Working around Loops for Infeasible Path Detection in Binary Programs

Jordy Ruiz, Hugues Cassé, Marianne de Michiel
IRIT - Université de Toulouse, France
{jruiz,casse,michiel}@irit.fr

***Abstract*—The research of a safe Worst-Case Execution Time (WCET) estimation is necessary to build reliable hard, critical real-time systems. Infeasible paths are a major cause of overestimation of the Worst-Case Execution Time (WCET): without data flow constraints, static analysis by implicit path enumeration will take into account semantically impossible, potentially expensive execution paths, making the Worst-Case Execution Path unreachable in practice. We present in this paper an approach that allows to significantly tighten the WCET by identifying infeasible paths, namely in loops, and injecting them as additional Integer Linear Programming (ILP) constraints during the WCET computation. Our entire analysis, albeit platform independent, works directly on binary programs in order to get the tightest, most reliable WCET. Impactful infeasible paths are largely found within (often nested) loops; therefore having an efficient, exploitable and reasonably scalable representation of the state of a program within loops is a key challenge of infeasible path analysis. We show ours to yield decidedly significant results on a selection of benchmarks from actual hard real-time applications as well as the classic Mälardalen suite.**

I. Introduction

Worst-Case Execution Time (WCET) estimation is indispensable in order to ensure the safety of critical embedded systems: tasks in such systems must always be executed in a reasonable time. Because these tasks may be repeatedly executed at a high frequency or on a large amount of devices, extensive tests are not enough: exceptional scenarios in which the program performs unordinarily poorly are a threat to the system safety that cannot reliably be detected by anything but static analysis.

On top of the difficulty of accurately modelling the hardware of these systems, the search of the longest path through the program is very computationally complex. The Implicit Path Enumeration Technique (IPET) is a realistic approach to find a reasonable upper bound to such programs. Because finding the real WCET is practically impossible, we are always looking for an estimation proven to be higher than the real WCET.

Hence, the WCET may be estimated above its real value, leading to additional costs due to consequently heavier hardware requirements. We aim to reduce this unnecessary overhead by *tightening* (i.e., lowering the value of) the WCET estimation while keeping it *safe*, that is, above the real WCET.

We observe that, often, one important overestimation factor is that the program can never actually run through the Worst-Case Execution Path (WCEP) given by the Integer Linear Programming (ILP) system representing the program: the output WCET is actually an estimation of the execution time of an *infeasible*, semantically impossible path (for example, because of conflicting conditions).

In order to get the most reliable estimation, analysis is best performed directly on the binary code: it is usually hard to link flow information between the source code and the binaries due to compiler tricks, including some average-case optimisations that worsen the WCET. Although it is possible to work on the source code [16] or on an intermediary representation of a compiler [8] (thus making the WCET analysis compiler specific) and even propagate some optimisations to the WCET [11], propagating facts from the source code can be difficult; working on the binary code is the safest and most direct approach to deal soundly with data flow information.

Contribution: This paper proposes a reasonably scalable approach to infeasible path detection on binary programs that deals with loops and function calls, in a way that preserves enough information on the program states to detect impactful infeasible paths within loops. Our analysis uses a composable representation of the state of the program, and can be performed on local parts of code such as the body of subroutines and loops. Not only does this improve its efficiency, it also helps to output infeasible paths with the most possibly general scope, thus maximising their impact on the estimated WCET.

Outline: After a quick overview of published works around the issue in Section II, we present our view on binary programs in Section III. Section IV presents our choice of abstraction of program states illustrated with a running example, and Section V details our infeasible path detection algorithm. Section VI showcases it on real-time benchmarks.

II. Related works

Several works have previously been published on the topic of infeasible paths detection.

In [7], [8], Gustafsson et al. present the abstract execution of a program, that is, the execution of the program considering ranges of values for input data and program variables. Observers are used along the execution to detect interesting flow facts: loop bounds, mutually exclusive blocks or infeasible paths. The approach is powerful and exhaustive but, in our opinion, does not scale well with sizeable applications such as those found in the industry: although the collected data is abstracted to ranges of values, a lot of mostly concrete paths still need to be executed.

Chen et al. [4] propose a combined approach to compute the WCET and the feasibility of the corresponding path. Their

2470-6892/17 $31.00 © 2017 IEEE
DOI 10.1109/SCAM.2017.13

analysis performs a backward traversal of the Control Flow Graph (CFG) paths of a binary program and checks for the satisfiability of pairs of condition-condition and assignment-condition. During the computation, if a contradiction is exposed, the corresponding path is rejected and ignored for the remainder of the analysis. This approach has two main issues: (a) it does not address programs with loops and (b) it only supports very simple forms of conditions and assignments made of a variable and a constant.

In [9], Holsti also presents a combined approach where the time, loop bounds and program states are abstracted using Presburger algebra extended to support undefined values. A disjunction of possible states is maintained and, depending on the condition, some states may be removed, along with their corresponding execution path. The loop state blow-up is reduced by expressing induction variables as a function of the iteration number (as in our approach). Yet, the state explosion caused by alternatives in the execution paths leaves an important scalability issue which we will address.

Andalam et al. in [1] propose an integrated approach computing Worst-Case Responsive Time (WCRT) and infeasible paths together for synchronous languages (namely Esterel). The application semantics, leading to infeasible paths, and the WCRT calculation are supported by a model checker. The approach does not seem scalable due to the state explosion in the model checker, and the way execution times are accounted for is not realistic except for some very simple microprocessor architectures.

In previous work [15], we have proposed a first approach based on SMT solving to detect infeasible paths. It involves building abstract states of the program as sets of predicates on the processor registers and on the content of the memory. At any point of the program, getting a non-satisfiable set of predicates signals that the corresponding path is infeasible and can be removed from the WCET computation. This paper focused on the generation of minimal infeasible paths using SMT unsat cores, which will not be detailed here. Although the experimental results were encouraging, our representation of the program states was difficult to maintain and the limited support for loops was a major issue: abstract states at loop entries were merged into one using a destructive intersection of predicate sets, causing significant loss of information. In the following sections, we propose a different approach to detect infeasible paths that partially resolves the issues encountered.

III. Program representation

This section presents the concrete semantics of the processed program, using a CFG to represent the control flow and semantics instructions to represent the data flow. It is applied to an example which is addressed throughout the paper.

A. Motivating example

Fig. 1 showcases an example of an impactful infeasible path: the function `compute` can be called at most once per call to `g` because the path that enters both `if` conditions is infeasible.

```
void f(int a) {
        int x = 0, y = 0, i;
        for(i = 0; i < N; i++) {
                x = x + 1;
                y = y + 2;
        }
        if(2 * x == y + a)
                compute();
}
void g(int a) {
        f(a);
        if(a % 4 != 0)
                compute();
}
```

Fig. 1. Example 1

If `compute` is an expensive function with a high WCET compared to the operations in `f`, detecting this infeasible path would almost halve the WCET estimation, because it would otherwise account for two calls to `compute`.

The analysis of the function `f` would result in two paths (due to the `if` statement): one that asserts that `f` was called with $a \neq 0$ and one that asserts $a = 0$ (and goes through the call to `compute`), because the loop analysis would deduce that $y = 2x$ and thus $(2x = y + a) \Leftrightarrow (a = 0)$. Upon reaching the condition in the function `g`, we observe that $(a \mod 4 \neq 0)$ conflicts with $(y = 2x) \wedge (2x = y + a)$, and therefore that the execution path going through both calls to `compute` is semantically impossible.

In order to achieve this, we have to analyse the control flow of the program and the semantics of the program execution.

B. Control Flow Graph

Parsing binary programs requires an adequate representation of the control flow, which is provided by CFGs. A CFG $G := (V, E, \epsilon, \omega)$ is a directed graph represented by a set of nodes (the basic blocks V), a set of edges ($E \subseteq V \times V$), an entry node ($\epsilon \in V$) and an exit node ($\omega \in V$). Each node represents a basic block, that is, a block of instructions that are always read sequentially: an instruction cannot branch to the middle of a block, and similarly, a branching instruction must be at the end of a block.

A CFG represents the execution flow of a subroutine: calls to a function are represented by a virtual block that substitutes the execution of the callee function. A semantically equivalent result is obtained by replacing that virtual block by the CFG of the called subroutine. The CFG of the example in Fig. 1 is displayed in Fig. 2.

C. Concrete program states

The state of the machine at any point of the program is represented by a map from the registers $Reg := \{r_0, r_1, ..., r_{n-1}\}$ and the memory $Mem = \mathbb{N}/2^{32}$ (the set of addressable memory cells[1]) to 32-bit integers: $S := \mathbb{V} \to \mathbb{Z}/2^{32}$, where

[1] $\mathbb{N}/2^{32}$ and $\mathbb{Z}/2^{32}$ are respectively the sets of naturals and integers that can be represented on 32-bits, i.e. the size of machine words.

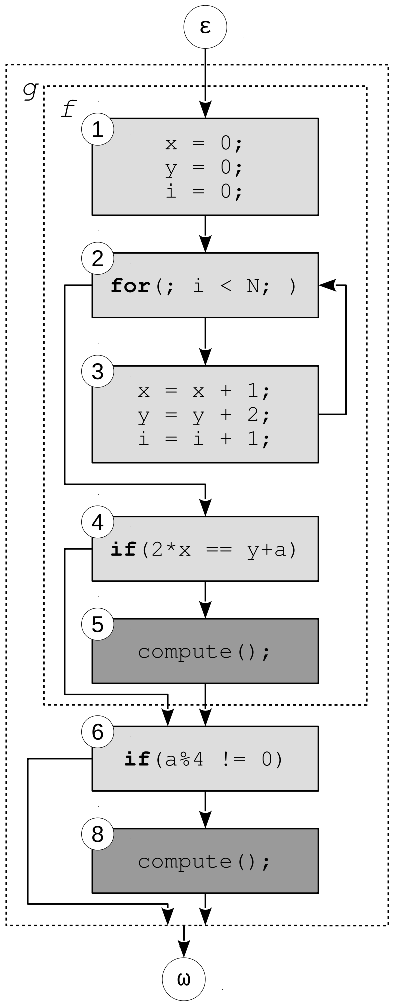


Fig. 2. CFG of Example 1

$\mathbb{V} := (\mathit{Reg} \cup \mathit{Mem})$ is the set of program variables. S is called the set of *concrete* states.

Concrete states evolve throughout the program, and are updated by each instruction execution. For this purpose, we define the concrete interpretation function $\mathbb{I}$ that works on a set of semantic instructions [2] noted I_{sem}, that is, an architecture-independent abstraction of machine instructions[2]. Each machine instruction is translated by our framework into one or several semantic instructions. This semantic instruction set is built in a similar fashion to a RISC instruction set. $\mathbb{I}$ maps each semantic instruction to a function on S that updates the concrete state. For instance, the `ADD`, `SUB`, `MUL`, `DIV`, `MOD` semantic instructions write in the first register the result of the corresponding arithmetic operation on the two other registers. Hence, for example, for any $s \in S$:

$$\mathbb{I}[\texttt{SUB}\ r_0, r_0, r_1] := s \mapsto s\,[r_0 \mapsto s(r_0) - s(r_1)]$$

[2] Semantic instructions enable the easy extension of our analysis to any instruction set.

where $s\,[x \mapsto k]$ denotes s where x now maps to k.

In order to detect conflicting conditions, the next section proposes an abstract interpretation function $\hat{\mathbb{I}}$ and a corresponding abstract state $\hat{S}$ expressing the states as predicates. For example, any state of the block ⑧ from Fig. 2 includes the predicate $(y = 2x) \wedge (a \mod 4 \neq 0)$, which interestingly contradicts $(2x = y + a)$. Section IV formalizes this abstraction.

IV. Abstract interpretation

We propose an abstract domain that expresses the machine states as predicates on registers and memory cells. This abstraction is then extended to efficiently support the analysis of loop iterations and of functions, independently of their call context.

A. Abstraction by predicates

1) Definitions: An abstract program state is defined as a disjunction of abstract states, in $\mathcal{P}(\hat{S})$. It represents the set of states from all paths leading to this program point.

An abstract state in $\hat{S}$ is a triplet made of:

- $L := \mathit{Reg} \to \mathit{Expr}$, an abstraction of the local variables;
- $M := \mathit{Mem} \to \mathit{Expr}$, an abstraction of the memory (heap and stack);
- $P^* := (\mathit{Expr} \times \Phi \times \mathit{Expr})^*$, a set of predicates in conjunction, each made of an operator and two operands, where the set of operators Φ is $\{=, \neq, <, \leq\}$.

L and M are merely a particular form of equality predicates. *Expr* is inductively defined as either:

- a constant $c \in C$;
- a register $r_k \in \mathit{Reg}$, identified by a number (between 0 and 16 for ARM processors, for example);
- a memory cell $m_c \in \mathit{Mem}$, identified by its address (either absolute or relative to the initial stack pointer SP);
- an arithmetic expression $(\mathit{Expr} \times \Psi \times \mathit{Expr})$ where Ψ is the set of arithmetic operators $\{+, -, \times, \div, \mathrm{mod}\}$;
- $\top$, which denotes an unknown value.

Constants in $C := \mathbb{Z}/2^{32} \times \{\emptyset, +\mathit{SP}\}$ are 32-bit signed or unsigned integers that may or may not be relative to the initial state of the stack pointer ($+\mathit{SP}$ marker). Considering the stack pointer SP as a symbolic value enables the analysis to be independent of its actual value; this assumes that subroutines cleanly manage the call stack.

2) State concretization: The concretization function $\gamma : \hat{S} \to \mathcal{P}(S)$ is a common tool used to check the soundness of an abstraction. For any state $\hat{s} := (l, m, p) \in \hat{S} = L \times M \times P^*$, we define the concretization function as

$$\gamma(\hat{s}) := \gamma_L(l, p) \cap \gamma_M(m, p)$$

$$\gamma_L(l, p) := \{s \in S \mid s(r_k) \in \gamma_e(l(r_k), p)\ \forall r_k \in \mathit{Reg}\}$$
$$\gamma_M(m, p) := \{s \in S \mid s(m_c) \in \gamma_e(m(m_c), p)\ \forall m_c \in \mathit{Mem}\}$$

where $\gamma_e : \mathit{Expr} \to \mathcal{P}(\mathbb{Z}/2^{32})$ is the concretization of an expression over S, inductively defined in what follows.

Constants ($c \in C$) are evaluated to a singleton as such, where sp_0 designates the initial value of the stack pointer:

$$\gamma_e(c,p) := \begin{cases} \{k\} & \textit{if } \exists k \in \mathbb{Z}/2^{32},\ c = k \\ \{k + sp_0\} & \textit{if } \exists k \in \mathbb{Z}/2^{32},\ c = k + SP \end{cases}$$

$\gamma_e(\top, p) := \mathbb{Z}/2^{32}$, because $\top$ represents any value.

In the case of an arithmetic expression $e := (e_1, \psi, e_2)$, we define γ_e using f_ψ, the function that applies the binary operation (e.g. addition) corresponding to the arithmetic operator ψ.

$$\gamma_e(e,p) := \{f_\psi(k_1,k_2) \mid k_1 \in \gamma_e(e_1,p) \wedge k_2 \in \gamma_e(e_2,p)\}$$

Finally, we define the concretization of any variable $v \in \mathbb{V}$ as the set of values with which the system of predicates remains satisfiable:

$$\gamma_e(v,p) := \{x \in \mathbb{Z}/2^{32} \mid ((v = x) \wedge \bigwedge_{p_i \in p} p_i) \neq \bot\}$$

Using this, states $x \in \mathcal{P}(\hat{S})$ at a program point can be concretized as $\gamma(x) := \bigcup_{\hat{s} \in x} \gamma(\hat{s})$.

3) Joining: While the analysis runs through the CFG, the number of states at program points quickly increases as they must encompass all the execution paths to that point. Since each condition doubles the path count, the growth is exponential even in loopless programs.

We therefore introduce the operator $\sqcup : \mathcal{P}(\hat{S}) \times \mathcal{P}(\hat{S}) \to \mathcal{P}(\hat{S})$, which *joins* states using the set union operator, and may *shrink* them using the $\sqcap : \mathcal{P}(\hat{S}) \to \hat{S}$ operator. This happens when the states count reaches a tunable thresold $\eta \in [1, +\infty]$, which offers a trade-off between performance (speed and memory usage) and efficiency[3].

For any program states $x, x' \in \mathcal{P}(\hat{S})$, we define:

$$\begin{aligned} x \sqcup x' &:= \left\{\bigsqcap(x \cup x')\right\} && \text{if } |x| + |x'| > \eta \\ x \sqcup x' &:= x \cup x' && \text{otherwise} \end{aligned}$$

based on the shrinking operator $\bigsqcap$, defined on $x = \{\hat{s}_1, \ldots, \hat{s}_n\}$ as the *intersection* of all its abstract states $\bigsqcap x := \bigsqcap_{i \in [1,n]} \hat{s}_i \in \hat{S}$. The concretization of the state resulting from this intersection will include the concretization of each of the original states: $\gamma(\bigsqcap x) \subseteq \bigcup_{i \in [1,n]} \gamma(\hat{s}_i)$.

The intersection is defined on two states $\hat{s} := (l, m, p) \in \hat{S}$ and $\hat{s}' := (l', m', p') \in \hat{S}$ by:

$$\hat{s} \sqcap \hat{s}' := \left(\begin{array}{c} r_k \mapsto \begin{cases} l(r_k) & \text{if } l(r_k) = l'(r_k) \\ \top & \text{else} \end{cases} \\ m_c \mapsto \begin{cases} m(m_c) & \text{if } m(m_c) = m'(m_c) \\ \top & \text{else} \end{cases} \\ p \cap p' \end{array} \right)$$

Predicates are expressed in a canonical form, such that $p \cap p'$ preserves most semantically equivalent predicates. For example, there cannot be $a + 1 > 0$ and $0 < 1 + a$ in different states: we make sure there are no syntactically different predicates that are equivalent by commutativity.

[3] We empirically chose $\eta = 250$ for the most sizeable benchmarks, yet $\eta = +\infty$ (never shrink) suffices for small benchmarks.

B. Interpreting instructions

We define the interpretation function on the abstract domain $\hat{\mathbb{I}} : I_{sem} \to \hat{S} \to \hat{S}$, using the notation $l[r_k \mapsto e]$ to denote l where r_k has been changed to map to e (and similarly on m).

For example, the addition instruction is interpreted as such:

$$\hat{\mathbb{I}}[\texttt{ADD } r_0, r_1, r_2] := (l, m, p) \mapsto (l\left[r_0 \mapsto l(r_1) + l(r_2)\right], m, p)$$

Sometimes, assembly instructions are too complex to be translated to semantic instructions, or to be completely exploited. For this purpose, we introduce the `SCRATCH` r_i semantic instruction which sets an undefined value to r_i.

Although the register r_i could be set to $\top$, it is more beneficial to introduce a new variable $v_k \in \mathit{Var}$ and to enrich the set of expressions *Expr* with this new *Var* set. The $\top$ value is destructive for the subsequent computations and conditions, while the introduced variables record the identity and source of an unknown value. This allows us to maintain existing links between different variables that use an unknown result, and these links may suffice to detect conflicting conditions. We will thus introduce a new, arbitrarily named $v_k \in \mathit{Var}$ variable to interpret, for example, the `SCRATCH` r_4 semantic instruction:

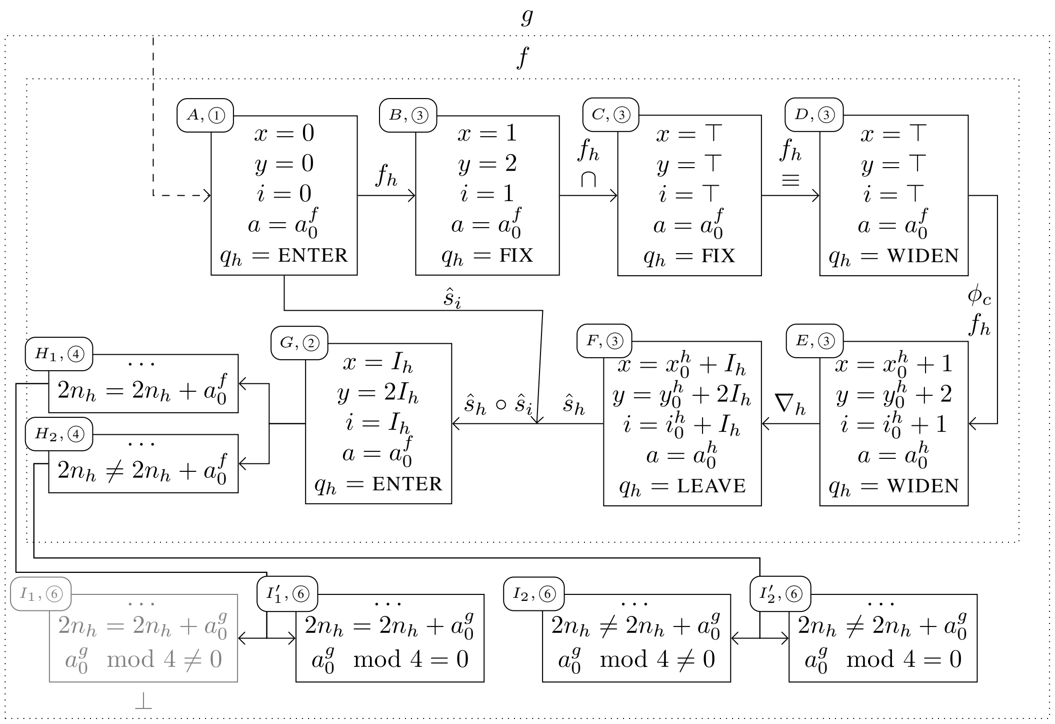
$$\hat{\mathbb{I}}[\texttt{SCRATCH } r_4] := (l, m, p) \mapsto (l[r_4 \mapsto v_k], m, p)$$

Example: to highlight the benefits of this practice, assume a program stores the result of a complex computation in a register r_i. Then, r_j receives $r_i + 1$. Later, an `if` block is entered if and only if $r_i = r_j$. This is a trivial infeasible path (dead code), yet if we had represented the result of the complex computation by a $\top$, we would be unable to detect it. This is because r_i and r_j would have been set to an anonymous $\top$, making their comparison impossible ($\top + 1 = \top$). Instead, we (automatically) represent the result of the complex computation by a v_k variable, and yield the $v_k = v_k + 1$ predicate, exhibiting an obvious contradiction.

This variable introduction technique can also be used to deal with memory reads to unknown addresses by `LOAD`. The `LOAD` r_i, r_j semantic instruction reads an integer from the memory at the address contained in the register r_j and stores it in the register r_i. If $l(r_j)$ is a constant a, then we update l to map r_i to $m(a)$. If $l(r_j)$ is not a constant, the unknown result of the memory access is represented by a newly generated variable $v_k \in \mathit{Var}$. Thus, for any $\hat{s} := (l, m, p) \in \hat{S}$,

$$\hat{\mathbb{I}}[\texttt{LOAD } r_i, r_j](\hat{s}) := \begin{cases} (l\left[r_i \mapsto m(l(r_j))\right], m, p) \text{ if } l(r_j) \in C \\ (l\left[r_i \mapsto v_k\right], m, p) \text{ if } l(r_j) \notin C \end{cases}$$

The `STORE` instruction, which puts a value in a memory cell, is more damaging when the address is not a known constant: we have to "scratch" the whole memory and lose any information previously collected. For this reason, it is crucial to handle all instructions that modify the stack pointer well. Assuming that the heap and stack memories are semantically separated (stack pointer relative addresses never hit the heap and vice versa), memory accesses may sometimes be categorised into accesses to the heap or stack memory depending on the way the address was computed, resulting in a lesser loss of information.



g
f
A, ①
$x = 0$, $y = 0$, $i = 0$, $a = a_0^f$, $q_h =$ ENTER
B, ③
$x = 1$, $y = 2$, $i = 1$, $a = a_0^f$, $q_h =$ FIX
C, ③
$x = \top$, $y = \top$, $i = \top$, $a = a_0^f$, $q_h =$ FIX
D, ③
$x = \top$, $y = \top$, $i = \top$, $a = a_0^f$, $q_h =$ WIDEN
f_h
f_h $\cap$
f_h $\equiv$
ϕ_c f_h
$\hat{s}_i$
E, ③
$x = x_0^h + 1$, $y = y_0^h + 2$, $i = i_0^h + 1$, $a = a_0^h$, $q_h =$ WIDEN
∇_h
F, ③
$x = x_0^h + I_h$, $y = y_0^h + 2I_h$, $i = i_0^h + I_h$, $a = a_0^h$, $q_h =$ LEAVE
$\hat{s}_h$
$\hat{s}_h \circ \hat{s}_i$
G, ②
$x = I_h$, $y = 2I_h$, $i = I_h$, $a = a_0^f$, $q_h =$ ENTER
H_1, ④
$2n_h = 2n_h + a_0^f$
H_2, ④
$2n_h \neq 2n_h + a_0^f$
I_1, ⑥
$2n_h = 2n_h + a_0^g$, $a_0^g \mod 4 \neq 0$
$\perp$
I_1', ⑥
$2n_h = 2n_h + a_0^g$, $a_0^g \mod 4 = 0$
I_2, ⑥
$2n_h \neq 2n_h + a_0^g$, $a_0^g \mod 4 \neq 0$
I_2', ⑥
$2n_h \neq 2n_h + a_0^g$, $a_0^g \mod 4 = 0$

Fig. 3. Abstract interpretation of Example 1

C. Modular analysis

Abstracting away subroutine calls in a program by virtually inlining the called code may simplify binary code analysis, but it comes with several drawbacks. Firstly, it duplicates the abstract interpretation process as many times as there are calls to the same subroutine. Secondly, it struggles to find infeasible paths that are independent of the call point of a subroutine: it will duplicate them and output one infeasible path per call point. Finally, designing a composable analysis that can be limited to the scope of any single-entry region, similarly to [10], also benefits loop analysis, what will be demonstrated in the next section.

Because of this, we want to be able to individually process any Single Entry Single Exit (SESE) region as a program state before composing it with the rest of the analysis. Interesting SESE regions may be functions (with an entry and a virtual exit node) or loop bodies (the loop header acts as both the entry and the exit).

In order to make abstract states easily composable, we express them in function of the initial state of the program at the entry of the SESE region. This region will be entered with the *identity* state, defined as $(Id_L, Id_M, \emptyset)$ where Id_L (resp. Id_M) maps each register (resp. each memory cell) to itself, in fact to its initial value. With this convention, the variables of *Expr* represent their (unknown) value at the entry of the region, and the abstract states represent the operations that are performed on the machine since the beginning of the local analysis.

We denote the map of a register or memory address to an expression as a pseudo-predicate. For example, if x maps to $x+1$ in l, we will express it as $x = x_0+1$, where x_0 represents the initial value of x at the beginning of the region. When it is unclear what the scope of x_0 is, we note it x_0^k, where k may either be a function or a loop identifier.

This enables us to define a *state composition* operation where $\forall \hat{s}, \hat{s}' \in \hat{S}, \hat{s} \circ \hat{s}'$ is $\hat{s}$ applied to $\hat{s}'$. If $x, x' \in \mathcal{P}(\hat{S})$, we define $x \circ x'$ as the cartesian product of its states: $x \circ x' := \{\hat{s} \circ \hat{s}' \mid \hat{s} \in x, \hat{s}' \in x'\}$.

For $\hat{s} := (l, m, p)$, $\hat{s}' := (l', m', p')$, we define:

$$\hat{s} \circ \hat{s}' := \big(r_k \mapsto \sigma_{\hat{s}'}(l(r_k)), m_c \mapsto \sigma_{\hat{s}'}(m(m_c)), \{(\sigma_{\hat{s}'}(e_1), \phi, \sigma_{\hat{s}'}(e_2)) \mid (e_1, \phi, e_2) \in p\} \cup p'\big)$$

where $\sigma_{\hat{s}'}$ is the function that transfers an expression to the scope of $\hat{s}'$, defined as:

$$\sigma_{\hat{s}'}(c) := \begin{cases} k & \text{if } \exists k \in \mathbb{Z}/2^{32},\ c = k \\ k + l(sp) & \text{if } \exists k \in \mathbb{Z}/2^{32},\ c = k + SP \end{cases} \quad \forall c \in C$$

$$\sigma_{\hat{s}'}(r_k) := l'(r_k) \quad \forall r_k \in Reg$$

$$\sigma_{\hat{s}'}(m_c) := m'(m_c) \quad \forall m_c \in Mem$$

$$\sigma_{\hat{s}'}(e_1, \psi, e_2) := (\sigma_{\hat{s}'}(e_1), \psi, \sigma_{\hat{s}'}(e_2)) \ \forall e_1, e_2 \in Expr, \psi \in \Psi$$

and the identity Id_{Expr} for any other expression.

Using this definition, the application of $\hat{s}$ to $\hat{s}'$ is a simple operation that, on $\hat{s}$, (a) replaces all registers and memory cells in the expressions of l, m and p by their current value in $\hat{s}'$, and (b) adds the unchanged predicates of $\hat{s}'$. Intuitively, we update the initial values of $\hat{s}$ by the results of $\hat{s}'$. The resulting state will be expressed in function of the initial values from the scope of $\hat{s}'$.

For instance, applying a state mapping $\{x \mapsto y_0{+}1, y \mapsto 2\}$ (for example, from a called function) to another state mapping $\{x \mapsto \top, y \mapsto 0\}$ (for example, from a call point) gives a state with $\{x \mapsto 0+1, y \mapsto 2\}$.

D. *Working on loops*

As we want to precisely represent the state of the program at any iteration of a loop, we will augment *Expr* with induction variables of the form I_h, which represent the amount of past iterations of the loop identified by the loop header $h \in V$. This enables us to accurately represent linear progressions inside loops, such as simple integer iterators (in a typical C `for` loop, for example).

We will later use this information to find conflicts valid for any iteration of a loop, which are usually the most interesting ones because of their significant potential impact on the WCET. This information can also be used to better control write accesses to arrays being parsed in loops (because we can identify the range of accessed addresses in memory).

In order to get abstract states expressed in function of those induction variables, we need to define a *widening*. The widening $\nabla_h : \hat{S} \to \hat{S}$ of a state on a loop identified by its header $h \in V$ is a key function that is obtained, for any $\hat{s}_h \in \hat{S}$, by computing a fixpoint on $f : \hat{s} \mapsto \hat{s}_h \nabla_h \hat{s}$, starting with the state $\top$[4]. For any $\hat{s}_h = (l_h, m_h, p_h)$,

$$\hat{s}_h \nabla_h \hat{s} := (r_k \mapsto \nabla_h^{r_k}(\hat{s}, l_h(r_k)), m_c \mapsto \nabla_h^{m_c}(\hat{s}, m_h(m_c)), \emptyset)$$

∇_h transforms a state representing the effect of *one* iteration of the loop (any path going from h to h without exiting the loop) to a state representing the effect of I_h iterations.

The widening of an expression e is defined, for any $x \in \mathbb{V}$ and $\hat{s} = (l, m, p) \in \hat{S}$ as $\nabla_h^x(\hat{s}, e) :=$

$$\begin{cases} e & \text{if } e = x \vee e \in C \\ l(e)[I_h \to I_h - 1] & \text{if } e \in \mathit{Reg} \wedge e \neq x \\ m(e)[I_h \to I_h - 1] & \text{if } e \in \mathit{Mem} \wedge e \neq x \\ x + I_h \times k & \text{if } \exists k \in C, e = x + k \\ x - I_h \times k & \text{if } \exists k \in C, e = x - k \\ \phi(\nabla_h^x(\hat{s}, e_1), \nabla_h^x(\hat{s}, e_2)) & \text{if } \exists e_1, e_2 \in \mathit{Expr}_{\bar{x}}, e = \phi(e_1, e_2) \\ \top & \text{else} \end{cases}$$

where the notation $e[I_h \to I_h - 1]$ denotes the expression e where all occurences of I_h have been replaced by $I_h - 1$ and where $\mathit{Expr}_{\bar{x}}$ is a subset of *Expr* without the variable x.

In the Example 1 illustrated in Fig. 2, the basic blocks ② and ③ make up a loop. If we considered the C variables x, y, and i to be machine registers or memory cells, the state $\hat{s}_0$ at the entry of ② would be initialised to $\{x \mapsto x_0^{\textcircled{2}}, y \mapsto y_0^{\textcircled{2}}, i \mapsto i_0^{\textcircled{2}}\}$. After one iteration, we get $\hat{s}_1 : \{x \mapsto x_0^{\textcircled{2}} + 1, y \mapsto y_0^{\textcircled{2}} + 2, i \mapsto i_0^{\textcircled{2}} + 1\}$, that is, the state (E) on the Fig. 3. The widening produces $\hat{s}_h := \hat{s}_0 \nabla_{\textcircled{2}} \hat{s}_1 : \{x \mapsto x_0^{\textcircled{2}} + I_{\textcircled{2}}, y \mapsto y_0^{\textcircled{2}} + 2I_{\textcircled{2}}, i \mapsto i_0^{\textcircled{2}} + I_{\textcircled{2}}\}$ (F).

[4]The abstract state $\top \in \hat{S}$ maps any variable to $\top$.

Then, using information from $\hat{s}_i$, the state on the entry edge ① $\to$ ② of the loop, which is $\{x \mapsto 0, y \mapsto 0, i \mapsto 0\}$ (A), we deduce that the general state valid at the block ② for any iteration of the loop is $\hat{s} := \hat{s}_h \circ \hat{s}_i = (\hat{s}_0 \nabla_{\textcircled{2}} \hat{s}_1) \circ \hat{s}_i$, which gives $\{x \mapsto 0 + I_{\textcircled{2}}, y \mapsto 0 + 2I_{\textcircled{2}}, i \mapsto 0 + I_{\textcircled{2}}\}$ (G).

The resulting state at the loop exit is $\{x \mapsto n_{\textcircled{2}}, y \mapsto 2n_{\textcircled{2}}, i \mapsto n_{\textcircled{2}}\}$ (H) where $n_{\textcircled{2}}$ is the final iteration count. If knowledge of $n_{\textcircled{2}}$ has been fed from an external loop analysis or from given flow facts of the benchmark, it is replaced by its constant value. Otherwise, it is kept as a variable throughout the rest of the program analysis, which could still be useful because taking the conditional edge ② $\to$ ④ which exits the loop will generate an $i \geq N$ predicate, indirectly bounding $n_{\textcircled{2}}$. Interestingly, the analysis finds some loop bounds on-the-fly as a side effect of this abstract interpretation.

Widening uses simple algebraic properties to generalise linear sequences in loops. For example, the predicate $x = x_0^h - 3$ obtained after an iteration over a loop h would be widened to $x = x_0^h - 3I_h$. Although more general arithmetico-geometric sequences (of the type $x_{n+1} = ax_n + b$, a and b constants) could easily be generalised as well, they contain a power term (a^n). This would make the predicate too complex to be used in an SMT solver, such non-linear predicates are thus safely removed from the conjunction fed to the solver.

V. Program analysis

We will now use the previously defined abstraction and operations defined on abstract states to perform the analysis of the whole program and detect infeasible paths.

A. *Identifying an infeasible path*

The program is parsed in order to acquire knowledge of which program states are possible for each path and each point of the program. Our analysis aims to find infeasible paths, that is, paths that lead to an empty set of possible concrete states. Because obtaining such precise information is realistically unachievable, the abstract states described in the previous section provide an overestimation of the set of concrete states: an empty set of possible program states still safely denote an infeasible path. While the overestimation may cause the analysis to miss some infeasible paths, the soundness of the estimated WCET is not threatened: a missing infeasible path always leads to safe overestimation.

Notation: we use the abstract state $\bot \in \hat{S}$ to represent an impossible state: $\gamma(\bot) = \emptyset$. At the opposite, $\top \in \hat{S}$ represents any state: $\gamma(\top) = S$.

The $\top$ state may be used by the analysis to enter the program with an initial state devoid of any information, making it valid for any execution of it.

The aim of our analysis is to reduce states to $\bot$ whenever possible, which indicates that the path represented by the state is infeasible. In order to exploit this property, we will remember the path any abstract state represents by annotating it with any edge it goes through.

Because our abstract state representation is isomorphic to a conjunction of predicates over integer variables, linear integer SMT solving arises as an appropriate solution to the satisfiability problem, that is, asserting the existence or lack thereof of a solution to the system.

Although we have chosen to use CVC4 [3] as the SMT solver for our experiments, our analysis is largely solver-independent and has been made to work with z3 as well with almost identical results, overall slightly faster execution times, but a slightly heavier memory load.

Each abstract state is annotated with the path it represents, and each predicate is associated with the CFG edges responsible for it. Upon finding an unsatisfiable ($\perp$) state, instead of using the entire path it represents, a family of infeasible paths is produced when possible with a minimal amount of edges, using only edges that have affected the predicates responsible for the conflict. This is made possible by the exploitation of unsat cores, a minimal subset of constraints responsible for the unsatisfiability of a problem, which some SMT solvers are able to output.

In order to further minimise infeasible paths, we use dominance and post-dominance information from the CFG to detect unnecessary edges. For example, if an infeasible path contains two edges e_1, e_2, then e_1 is superfluous if e_1 dominates e_2; inversely, e_1 is superfluous if e_1 post-dominates e_2. The historical definition of dominance according to [5], which can easily be extended to edges in a CFG, is that "box i dominates box j if every path which passes through box j must also pass through box i". Reciprocally, an edge e_2 post-dominates e_1 if every path from e_1 to an exit node contains e_2.

Because our analysis on a subroutine produces predicates that are function of the initial state of the registers and the memory, we are also able to find infeasible paths *a posteriori* in subroutines, when we learn the context of the various call points of that subroutine and the parameters it was called with.

B. Parsing the CFG

The functions of the program are parsed separately, in an order that depends on the Program Call Graph (PCG): a function is always parsed after the functions it calls (recursive functions are not supported). Upon reaching a function call, we compose the states of the caller with the resulting states of the callee function that would have been analysed beforehand.

The analysis runs the Algorithm 1 on the CFG of the main subprogram, and its result is the union of the infeasible paths (ips) of all parsed CFGs.

1) Definitions: For any edge $e \in E$, $sink(e)$ is the destination basic block of e and for any block $b \in V$, $ins(b)$ is the set of incoming edges to b and $outs(b)$ is the set of outgoing edges. Two states $\hat{s}$ and $\hat{s}'$ are equivalent, noted $\hat{s} \equiv \hat{s}'$, if and only if $\gamma(\hat{s}) = \gamma(\hat{s}')$.

ϕ_C is a projection over constants: for each r_k in l (resp. m_c in m), ϕ_C keeps $l(r_k)$ (resp. $m(m_c)$) if it is a constant and behaves as the identity otherwise. For any $v \in \mathbb{V}$ ($Reg \cup Mem$),

$$\phi_C(v) \coloneqq x \mapsto \begin{cases} v(x) & \text{if } v(x) \in C \\ x & \text{else} \end{cases}$$

This builds an initial state enhanced with information on registers and memory cells that remain constant between loop iterations, which is used to generate the state that will be widened after one iteration on the loop.

For any edge e, $\hat{\mathbb{I}}[e] : \hat{S} \to \hat{S}$ is the interpretation function that processes all the semantic instructions associated to e. If e is a call edge, it parses recursively the called CFG, and applies $\hat{s}_\omega$, the state at the exit of the called function, to the current state $\hat{s}$; the program analysis continues with the state $(\hat{s}_\omega \circ \hat{s})$.

Finally, the $check(\hat{s}, \dots)$ function translates the abstract state $\hat{s}$ into SMT predicates and feeds them to the solver in order to check for the satisfiability of our system.

2) The algorithm: The core analysis of the CFG is made of the non-greyed part of Algorithm 1. All edges e are annotated by a program state, noted $s_e \in \mathcal{P}(\hat{S})$. We set up a working queue, wl, to collect basic blocks that need to be processed, initially the entry of the CFG. The algorithm iterates by popping a block from the wl queue until it is empty. If all the incoming edges of said block have been processed, then the states of those edges are joined, and the annotation cleared. The code associated to each outgoing edge e of the current block is then interpreted ($\hat{\mathbb{I}}[e]$) and annotated on s_e. Then, the solver checks for the satisfiability of each abstract path and we add the target of each outgoing edge to wl.

C. Parsing loops

1) Definitions: For any CFG G, we name H_G the set of its loop headers and B_G the set of its back edges. $L(h)$, defined for any $h \in H_G$, is the set of blocks within the body of the loop whose header is h. Similarly, $X(h)$ is the set of edges that exit from the loop identified by h.

2) Overview: Upon reaching a loop header with a state $\hat{s}$, the body of the loop is applied to an identity initial state, which is then widened and applied to $\hat{s}$. The interpretation function $\mathscr{L}_h(\hat{s})$ of a loop h is thus defined for any state $\hat{s}$ as:

$$\mathscr{L}_h(\hat{s}) \coloneqq \nabla_h(f_h(\hat{s})) \circ \hat{s}$$

where $f_h : \hat{S} \to \hat{S}$ is the abstract function that applies one iteration of the loop h. The effect of the application of this function to $\hat{s}$ is obtained by parsing the loop once with $\hat{s}$.

In order to be able to properly deal with memory addresses and better identify arithmetic progressions, a fix point is run on the loop beforehand to identify constants.

Loop headers are annotated by two abstract states $\hat{s}_h$ (used to find a fix point) and $\hat{si}_h$ (used to remember the state we entered the loop with), as well as a status indicator q, which follows the automaton of Fig. 4. We present an algorithm that applies $\mathscr{L}_h$ on any state entering a loop h.

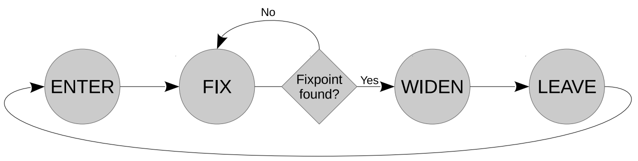


Fig. 4. Parsing a loop

Algorithm 1 Processing a program

```
Data: $G = (V, E, \epsilon, \omega)$, the CFG of the program.
Result: $ips_G$, the set of infeasible paths; and $s_\omega$.
  for e $\in E$ do
      $s_e \leftarrow nil$
  end for
  for h $\in H_G$ do
      $s_h \leftarrow nil$
      $si_h \leftarrow nil$
      $q_h \leftarrow$ ENTER
  end for
  $s_\epsilon \leftarrow \{(Id_L, Id_M, \emptyset)\}$
  $wl \leftarrow \{sink(\epsilon)\}$
  $ips_G \leftarrow \emptyset$
  while $wl \neq \emptyset$ do
      $b \leftarrow pop(wl)$
      $pred \leftarrow \begin{cases} ins(b) \text{ if } b \notin H_G \\ ins(b) \setminus B_G \text{ if } b \in H_G \wedge q_b = \text{ENTER} \\ ins(b) \cap B_G \text{ if } b \in H_G \wedge q_b \neq \text{ENTER} \end{cases}$
      if $\forall e \in pred, s_e \neq nil$ then
          $s \leftarrow \bigsqcup_{e \in pred} s_e$
          for $e \in pred$ do
              $s_e \leftarrow nil$
          end for
          $suc \leftarrow outs(b)$
          if $b \in H_G$ then
              $s \leftarrow \bigsqcap s$
          else if $q_b =$ LEAVE then
              $q_b \leftarrow$ ENTER
              $s_b \leftarrow nil$
              $suc \leftarrow \emptyset$
              if $\exists e \in ins(b), s_e = nil$ then
                  $wl \leftarrow wl \cup \{b\}$
                  if $q_b =$ ENTER then
                      $q_b \leftarrow$ FIX
                      $si_b \leftarrow s$
                  else if $q_b =$ FIX $\wedge s \equiv s_b$ then
                      $q_b \leftarrow$ WIDEN
                      $s \leftarrow (\phi_C(l), \phi_C(m), \emptyset)$
                  else if $q_b =$ WIDEN then
                      $q_b \leftarrow$ LEAVE
                      $s \leftarrow \nabla_b(s) \circ si_b$
                  end if
                  $s_b \leftarrow s$
              end if
          end if
          for $e \in suc \setminus \{X(h) \mid b \in L(h) \wedge q_h \neq \text{LEAVE}\}$ do
              $s_e \leftarrow \mathbb{I}[e](s)$
              $ips_G \leftarrow ips_G \cup check(s_e, \{(h, q_h) \mid b \in L(h)\})$
              $wl \leftarrow wl \cup \{sink(e)\}$
          end for
      end if
  end while
```

3) The algorithm: All loop headers are initialised at ENTER, indicating that the loop has not been parsed yet, as in state (A) on Fig. 3. After the algorithm enters the loop, it sets the status to remain on FIX (B, C) until a fix point is reached (D) then iterates on it once more (WIDEN status) with constant information only (E).

Finally, we widen, yielding a state that represents any iteration of the loop ((F), equivalent when applied to the function f_h), and we go through the exit edges with it (LEAVE status), before resetting the loop to ENTER (G). We only follow a loop exit edge when all the loops it is exiting from are in a LEAVE status.

Furthermore, the satisfiability check function $check(\hat{s}, \{(h_1, q_1), \ldots, (h_n, q_n)\})$ is made aware of the loops h_i that the edge being analysed is part of, as well as their respective statuses q_i. This enables it to select which loop iterations the conflict will be valid for (any iteration, first iteration, or last iteration). This is necessary to generate sound infeasible paths.

On the Example 1, the application of the algorithm leads us to reach an unsatisfiable state (I_1) on one of the four paths leaving from $\textcircled{6}$. This yields the infeasible path $\textcircled{4} \rightarrow \textcircled{5} \rightarrow \textcircled{6} \rightarrow \textcircled{8}$.

The next section showcases our techniques on several relevant real-time benchmark suites.

VI. Experiments

A. Overview

Estimating the WCET: We present on the Fig. 5 our experiments on the Mälardalen benchmarks suite [6] (excluding a few recursive programs, which are rare in real-time critical systems), the PapaBench real-time benchmarks [13], as well as benchmarks derived from the DEBIE-1 Flight Software[5], which was operated for several years on the ESA PROBA I satellite. All programs are statically compiled with no optimization by GCC 4.9.3 for the ARMv5T architecture and a LPC2138 processor.

For each benchmark, the first columns show its basic block count (BB), its instruction count (Inst.), its loop count (including loops that are introduced by the compiler) and the maximum loop depth in the program. The next two columns display the execution time of the infeasible path analysis performed on a 2.50GHz i7-6500U CPU, 4GB memory and the percentage of time that was spent in SMT solving. The next two columns show the amount of conflicts found (each represents one infeasible path), before and after a minimisation process (attempting to merge infeasible paths that come from the same predicate conflict) which aims to losslessly simplify the output and reduce the complexity of the WCET computation. Conflicts between CFG edges are translated to ILP constraints using the technique developed in [14], although better results could be achieved with more advanced exploitation techniques of path conflicts [12]. The gain obtained by adding these constraints is shown in the last column.

[5]https://www.irit.fr/wiki/doku.php?id=wtc:benchmarks:debie1

Benchmark / Task	BB (#)	Inst. (#)	Loops (#)	Max depth	Time (s)	SMT %	IPs (#)	Min. IPs (#)	WCET gain
TINY MÄLARDALEN BENCHMARKS (LESS THAN 30 BB)									
fibcall, insertsort, fdct, bs, jfdctint, janne_complex, ns, duff, bsort100, lcdnum, matmult, crc, fir: *no infeasible path found.*									
SMALL MÄLARDALEN BENCHMARKS (NO STATE SHRINKING)									
prime	43	193	14	2	1.491	98%	50	50	27.452%
expint	43	251	12	2	0.744	98%	56	20	6.553%
select	56	321	4	3	20.825	88%	2106	1178	0.000%
qsort-exam	58	369	6	3	24.488	99%	2096	1168	0.000%
edn	70	1123	18	3	0.428	92%	4	4	0.000%
ndes	83	796	12	2	2.187	98%	140	140	0.000%
cnt	94	511	7	2	42.469	99%	1040	1040	0.000%
compress	109	728	17	2	121.387	96%	23	23	0.000%
cover	205	822	3	1	0.504	96%	3	3	3.059%
LARGE MÄLARDALEN BENCHMARKS (REQUIRING STATE SHRINKING)									
ud	122	802	20	3	14.630	97%	231	57	0.000%
adpcm	171	1797	33	3	10.236	99%	11	11	0.002%
qurt	245	1390	111	2	44.929	98%	894	780	15.165%
sqrt	272	1236	11	2	479.508	99%	6591	1845	10.479%
ludcmp	276	1669	35	4	215.110	98%	1741	784	0.000%
minver	286	1730	35	4	140.115	99%	1584	405	3.363%
fft1	337	1882	216	4	3693.147	93%	134008	11596	14.607%
statemate	387	2530	1	1	250.279	99%	6433	5458	1.297%
lms	527	2582	116	3	657.364	97%	1396	615	2.177%
nsichneu	756	8088	1	1	261.522	74%	19415	19145	0.000%
DEBIE1 BENCHMARKS									
TM_InterruptService	19	96	0	0	0.066	98%	0	0	-
HandleHitTrigger	64	291	3	2	0.969	99%	15	11	41.959%
TC_InterruptService	69	276	0	0	2.191	98%	30	30	0.000%
HandleTelecommand	190	768	13	2	3.856	85%	144	141	0.000%
HandleAcquisition	303	1321	19	1	14.095	99%	126	126	0.156%
HandleHealthMonitoring	425	1670	100	3	230.381	98%	5094	4538	30.746%
PAPABENCH BENCHMARKS (FLY-BY-WIRE PROGRAM)									
servo_transmit	13	54	1	1	0.121	99%	10	4	0.000%
send_data_to_autopilot	121	562	10	1	11.667	98%	6	6	0.000%
check_failsafe	148	639	24	1	17.695	93%	114	114	0.000%
check_mega128_values	150	655	24	1	15.629	93%	114	114	0.000%
test_ppm	270	1170	36	1	36.438	96%	518	518	0.247%
PAPABENCH BENCHMARKS (AUTOPILOT PROGRAM)									
link_fbw_send	3	32	0	0	0.005	100%	0	0	-
altitude_control	96	388	2	1	4.673	98%	49	49	0.000%
stabilisation	200	859	13	1	8.844	97%	239	239	0.534%
climb_control	252	1066	15	1	23.751	96%	320	320	0.460%
reporting	418	3943	0	0	70.716	99%	2879	2879	0.000%
radio_control	424	2398	39	1	71.345	98%	2803	2803	0.000%
receive_gps_data	574	3310	57	1	89.189	98%	2618	2590	2.590%
navigation	1004	4643	1018	1	409.404	94%	3273	2899	0.000%

Fig. 5. Experimental results

B. Results

It is very difficult to confidently conclude on the efficiency of an infeasible path analysis (as for most WCET estimation analyses) because the actual set of infeasible paths (and the actual WCET) is unknown. Also, the amount of infeasible paths is a poor indicator: a large amount of data flow constraints may not result of a successful analysis but instead of poor infeasible path factorisation, that would contain paths including unimpactful edges, causing them to be duplicated many times in the output.

It is also clear that most infeasible paths do *not* affect the WCET estimation (because they are not on the WCEP), and that their overall impact is very variable. For example, the thousand constraints of `cnt` do not affect the WCET estimation, while the fifty of `prime` did very much. This phenomenon is expected: the results of the analysis are generally polluted by a lot of infeasible paths that would be cheap to execute. This does not undermine the rarer impactful conflicts.

A significant part of the benchmarks from the chosen suites is simply too small to be relevant: one can hardly imagine an infeasible path could be hidden in a program made of a dozen of basic blocks. Indeed, no infeasible paths were found in any benchmark smaller than 30 basic blocks (that is, 13 Mälardalens, one Debie1 and one PapaBench).

However, while the results obtained on PapaBench are inconclusive, the important gains on two Debie1 tasks are encouraging signs that such analyses significantly improve the WCET estimation of actual real-time applications. Some of the most complex, sizeable Mälardalen benchmarks also show significant reduction of the WCET estimation, with three large benchmarks exceeding 10%.

This supports the idea that algorithmic complexity favours the introduction of infeasible paths, partly because it becomes harder to structure the code in order to avoid conflicting conditions, and partly because subroutines in larger programs tend to be called with contexts that lead to the program flow skipping expensive chunks of code.

Important increases in analysis times are observed on programs with high amounts of calls and loops (such as `fft1`). Experiments suggest the analysis scales rather well with code size, as the analysis deals with path explosion by regularly shrinking states (this is best showcased on `nsichneu`). SMT solving represents most of the complexity cost of the analysis (both with CVC4 and z3). The problems we queried to the SMT solver are generally easy, with many small connex components; yet calls to the solver are the most time costly part of our analysis because of their high frequency, requiring a lot of initialisation work. The complexity of the analysis on larger benchmarks than the Mälardalens could be reduced with a policy that performs more sparse SMT calls, instead of making a call on each condition.

Considering the observed simplicity of the large majority of conflicting predicates detected, SMT solving may also be an overkill: a lighter (possibly ad-hoc) solver would probably be much faster while missing little to none of the conflicts an SMT solver detects in most common benchmarks.

VII. Conclusion

We have presented in this paper an infeasible path lookup analysis that benefits from being split and composed from the analysis of smaller parts of the program, the SESE regions made of the bodies of loops and subroutines. In doing so, we output data flow conflicts valid in the most possibly general scope, maximising their potential impact on the reduction of the WCET overestimation. Infeasible paths within loops are made independent of the context the iteration started with whenever the conflict is detected to be local to an iteration; similarly, those within functions are made independent of the calling arguments whenever possible.

Our experiments on the Mälardalen benchmarks display conclusive results, including the reduction of the WCET estimation for three of the larger benchmarks by over 10%. The results on the Debie1 and PapaBench benchmarks prove the analysis scalable enough to work on actual, moderately sizeable hard real-time applications. While the presence and impact of infeasible paths on real life applications are bound to vary a lot, the removal of infeasible paths from the ILP system computing the WCET has very importantly reduced the overestimation on some of the analysed tasks.

Although those results were promising, our works leave some clear paths of improvements. While the resource consumption of the analysis remained somewhat limited on most benchmarks, the use of a more appropriate technique to identify conflicting predicates is a simple, straight-forward way to massively boost the speed of the analysis. As noted in Section IV, loop bounds on the binaries could also be found during the analysis at little cost and combined to the set of additional ILP constraints we already produce.

Furthermore, several types of infeasible paths remain undetectable by our analysis, such as inter-loop conflicts (for example, a path that cannot be taken in two successive iterations). Finally, ILP solving supports numerical constraints that express that a path can be, for example, taken at most n times in a loop; such constraints could be output by the information we are already acquiring during loop analysis, given an appropriate solving technique.

References

[1] S. Andalam, P. S. Roop, and A. Girault. Pruning infeasible paths for tight wcrt analysis of synchronous programs. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, 2011.

[2] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: an open toolbox for adaptive WCET analysis. In *Software Technologies for Embedded and Ubiquitous Systems - 8th IFIP WG 10.2 International Workshop, SEUS 2010, Waidhofen/Ybbs, Austria, October 13-15, 2010. Proceedings*, pages 35–46, 2010.

[3] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA. Proceedings*, pages 171–177, 2011.

[4] T. Chen, T. Mitra, A. Roychoudhury, and V. Suhendra. Exploiting branch constraints without exhaustive path enumeration. In *OASIcs-OpenAccess Series in Informatics*, volume 1, 2007.

[5] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. A simple, fast dominance algorithm. *Software Practice & Experience*, 4:1–8, 2001.

[6] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET Benchmarks: Past, Present And Future. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 136–146, 2010.

[7] Jan Gustafsson, Andreas Ermedahl, and Björn Lisper. Algorithms for Infeasible Path Calculation. In *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*, volume 4 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2006. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[8] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Bjorn Lisper. Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution. In *Real-Time Systems Symposium, 2006. RTSS'06. 27th IEEE International*, pages 57–66. IEEE, 2006.

[9] N. Holsti. Computing time as a program variable: A way around infeasible paths. In *in: 8th International Workshop on Worst-Case Execution Time Analysis (WCET'2008)*, 2008.

[10] R. Johnson, D. Pearson, and K. Pingali. The program structure tree: Computing control regions in linear time. In *ACM SigPlan Notices*, volume 29, pages 171–185. ACM, 1994.

[11] H. Li, I. Puaut, and E. Rohou. Traceability of flow information: Reconciling compiler optimizations and wcet estimation. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, page 97. ACM, 2014.

[12] Vincent Mussot, Jordy Ruiz, Pascal Sotin, Marianne de Michiel, and Hugues Cassé. Expressing and exploiting conflicts over paths in wcet analysis. In *OASIcs-OpenAccess Series in Informatics*, volume 55. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[13] Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean-Paul Bahsoun, and Marianne De Michiel. PapaBench: a Free Real-Time Benchmark. In *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*, volume 4 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2006. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[14] Pascal Raymond. A general approach for expressing infeasibility in implicit path enumeration technique. In *Proceedings of the 14th International Conference on Embedded Software*, page 8. ACM, 2014.

[15] 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.

[16] A. C. Shaw and C. Y. Park. Experiments with a program timing tool based on source-level timing schema. *Computer*, 24:48–57, 1991.