157   //cptr_main_5 ++; #158
158   __retres4 = 0;
159   return (__retres4);
160 }
```

Figure 6 The reference `lcdnum.c` program.

Table 2 Mapping between counters and blocks.

line number(s)	block(s)	reliable	counter
136,144	1	yes	cptr__main__1
145	1;2	no	
147;148	4	yes	cptr__main__2
150;151;152	5	yes	cptr__main__3
155	6	yes	cptr__main__4
158;159;160	3	yes	cptr__main__5

several binary basic blocks. In this case, the counter is simply ignored: only counters that are associated to one single BB are retained.

▶ **Example 1** (cont.). Table 2 shows the mapping between counters and blocks that is built by our back-end.

- The linear constraints on the retained counters are then translated literally into linear constraints on BB, and added to the basic ILP system provided by OTAWA.

  ▶ **Example 1** (cont.). The translation of the constraints discovered by PAGAI is the following:

```
x4_main = 10;
x6_main = 10;
x5_main <= 5;
```

- At last, both systems are solved and the corresponding estimations can be compared.

  ▶ **Example 1** (cont.). After a second call to LP_SOLVE, the final result is printed:

```
Estimation WITHOUT PAGAI: 1540
Estimation WITH PAGAI: 945
```

4.4 Traceability and optimization

In our framework, traceability is the ability to relate execution paths in the binary code (bin. CFG) to execution paths in the source code (source CFG).

Some optimizations performed by the compiler may strongly modify the control structure and thus alter traceability: loop unrolling, block replication, out-of-order execution. This is why most of the related works assume no compiler optimization to guarantee a perfect matching between the two CFGs.

However forbidding optimization is not satisfactory in real-time domains, where execution times have to be predictable, but also short. For a standard compiler like gcc, the observed speed-up between no optimization (-O0 option) and a standard level of optimization (-O1) is around two.

The most satisfactory solution would be a compiler that provides a precise traceability even in case of CFG optimization. Some work has been done to design and/or adapt the compilation process for this purpose, for instance [26, 31, 22].

Unfortunately, off-the-shelf standard compilers such as gcc hardly provide a precise and reliable information in case of CFG optimization. The idea is then to use the compiler options in order to forbid (as far as possible) CFG transformations, but still allow other optimizations, in particular those that concern data management.

The gcc compiler proposes numerous options to control optimizations, but there hardly exists a comprehensive and exhaustive description of their effects and inter-dependencies. For this experiment, we have empirically defined a *customized level* (called `C0` in the sequel). We started from the standard `-O1` level, and removed about 20 individual optimizations using the `-fno` directive (see appendix B). We cannot guarantee that this customized level will preserve the CFG for all programs, but the method is safe: as explained in Section 4.3, a counter (and then a source code line) that is not associated to exactly one basic block of the binary code is simply ignored. As a consequence, the only risk is to lose information that would have made the WCET estimation tighter. Note that this statement suppose that the gcc debugging information is reliable, which is indeed unprovable, but *empirically* reasonable.

▶ **Example 2.** When applying the `CO` method to our running example, we get 100% traceability. As a consequence, the interesting counter property (`5-cptr_main_3 >= 0`) can still be translated into a BB constraint (`x11_main <= 5;`) leading to the final result:

```
Estimation WITHOUT PAGAI: 641
Estimation WITH PAGAI: 421
```

On this example, we observe that code optimization leads to an initial WCET estimation 2.4x smaller (641 vs 1540). The traceability is preserved and the improvement due to the counter analysis is of the same order (34.3% vs 38.6%).

5 Experiments

5.1 Benchmarks

We tested our approach on programs from the TacleBench [11], a set of C programs widely used in the WCET community. [8] A first check has been made to retain only purely sequential programs that compile "out of the box": 53 applications of the 58 in the TacleBench[9]

For each program, we try to estimate the WCET of all functions appearing in the code, including the top-level one (main). For each function, inner function calls are recursively inlined at the C level (see Section 4.2). Recursive functions are rejected during this step, and not considered for WCET analysis.

Our goal is to study the influence of our counter-based method (Fig. 4) on a classical estimation (Fig. 3). A prerequisite is therefore that a *reference* estimation exists; hence the programs for which the basic WCET estimation fails are not selected. The OTAWA estimation may fails because of unsupported programming features (pointer arithmetics), or because the analysis does not terminates before a chosen timeout (2 hours).

After this initial selection, 589 functions (out of 639) from the 53 programs of the TacleBench suite are retained.

5.2 Experimental setup

The proposed framework as presented on Fig. 4 has numerous parameters (C code instrumentation, linear analysis tuning, compiler optimization etc.) leading to a combinatorial numbers of possibilities. For this systematic experiment, we focus only on two kinds of parameters: those that influence the precision of linear analysis, and those that influence the traceability. The other parameters are fixed once and for all as follows:

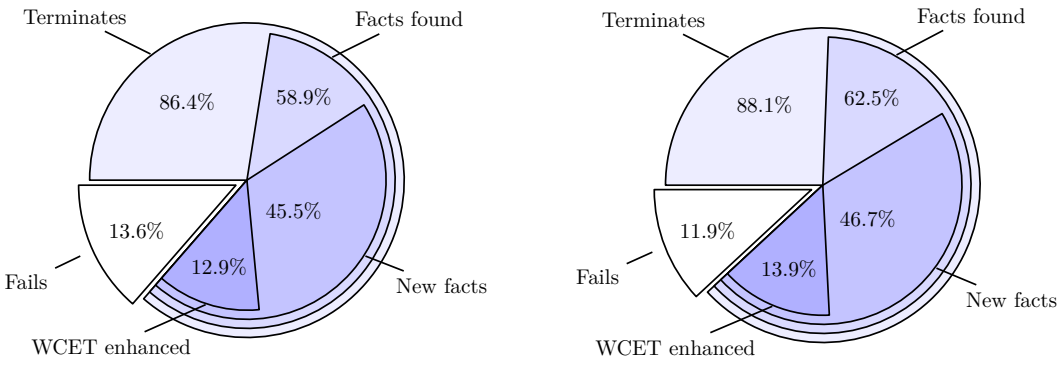
OTAWA hardware model: our goal is not to bench or "stress" OTAWA in terms of hardware. We only want it to give an initial IPET system in which we will insert flow facts discovered via LRA. In order to maximize the number of test benches for which OTAWA gives an initial ILP in reasonable time, we consider a very simple, cache-free, ARM-based architecture.

Misc. CFG transformations: some CFG transformations are necessary, due to limitations of OTAWA (switch statements not supported) and /or PAGAI (multiple return statements). This transformations are performed using the CIL library [34].

Inlining: because the current version of PAGAI has limited support for inter-procedural analysis, function calls are systematically inlined. This transformation is also implemented using the CIL library. This method improves the precision of the analysis, but makes the analysis much more costly in time and memory.

[8] The material necessary for reproducing the experiment is freely available at `https://gricad-gitlab.univ-grenoble-alpes.fr/verimag/reproducible-research/LRA4w7`.

[9] The 5 missing applications are OS and/or architecture dependent.



(a) polyhedra domain **(b)** octagon domain

Figure 7 LRA analysis statistics on 589 functions, for the two relational abstract domains.

Loop bounds: as explained in 3.2.3, our method is intrinsically unable to bound nested loops, so a complementary method is necessary to find loop bounds. For this purpose, we can use oRange, but it appears that the CFG transformations performed using CIL strongly alters its performance[10]. In order to maximize the size of the benchmark we thus systematically exploit, when available, the user pragmas given in source code. Nevertheless, we made a complementary experiment, without using pragmas nor oRange, in order to identify the cases where LRA is sufficient to bound the execution time.

5.3 Lessons learnt

This section presents the lessons learnt form the experiment, by focusing on several points: the ability of the linear analysis to discover "flow facts", and hopefully to enhance the WCET estimation; the influence of the abstract domain on the analysis; the ability of linear analysis to discover loop bounds, and finally the influence of compiler optimizations on traceablity.

5.3.1 Linear analysis and flow facts discovery

When traceability allows it, the constraints discovered by linear analysis are directly translated into *flow facts* giving information on the (im)possible execution paths. These flow facts may be useless if they are redundant with the structural constraints, otherwise they are *new facts*, giving non trivial information on the execution paths. However, even new facts can be useless if they do not concern the worst case execution path. A utility has been developed to check whether the facts discovered by LRA analysis are new or not. Each fact is checked by adding its negation to the set of structural constraints: the fact is redundant if and only if the system becomes infeasible.

Figure 7 gives statistic on the behavior of the LRA method, for the two relational domains (octagon and polyhedra). Let us focus on the polyhedra case first (a). The PAGAI tool terminates for 509 cases out of 589 (86.4%); for the missing cases (13.6%), it runs out of resources in memory or time. Flow facts are found in 347 cases, and at least one fact is new for 268 ones; finally, new facts lead to a WCET improvement for 76 cases. Statistics are similar for the octagon domain, except that it terminates more often: this explains why the WCET is enhanced more often with octagons, even if this domain is less precise.

[10] The CIL tool normalizes the code by using only unbounded *while* and *break* statements, that are badly handled by oRange. However oRange performs well for the original programs, made of human-written *for* loops.

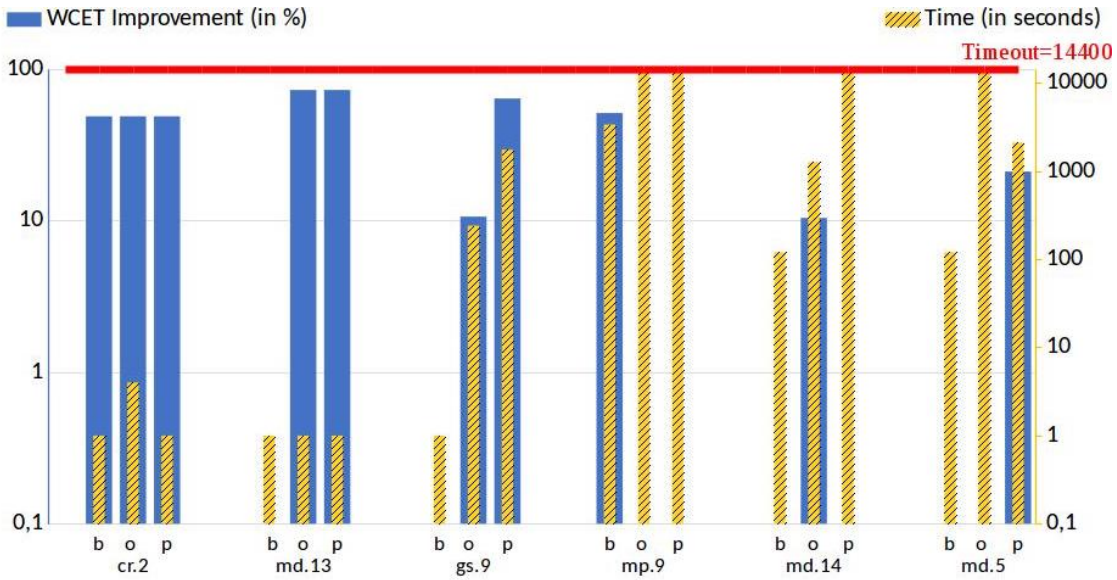


Figure 8 WCET improvement and analysis time depending on abstract domains (b=box, o=octagons, p=polyhedra).

Table 3 Some WCET improvement results (full table page 24).

Ref	Initial WCET	Box			Octagons			Polyhedra		
		Δ	Imp^t	Time	Δ	Imp^t	Time	Δ	Imp^t	Time
cr.2	227K	111K	**48.7**	<1s	111K	**48.7**	4s	111K	**48.7**	1s
md.13	2648	0	0.0	<1s	1920	**72.5**	<1s	1920	**72.5**	<1s
gs.9	6934	0	0.0	1s	738	10.6	4m	4428	**63.8**	29m
an.0	466M	0	0.0	4s	4M	0.8	7m	115M	**24.6**	1m
mp.9	52M	27M	**51.1**	56m	-	-	-	-	-	-
md.14	51K	0	0.0	2m	5K	**10.4**	21m	-	-	-
md.5	13M	0	0.0	2m	-	-	-	3M	**21.0**	35m

A possible conclusion is that LRA, when it works, is actually good at finding non redundant semantic facts (more than half of the time, when it terminates), but that those facts do not necessarily lead to a WCET improvement (about 15% of the termination cases).

5.3.2 Abstract domains

The main goal of the experiment is to observe the influence of the linear analysis on the WCET estimation. The linear analysis performed by PAGAI is parameterized by the choice of an abstract domain to represent the possible values of the counters. Two domains proposed by PAGAI are *relational*, and thus are likely to express relations between our counters and the original variables in the programs:

- The polyhedra domain is the most precise since it can handle any linear relation, and its algorithmic cost is exponential in the worst case.
- The octagon domain handles intervals and bounded pairwise sums or differences. It is less precise, but has a polynomial cost in the worst case: $O(n^3)$ in time, and $O(n^2)$ in space.

To be exhaustive, we also consider the domain *box*, which handles only intervals. Since this domain is non-relational, it is intrinsically unable to relate our additional counters to the program variables. The flow facts that can be discovered with the box domain are thus limited (basically, counters stuck down to zero, which correspond to dead code).

The WCET estimation is improved by at least one domain for 90 functions. The gain ranges from negligible (0.1%) to interesting (around 10%) or even huge (more than 50%). We limit here the comments to the cases where the enhancement is greater 0.8%. The detailed results for these 60 cases are given in appendix (table 6, page 24), and a selection of typical cases is given in table 3.

The experiment gives some interesting information:

- The interest of the *box domain* is very limited: it is an indirect way of performing constant propagation and dead code "pruning". Most of the time it gives no improvement (42 out of 60, e.g., md.13, gs.9). However, since it is the cheapest domain, it may give results when other domains fail (6 times, e.g., mp.9).
- When both *octagons* and *polyhedra* terminate, they often give the same result (34 out of 60 cases, e.g., cr.2, md.13). However there are some cases (12 out of 60, e.g., gs.9), where the expressiveness of polyhedra is actually useful (constraint involving 3 or more variables, and pairwise relations with non unit coefficients).
- In compliance with the theoretical complexity, *octagons* may terminates while *polyhedra* fails (7 cases, e.g., md.14). Nevertheless, there is also one case where *octagons* fail while *polyhedra* works (md.5). This is due to the fact that the cost of octagons is almost always cubic in the number of variables, while the exponential cost of polyhedra is rarely reached in practice.

5.3.3 Loop bounds

LRA is intrinsically limited to the discovery of single loop bounds (cf. 3.2.3). We made a complementary experiment to check if and when LRA actually finds such loops. For this experiment, we only consider the short-list of programs from Table 6 where PAGAI terminates when using a relational domain (octagon or polyhedra); as a matter of fact, using the *box* domain is irrelevant since it can't find any loop bound other than 0.

For these 54 programs, we have:

- computed the *loop level*, which is maximal depth of nested loops appearing in the program (0: no loop at all, 1: only single loops, 2 or more: nested loops);
- launched our tool without using oRange nor user-pragmas. The LRA analysis is performed twice: with the octagon and the polyhedra domain, and we keep only the best result.

Table 7 (page 25) lists the results; the column "pagai" simply indicates if the analysis give a bounded WCET, since the WCET value is, in this case, the same as the one in Table 6.

There are 10 test cases that are loop-free, and thus with no bounds to found. There are 25 programs with only single loops (level=1); these are the cases where PAGAI is supposed to find bounds, and it actually does it for most of the cases (19 out of 25). In fact, PAGAI finds the bounds for all loops that are semantically guarded by a counter condition, that is, *for* loops or equivalent. The cases where PAGAI does not find bounds are those where the loop is guarded by a *points-to* condition (e.g., `while (*p++)`).

We expected PAGAI not to bound any program with a loop level greater than 1, which is the case except for one program (ex.2). In fact this example is a "false counter-example": the loop depth is *syntactically* 2, but the inner-loop appears in a branch which is never executed. The loop depth is then *semantically* 1.

Table 4 Impact of compiler optimizations on WCET and LRA

	O0			CO						
Ref	Initial WCET	Best WCET	Best Imp^t	Opt. speedup	Initial WCET	Best WCET	Best Imp^t	Traceability		
md.13	2648	728	72.5	3.3x	791	215	72.8	100%	of	2
an.0	466M	351M	24.6	3.0x	157M	116M	25.9	100%	of	44
cr.2	227K	116K	48.7	2.3x	97K	50K	48.7	41%	of	24
md.5	13M	10M	21.0	3.4x	4M	3M	15.0	80%	of	40
ex.2	278K	224K	19.2	1.3x	218K	218K	0.0	46%	of	13

5.3.4 Optimization level and traceability

The main focus of this work is the influence of linear analysis on the precision of the WCET estimation. Nevertheless, since analysis is performed at the C level, the problem of the traceability between the C and the binary code must be considered. Forbidding any optimization is not an option in real-time domain. We argue that a well-chosen set of optimizations can lead to a reasonable compromise between traceability and program speed-up.

For all functions that give some enhancement on the non-optimized code, we run the experiments using the custom optimization (CO) level defined in 4.4. Since the counter analysis is completely independent to the compilation method, the linear relations found are the same, and the ability to enhance the WCET estimation is only due to traceability.

The detailed results of this experiment are given in appendix (table 8, page 26), and a selection of typical cases is given in table 4. The table gathers the results obtained with the non-optimized binary code O0, and the optimized one CO. For each optimization level, the table gives:

- the *Initial WCET*, in CPU cycles, computed by OTAWA,
- the *Best WCET*, enhanced thanks to the properties discovered with PAGAI, with some abstract domain,
- the corresponding **Improvement** percentage.

The table also shows the *Optimization speed-up*, which is the ratio between the initial O0 and the initial CO estimation, i.e., it measures the gain obtained just because of the compilation, before applying the counter method. Finally, for CO compilation, the table gives an information on the *Traceability*: the percentage of counters introduced for LRA at C level, that are actually associated to some basic block, at binary level. Traceability in the O0 mode is not shown in the table as it is always 100%.

The interesting information given by the experiment are:

- Even if the CO level is very limited (subset of O1 level, and a fortiori of O2), it generates a fairly optimized code: the speed-up is mostly between 2x and 4x.
- In most of the cases (53 out of 60) traceability is 100%, and one can observe an enhancement due to LRA similar to the one obtained with O0 code. Indeed, this improvement is obtained on the CO initial WCET, which is already much smaller than the one obtained for the non-optimized code (e.g., md.13, an.0).
- In some cases, traceability is partly lost, but remain sufficient to enhance the estimation (4 cases, e.g., cr.2, md.5).
- Finally, for 3 cases, partial traceability leads to no enhancement (e.g., ex.2).

6 Conclusion and future work

Linear Relation Analysis is a powerful technique to discover invariant linear relations between numerical variables of a program. On the other hand, the classical evaluation of WCET using Implicit Path Enumeration Technique is based on expressing the WCET as the solution of an Integer Linear Program, the variables of which are counters associated with the basic blocks of the program. So, the idea of adding these counters as auxiliary variables in the program, and using the results of LRA as semantic flow-facts to be added to the ILP, is rather natural. Our goal, in this paper, was to conduct a light-weight experiment – by combining existing tools – to evaluate the benefits of the approach. Secondarily, such an experiment raised the question of traceability, since semantic flow-facts are discovered on the source program, while the WCET is evaluated on the executable code. The conclusion of this experiment on public benchmarks is manyfold:

- LRA finds new semantic facts in many examples (46%), but many of these new facts do not influence the evaluated WCET. However, the WCET is improved on a significant subset (almost 14%) of the examples, and the improvement is often interesting.
- the traceability problems can be safely dealt with, using the debugging information provided by the compiler; this is the case even in the presence of strong compiling optimizations, as long as these optimizations do not modify too much the control structure of the program.

This work could be continued in several directions.

- It would be interesting to limit the number of counters, as the cost of LRA can be exponential in the number of variables. Of course, counters which are structurally related by flow equations can be saved, but their cost is low in polyhedra computations (they are linked to each other by equations). An appealing idea would be to introduce counters on the branches of a conditional, only when these branches appear to have strongly different execution times, a measure that is roughly available after the micro-architectural analysis [43].
- Existing LRA analyzers (like PAGAI) are generally not inter-procedural, which forced us to inline the procedures in our experiments. An inter-procedural version of LRA must be studied to solve this problem. The *relational* nature of LRA is surely an advantage, since a procedure can be associated a summary as an input-output relation. Summaries of called procedures can then be used in the caller, in a bottom-up fashion.
- Traceability is still a concern, which would benefit from a better cooperation of the compiler [25].

References

1 Mihail Asavoae, Claire Maiza, and Pascal Raymond. Program Semantics in Model-Based WCET Analysis: A State of the Art Perspective. In *13th International Workshop on Worst-Case Execution Time Analysis, WCET 2013, July 9, 2013, Paris, France*, volume 30 of *OASICS*, pages 32–41. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.

2 Roberto Bagnara, Elisa Ricci, Enea Zaffanella, and Patricia M. Hill. Possibly Not Closed Convex Polyhedra and the Parma Polyhedra Library. In M. V. Hermenegildo and G. Puebla, editors, *9th International Symposium on Static Analysis, SAS'02*, Madrid, Spain, September 2002. LNCS 2477. doi:10.1007/3-540-45789-5_17.

3 Gogul Balakrishnan and Thomas W. Reps. DIVINE: DIscovering variables IN executables. In *Verification, Model Checking, and Abstract Interpretation, VMCAI 2007*, pages 1–28, Nice, France, January 2007.

4 Gogul Balakrishnan, Thomas W. Reps, David Melski, and Tim Teitelbaum. WYSINWYX: what you see is not what you execute. In *Verified Software: Theories, Tools, Experiments, VSTTE 2005*, pages 202–213, Zurich, Switzerland, October 2005.

5 Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: An open toolbox for adaptive WCET analysis. In *SEUS*, 2010.

6 Duc-Hiep Chu, Joxan Jaffar, and Rasool Maghareh. Precise Cache Timing Analysis via Symbolic Execution. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, 2016.

7 Patrick Cousot and Radia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages, POPL'77*, Los Angeles, January 1977.

8 Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th ACM Symposium on Principles of Programming Languages, POPL'78*, Tucson (Arizona), January 1978.

9 Marianne de Michiel, Armelle Bonenfant, Hugues Cassé, and Pascal Sainrat. Static loop bound analysis of C programs based on flow analysis and abstract interpretation. In *IEEE Int'l Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2008.

10 Sun Ding, Hee Beng Kuan Tan, and Kaiping Liu. A Survey of Infeasible Path Detection. In *Proceedings of the 7th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2012), Wroclaw, Poland, 29-30 June, 2012.*, pages 43–52, 2012.

11 Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sorensen, Peter Wägemann, and Simon Wegener. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In *16th International Workshop on Worst-Case Execution Time Analysis, WCET 2016, July 5, 2016, Toulouse, France*, pages 2:1–2:10, 2016.

12 Paul Feautrier and Laure Gonnord. Accelerated Invariant Generation for C Programs with Aspic and C2fsm. In *Tools for Automatic Program AnalysiS (TAPAS)*, Perpignan, France, September 2010.

13 Christian Ferdinand, Florian Martin, Christoph Cullmann, Marc Schlickling, Ingmar Stein, Stephan Thesing, and Reinhold Heckmann. New Developments in WCET Analysis. In *Program Analysis and Compilation*, pages 12–52, 2006.

14 Denis Gopan and Thomas Reps. Lookahead widening. In *CAV'06*, Seattle, 2006.

15 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET Benchmarks: Past, Present And Future. In *Proc. of WCET*, pages 136–146, 2010.

16 Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis Using Abstract Execution. In *RTSS*, 2006.

17 Nicolas Halbwachs, Yann-Eric Proy, and Patrick Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, August 1997.

18 Julien Henry, Mihail Asavoae, David Monniaux, and Claire Maiza. How to compute worst-case execution time by optimization modulo theory and a clever encoding of program semantics. In *SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems 2014, LCTES '14*, pages 43–52, June 2014.

19 Julien Henry, David Monniaux, and Matthieu Moy. PAGAI: A Path Sensitive Static Analyser. *Electr. Notes Theor. Comput. Sci.*, 289:15–25, 2012.

20 François Irigoin, Pierre Jouvelot, and Rémy Triolet. Semantical Interprocedural parallelization: An overview of the PIPS Project. In *ACM Int. Conf. on Supercomputing, ICS'91, Köln*, 1991.

21 Bertrand Jeannet and Antoine Miné. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *Computer Aided Verification (CAV 2009), Grenoble, France*, pages 661–667, June 2009.

22 Raimund Kirner, Peter Puschner, and Adrian Prantl. Transforming flow information during code optimization for timing analysis. *Journal on Real-Time Systems*, 45(1-2), 2010.

23 Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. WCET squeezing: on-demand feasibility refinement for proven precise WCET-bounds. In *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, pages 161–170. ACM, 2013.

24 Chris Lattner and Vikram Adve. LLVM: a compilation framework fopr lifelong program analysis & transformation. In *CGO'04*, pages 75–86, Washington, DC, August 2004. IEEE Computer Society.

25 Hanbing Li, Isabelle Puaut, and Erven Rohou. Traceability of Flow Information: Reconciling Compiler Optimizations and WCET Estimation. In *22nd International Conference on Real-Time Networks and Systems, RTNS'14, Versailles, France, October 8-10, 2014*, 2014.

26 Hanbing Li, Isabelle Puaut, and Erven Rohou. Tracing Flow Information for Tighter WCET Estimation: Application to Vectorization. In *21st IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, page 10, Hong-Kong, China, August 2015. URL: `https://hal.inria.fr/hal-01177902`.

27 Xianfeng Li, Liang Yun, Tulika Mitra, and Abhik Roychoudhury. Chronos: A timing analyzer for embedded software. *Sci. Comput. Program.*, 69(1-3):56–67, 2007.

28 Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 16(12), 1997.

29 Björn Lisper. SWEET – a tool for WCET flow analysis. In *6th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISOLA)*, October 2014.

30 Paul Lokuciejewski, Daniel Cordes, Heiko Falk, and Peter Marwedel. A Fast and Precise Static Loop Analysis Based on Abstract Interpretation, Program Slicing and Polytope Models. In *Proceedings of the CGO 2009, The Seventh International Symposium on Code Generation and Optimization*, pages 136–146, Seattle, Washington, USA, March 2009.

31 Paul Lokuciejewski and Peter Marwedel. *Worst-Case Execution Time Aware Compilation Techniques for Real-Time Systems*. Springer, 2011. `doi:10.1007/978-90-481-9929-7`.

32 Ravindra Metta, Martin Becker, Prasad Bokil, Samarjit Chakraborty, and R. Venkatesh. TIC: a scalable model checking based approach to WCET estimation. In *Proceedings of the 17th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory for Embedded Systems, LCTES 2016, Santa Barbara, CA, USA, June 13 - 14, 2016*, pages 72–81, 2016. `doi:10.1145/2907950.2907961`.

33 Antoine Miné. The Octagon Abstract Domain. In *Proceedings of the Eighth Working Conference on*

Reverse Engineering, WCRE'01, Stuttgart, Germany, October 2-5, 2001, page 310, 2001. doi:10.1109/WCRE.2001.957836.

34 George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In R. Nigel Horspool, editor, *Compiler Construction*, pages 213–228, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

35 Pascal Raymond, Claire Maiza, Catherine Parent-Vigouroux, Fabienne Carrier, and Mihail Asavoae. Timing analysis enhancement for synchronous program. *Real-Time Systems*, pages 1–29, 2015.

36 Jordy Ruiz and Hugues Cassé. Using SMT Solving for the Lookup of Infeasible Paths in Binary Programs (regular paper). In *Workshop on Worst-Case Execution Time Analysis, Lund, Sweden, 07/07/2015*, pages 95–104. OASICs, Dagstuhl Publishing, July 2015.

37 Thomas Sewell, Felix Kam, and Gernot Heiser. Complete, High-Assurance Determination of Loop Bounds and Infeasible Paths for WCET Analysis. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Vienna, Austria, April 11-14, 2016*, pages 185–195, 2016. doi:10.1109/RTAS.2016.7461326.

38 Björn Lisper Stefan Bygde, Andreas Ermedahl. An Efficient Algorithm for Parametric WCET Calculation. *Journal of Systems Architecture*, 57(6):614–624, May 2011.

39 Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. Efficient detection and exploitation of infeasible paths for software timing analysis. In *DAC*, pages 358–363, 2006.

40 Stephan Thesing, Jean Souyris, Reinhold Heckmann, Famantanantsoa Randimbivololona, Marc Langenbach, Reinhard Wilhelm, and Christian Ferdinand. An Abstract Interpretation-Based Timing Validation of Hard Real-Time Avionics Software. In *DSN*, pages 625–632, 2003.

41 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst. (TECS)*, 7(3), 2008.

42 Jakob Zwirchmayr, Armelle Bonenfant, Marianne de Michiel, Hugues Cassé, Laura Kovács, and Jens Knoop. FFX: A portable WCET annotation language (regular paper). In *International Conference on Real-Time and Network Systems (RTNS), Pont-à-Mousson, 08/11/2012-09/11/2012*, pages 91–100, November 2012.

43 Jakob Zwirchmayr, Pascal Sotin, Armelle Bonenfant, Denis Claraz, and Philippe Cuenot. Identifying Relevant Parameters to Improve WCET Analysis (regular paper). In *Workshop on Worst-Case Execution Time Analysis, Madrid, 08/07/2014*, pages 91–100. OASICs, Dagstuhl Publishing, July 2014.

A Experiment Results

The material necessary for reproducing the experiment presented here is freely available at `https://gricad-gitlab.univ-grenoble-alpes.fr/verimag/reproducible-research/LRA4w7`.

Experiment was performed on 589 individual C functions extracted from the TACLeBench [11]. An improvement of the WCET estimation is observed for 90 functions (15% of the cases). This section details the results for the 60 cases where the improvement is greater than 0.8%.

Table 5 contains label definitions to ease and shorten the reference to the bench functions: the label (column 1), the source folder in the TACLeBench (column 2), and the function name (column 3).

Table 6 contains the experiment results using the gcc `-O0` compilation level. The first column holds the function label, and the second one holds the initial WCET estimation computed by OTAWA. The remaining columns hold information related to the improvement obtained (or not) with Linear relation analysis, using 3 different abstract domains: boxes (intervals), octagons and polyhedra. For each domain, the table gives the improvement in number of cycles (Δ) and percentage (Imp^t), and the time necessary to perform the LRA with PAGAI[11]. Numbers in bold highlight the best improvements among various methods (box, octagons, polyhedra). Empty cells ('-') mean that the corresponding case triggered the 2 hours timeout set for the experiment.

Table 7 gives information on the ability of PAGAI to discover loop bounds ; to obtain this table, the experiments are re-played without the help of any external method (neither oRange nor the user-given pragmas). For each program, the table gives its loop level (maximal depth of nested loops) and indicates wheter PAGAI finds a bounded WCET or not.

Finally, table 8 aims at observing the impact of compiler optimization on WCET estimation in general, and our method in particular. We consider two optimization levels: the standard `-O0` (no optimization at all), and the ad hoc customized `-O1` level (designed to limit CFG transformation and maximize traceability). Since the LRA analysis is performed at the C level, the flow facts discovered are the same whatever is the optimization level. A lack of improvement in the case of optimized code is then necessarily due to an "imperfect" traceability.

The first group of columns recalls the results optained with `-O0`; it only gives the best result, obtained for some abstract domain (refer to Table 6 for details). The second group gives information on the optimized code:

- the initial WCET estimation given by OTAWA, together with the corresponding *speed-up* factor which indicates how "faster" is the optimized code compared to the non-optimized one;
- the best WCET estimation (together with the improvement percentage) optained using PAGAI;
- the *traceability* ratio indicates how many counters introduced by our method are actually associated to some basic block in the binary code. With a traceability of 100%, we expect to observe an improvement percentage of the same order than the one obtained on the non-optimized code. Note that the traceability with the non-optimized code is not given since it is always 100%.

[11] Results obtained on an Intel(R) Xeon(R) CPU E5-2683 v3 @ 2.00GHz

Table 5 TacleBench functions Reference Labels.

Ref	Directory	Function Names
ad.6	sequential/adpcm_dec	adpcm_dec_logsch
ad.7	sequential/adpcm_dec	adpcm_dec_logscl
ad.14	sequential/adpcm_dec	adpcm_dec_uppol2
ae.7	sequential/adpcm_enc	adpcm_enc_logsch
ae.8	sequential/adpcm_enc	adpcm_enc_logscl
ae.10	sequential/adpcm_enc	adpcm_enc_quantl
ae.16	sequential/adpcm_enc	adpcm_enc_uppol2
am.12	sequential/ammunition	ammunition_bit_string_set
am.17	sequential/ammunition	ammunition_divide_unsigned_integer
am.18	sequential/ammunition	ammunition_divide_unsigned_integer_without_overflow
am.47	sequential/ammunition	ammunition_multiply_integer
am.49	sequential/ammunition	ammunition_multiply_unsigned_integer
am.50	sequential/ammunition	ammunition_multiply_unsigned_integer_without_overflow
am.68	sequential/ammunition	ammunition_unsigned_integer_remainder
an.0	sequential/anagram	anagram_AddWords
an.2	sequential/anagram	anagram_BuildWord
an.8	sequential/anagram	anagram_init
an.14	sequential/anagram	anagram_ReadDict
an.15	sequential/anagram	anagram_Reset
an.16	sequential/anagram	anagram_return
bs.0	kernel/bsort	bsort_BubbleSort
bs.3	kernel/bsort	bsort_main
bs.4	kernel/bsort	bsort_return
bs.5	kernel/bsort	main
cr.2	crc	main
du.2	test/duff	duff_init
du.5	test/duff	main
ex.2	expint	main
gd.4	sequential/gsm_dec	gsm_dec_Coefficients_0_12
gd.5	sequential/gsm_dec	gsm_dec_Coefficients_13_26
gd.6	sequential/gsm_dec	gsm_dec_Coefficients_27_39
gd.11	sequential/gsm_dec	gsm_dec_Decoding_of_the_coded_Log_Area_Ratios
gd.16	sequential/gsm_dec	gsm_dec_Postprocessing
ge.11	sequential/gsm_encode	Gsm_Preprocess
ge.13	sequential/gsm_encode	Gsm_Short_Term_Analysis_Filter
gs.2	sequential/g723_enc	g723_enc_fmult
gs.8	sequential/g723_enc	g723_enc_predictor_pole
gs.9	sequential/g723_enc	g723_enc_predictor_zero
gs.10	sequential/g723_enc	g723_enc_quan
gs.11	sequential/g723_enc	g723_enc_quantize
gs.16	sequential/g723_enc	g723_enc_update
hd.1	sequential/h264_dec	h264_dec_init
lc.0	lcdnum	main
li.3	app/lift	lift_controller
li.7	app/lift	lift_ctrl_set_vals
md.3	kernel/md5	md5_final
md.5	kernel/md5	md5_InitRandomStruct
md.13	kernel/md5	md5_R_RandomInit
md.14	kernel/md5	md5_R_RandomUpdate
md.15	kernel/md5	md5_transform
md.16	kernel/md5	md5_update
mp.9	sequential/mpeg2	mpeg2_frame_estimate
mp.11	sequential/mpeg2	mpeg2_fullsearch
sh.2	kernel/sha	sha_final
sm.0	sequential/statemate	main
sm.1	sequential/statemate	statemate_FH_DU
sm.2	sequential/statemate	statemate_generic_BLOCK_ERKENNUNG_CTRL
sm.3	sequential/statemate	statemate_generic_EINKLEMMSCHUTZ_CTRL
sm.4	sequential/statemate	statemate_generic_FH_TUERMODUL_CTRL
sm.8	sequential/statemate	statemate_main

Table 6 How LRA can improve the estimated WCET of TacleBench.

	Initial	Box			Octagons			Polyhedra		
Ref	WCET	Δ	Imp^t	Time	Δ	Imp^t	Time	Δ	Imp^t	Time
md.13	2648	0	0.0	<1s	1920	**72.5**	<1s	1920	**72.5**	<1s
an.15	173K	0	0.0	<1s	121K	**69.9**	1s	121K	**69.9**	<1s
gs.2	1105	0	0.0	<1s	738	**66.7**	9s	738	**66.7**	4s
hd.1	2092K	0	0.0	<1s	1371K	**65.5**	<1s	1371K	**65.5**	<1s
gs.8	2268	0	0.0	<1s	1476	**65.0**	1m	1476	**65.0**	1m
gs.9	6934	0	0.0	1s	738	10.6	4m	4428	**63.8**	29m
du.2	19K	0	0.0	<1s	12K	**60.1**	<1s	12K	**60.1**	<1s
du.5	22K	0	0.0	<1s	12K	**53.8**	<1s	12K	**53.8**	<1s
mp.9	52M	27M	**51.1**	56m	-	-	-	-	-	-
mp.11	10M	5M	**51.1**	2m	-	-	-	-	-	-
cr.2	227K	111K	**48.7**	<1s	111K	**48.7**	4s	111K	**48.7**	1s
lc.0	1540	0	0.0	<1s	595	**38.6**	<1s	595	**38.6**	<1s
md.15	8600	0	0.0	<1s	2304	**26.7**	<1s	2304	**26.7**	<1s
an.2	235K	0	0.0	<1s	58K	**24.8**	16s	58K	**24.8**	2s
an.0	466M	0	0.0	4s	4M	0.8	7m	115M	**24.6**	1m
md.5	13M	0	0.0	2m	-	-	-	3M	**21.0**	35m
ex.2	278K	1K	0.1	4s	28K	9.9	23s	53K	**19.2**	6s
sm.3	87	16	**18.3**	<1s	16	**18.3**	<1s	16	**18.3**	<1s
md.3	34K	0	0.0	18s	6K	17.2	6m	6K	**18.2**	31m
md.16	13K	0	0.0	7s	2K	**17.7**	19s	2K	**17.7**	10s
sm.0	268K	46K	**17.1**	16s	-	-	-	-	-	-
gs.10	942	0	0.0	<1s	126	**13.3**	1s	126	**13.3**	<1s
am.47	3649	0	0.0	13m	76	2.0	14m	485	**13.2**	14m
gs.11	2020	0	0.0	1s	252	**12.4**	17s	252	**12.4**	4s
md.14	51K	0	0.0	2m	5K	**10.4**	21m	-	-	-
sm.4	595	62	**10.4**	3s	62	**10.4**	2m	62	**10.4**	72m
li.7	1088	0	0.0	<1s	102	**9.3**	2s	102	**9.3**	<1s
gs.16	3760	0	0.0	7s	254	**6.7**	63m	-	-	-
ad.6	66	0	0.0	<1s	4	**6.0**	<1s	4	**6.0**	<1s
ae.7	66	0	0.0	<1s	4	**6.0**	<1s	4	**6.0**	<1s
ad.7	67	0	0.0	<1s	4	**5.9**	<1s	4	**5.9**	<1s
ae.8	67	0	0.0	<1s	4	**5.9**	<1s	4	**5.9**	<1s
sm.1	266K	14K	**5.1**	21s	-	-	-	-	-	-
sm.8	266K	14K	**5.1**	26s	-	-	-	-	-	-
an.16	530	0	0.0	<1s	25	**4.7**	<1s	25	**4.7**	<1s
am.50	2251	0	0.0	15m	76	3.3	15m	95	**4.2**	15m
am.49	2281	0	0.0	9m	76	3.3	14m	95	**4.1**	12m
sm.2	248	10	**4.0**	<1s	10	**4.0**	1s	10	**4.0**	1s
bs.4	5580	0	0.0	<1s	196	**3.5**	<1s	196	**3.5**	<1s
am.12	468	0	0.0	12m	16	**3.4**	12m	16	**3.4**	12m
ad.14	120	0	0.0	<1s	4	**3.3**	<1s	4	**3.3**	<1s
ae.16	120	0	0.0	<1s	4	**3.3**	<1s	4	**3.3**	<1s
li.3	3405	0	0.0	11s	102	**2.9**	47m	-	-	-
ae.10	1473	0	0.0	<1s	33	**2.2**	<1s	33	**2.2**	<1s
ge.11	47K	0.16K	0.3	1s	1K	**2.0**	1m	1K	**2.0**	16s
am.68	12K	0	0.0	10m	0.15K	**1.3**	73m	-	-	-
gd.16	23K	0.32K	**1.3**	<1s	0.32K	**1.3**	2s	0.32K	**1.3**	<1s
gd.4	1333	16	**1.2**	1s	16	**1.2**	6s	16	**1.2**	1s
gd.6	1333	16	**1.2**	<1s	16	**1.2**	3s	16	**1.2**	<1s
sh.2	25K	0.31K	**1.2**	6s	0.31K	**1.2**	3m	0.31K	**1.2**	1m
an.8	3079K	0	0.0	<1s	0	0.0	7s	34K	**1.1**	1s
an.14	3079K	0	0.0	<1s	0	0.0	5s	34K	**1.1**	1s
gd.11	1420	16	**1.1**	1s	16	**1.1**	35m	-	-	-
ge.13	727K	8K	**1.0**	1m	-	-	-	-	-	-
bs.0	1045K	0	0.0	<1s	0	0.0	2s	10K	**0.9**	<1s
bs.3	1045K	0	0.0	<1s	0	0.0	1s	10K	**0.9**	<1s
bs.5	1053K	0	0.0	<1s	0.20K	0.0	7s	10K	**0.9**	1s
gd.5	837	8	**0.9**	<1s	8	**0.9**	1s	8	**0.9**	<1s
am.17	8930	0	0.0	9m	76	**0.8**	20m	-	-	-
am.18	8901	0	0.0	11m	76	**0.8**	20m	-	-	-

Table 7 Loop bounds discovery, using PAGAI without the help of oRange nor the user-given bounds. This experiment is performed for the 54 cases from Table 6 where PAGAI terminates with either octagons or polyhedra ; the box domain is unable to find loop bounds and is not considered here. Within this test set, 10 programs contain no loop and are thus trivially bounded (adVI, adVII, adXIV, aeVII, aeVIII, aeXVI, gdXI, smII, smIII, smIV). For the remaining programs, first column gives the depth of nested loops and column two indicates if PAGAI gives a bounded (i.e., finite) WCET estimation. The WCET value is not given: it corresponds to the best PAGAI estimation in Table 6. The "paradoxal" result for ex.2 (loop depth 2 and bounded) is due to the fact that PAGAI "bounds" the inner-loop to 0 (i.e., the loop appears in a infeasible branch).

	loop depth	PAGAI
ae.10	1	**bounds**
bs.4	1	**bounds**
gd.4	1	**bounds**
gd.5	1	**bounds**
gd.6	1	**bounds**
gd.16	1	**bounds**
ge.11	1	**bounds**
gs.2	1	**bounds**
gs.8	1	**bounds**
gs.16	1	**bounds**
lc.0	1	**bounds**
li.7	1	**bounds**
md.15	1	**bounds**
du.2	1	**bounds**
du.5	1	**bounds**
hd.1	1	**bounds**
md.13	1	**bounds**
an.15	1	**bounds**
an.16	1	**bounds**
am.12	1	$\top$
an.2	1	$\top$
gs.10	1	$\top$
gs.11	1	$\top$
li.3	1	$\top$
sh.2	1	$\top$

	loop depth	PAGAI
ex.2	2	**bounds**
am.47	2	$\top$
am.49	2	$\top$
am.50	2	$\top$
an.8	2	$\top$
an.14	2	$\top$
bs.0	2	$\top$
bs.3	2	$\top$
bs.3	2	$\top$
cr.2	2	$\top$
gs.9	2	$\top$
md.3	2	$\top$
md.14	2	$\top$
md.16	2	$\top$
am.17	3	$\top$
am.18	3	$\top$
am.68	3	$\top$
an.0	3	$\top$
md.5	4	$\top$

Table 8 Observing the impact of compilation levels on LRA.

	O0			CO						
Ref	Initial WCET	Best WCET	Best Imp^t	Opt. speedup	Initial WCET	Best WCET	Best Imp^t	Traceability		
md.13	2648	728	72.5	3.3x	791	215	72.8	100%	of	2
an.15	173K	52K	69.9	3.0x	58K	17K	69.9	100%	of	6
gs.2	1105	367	66.7	2.3x	479	171	64.3	100%	of	14
hd.1	2092K	721K	65.5	2.1x	1019K	350K	65.6	100%	of	4
gs.8	2268	792	65.0	2.4x	963	347	63.9	100%	of	28
gs.9	6934	2506	63.8	2.4x	2910	1062	63.5	100%	of	28
du.2	19K	8K	60.1	2.3x	8K	3K	64.0	100%	of	2
du.5	22K	10K	53.8	2.4x	9K	4K	59.3	100%	of	3
mp.9	52M	25M	51.1	4.6x	11M	11M	0.0	79%	of	890
mp.11	10M	5M	51.1	4.6x	2M	2M	0.0	79%	of	178
cr.2	227K	116K	48.7	2.3x	97K	50K	48.7	41%	of	24
lc.0	1540	945	38.6	2.4x	641	421	34.3	100%	of	4
md.15	8600	6296	26.7	2.8x	3064	2200	28.1	100%	of	2
an.2	235K	176K	24.8	2.9x	80K	59K	25.9	100%	of	21
an.0	466M	351M	24.6	3.0x	157M	116M	25.9	100%	of	44
md.5	13M	10M	21.0	3.4x	4M	3M	15.0	80%	of	40
ex.2	278K	224K	19.2	1.3x	218K	218K	0.0	46%	of	13
sm.3	87	71	18.3	1.1x	78	59	24.3	100%	of	5
md.3	34K	28K	18.2	2.7x	13K	10K	17.1	100%	of	23
md.16	13K	11K	17.7	2.6x	5K	4K	17.2	100%	of	10
sm.0	268K	222K	17.1	1.1x	237K	193K	18.8	100%	of	108
gs.10	942	816	13.3	2.5x	381	325	14.6	100%	of	4
am.47	3649	3164	13.2	2.6x	1417	1282	9.5	100%	of	26
gs.11	2020	1768	12.4	2.4x	830	718	13.4	100%	of	11
md.14	51K	46K	10.4	2.7x	19K	17K	10.8	97%	of	37
sm.4	595	533	10.4	1.1x	547	493	9.8	100%	of	42
li.7	1088	986	9.3	2.1x	516	468	9.3	100%	of	9
gs.16	3760	3506	6.7	2.3x	1653	1539	6.8	100%	of	69
ad.6	66	62	6.0	3.0x	22	20	9.0	100%	of	3
ae.7	66	62	6.0	3.0x	22	20	9.0	100%	of	3
ad.7	67	63	5.9	2.9x	23	21	8.6	100%	of	3
ae.8	67	63	5.9	2.9x	23	21	8.6	100%	of	3
sm.1	266K	252K	5.1	1.1x	237K	224K	5.1	100%	of	97
sm.8	266K	252K	5.1	1.1x	237K	224K	5.1	100%	of	96
an.16	530	505	4.7	1.7x	316	299	5.3	100%	of	3
am.50	2251	2156	4.2	2.6x	860	823	4.3	100%	of	8
am.49	2281	2186	4.1	2.6x	865	839	3.0	100%	of	9
sm.2	248	238	4.0	1.1x	230	220	4.3	100%	of	11
bs.4	5580	5384	3.5	1.9x	2883	2687	6.7	100%	of	5
am.12	468	452	3.4	2.6x	181	177	2.2	100%	of	13
ad.14	120	116	3.3	2.6x	46	44	4.3	100%	of	8
ae.16	120	116	3.3	2.6x	46	44	4.3	100%	of	8
li.3	3405	3303	2.9	1.6x	2093	2045	2.2	100%	of	57
ae.10	1473	1440	2.2	2.1x	706	690	2.2	100%	of	6
ge.11	47K	46K	2.0	2.5x	19K	18K	5.0	100%	of	24
am.68	12K	12K	1.3	1.2x	9K	9K	0.5	100%	of	52
gd.16	23K	23K	1.3	2.3x	10K	10K	3.1	100%	of	12
gd.4	1333	1317	1.2	2.4x	553	537	2.8	100%	of	12
gd.6	1333	1317	1.2	2.4x	553	537	2.8	100%	of	12
sh.2	25K	24K	1.2	2.2x	11K	11K	0.9	76%	of	39
an.8	3079K	3045K	1.1	2.3x	1333K	1310K	1.7	100%	of	14
an.14	3079K	3045K	1.1	2.3x	1333K	1310K	1.7	100%	of	15
gd.11	1420	1404	1.1	2.4x	601	585	2.6	100%	of	121
ge.13	727K	719K	1.0	1.9x	377K	369K	2.0	100%	of	289
bs.0	1045K	1035K	0.9	2.7x	389K	385K	0.9	100%	of	6
bs.3	1045K	1035K	0.9	2.7x	389K	385K	0.9	100%	of	5
bs.5	1053K	1043K	0.9	2.7x	393K	389K	1.0	100%	of	10
gd.5	837	829	0.9	2.5x	337	329	2.3	100%	of	7
am.17	8930	8854	0.8	1.1x	8431	8405	0.3	100%	of	36
am.18	8901	8825	0.8	1.1x	8426	8400	0.3	100%	of	36

B Compiler optimization level

Options controlling optimizations are numerous and may vary a lot depending on the targeted processor and the compiler version. Options listed here are for the compiler used for our experiment (`arm-elf-gcc (GCC) 4.4.2`), with no guarantee that they apply directly to other compilers. Ensuring the coherence of a set of optimizations is technically hard, this is why we start with a predefined level of optimization (`-O1`) and remove optimizations that may modify the control structure (using the corresponding `-fno` flag). To select the options, we just rely on the user manual: we remove any optimization that mention a possible influence on the control structure or that may affect the precision of the debugging information (i.e. the association instruction/source line).

Loop transformations may drastically change the structure of the program, and are thus forbidden.

```
-fno-loop-block \
-fno-loop-interchange \
-fno-loop-strip-mine \
-fno-move-loop-invariants \
-fno-reschedule-modulo-scheduled-loops \
-fno-unroll-loops \
-fno-unroll-all-loops \
-fno-unsafe-loop-optimizations \
```

Miscelaneous CFG transformations concern dead code elimination, inlining, branch removal, block reordering etc.

```
-fno-dce \
-fno-dse \
-fno-guess-branch-probability \
-fno-inline-small-functions \
-fno-crossjumping \
-fno-if-conversion \
-fno-if-conversion2 \
-fno-jump-tables \
-fno-reorder-blocks \
-fno-reorder-blocks-and-partition \
-fno-unswitch-loops \
```

SSA tree optimizations and misc. global optimizations have an indirect influence in the control structure, by removing, regrouping or re-ordering instructions. They also affect the precision of the debugging information (dwarf) which is the only information we have to relate the binary and the source code.

```
-fno-tree-builtin-call-dce \
-fno-tree-ccp \
-fno-tree-ch \
-fno-tree-copyrename \
-fno-tree-dce \
-fno-tree-dominator-opts \
-fno-tree-dse \
-fno-tree-fre \
```

```
-fno-tree-loop-distribution \
-fno-tree-loop-im \
-fno-tree-loop-ivcanon \
-fno-tree-loop-linear \
-fno-tree-loop-optimize \
-fno-tree-sra \
-fno-tree-ter \
-fno-auto-inc-dec \
-fno-cprop-registers \
-fno-defer-pop \
-fno-ipa-pure-const \
-fno-ipa-reference \
-fno-merge-constants\
-fno-split-wide-types \
-fno-unit-at-a-time \
```