

Using SMT Solving for the Lookup of Infeasible Paths in Binary Programs*

Jordy Ruiz and Hugues Cassé

IRIT – Université de Toulouse, France
{jruiz, casse}@irit.fr

Abstract

Worst-Case Execution Time (WCET) is a key component to check temporal constraints of real-time systems. WCET by static analysis provides a safe upper bound. While hardware modelling is now efficient, loss of precision stems mainly in the inclusion of infeasible execution paths in the WCET calculation. This paper proposes a new method to detect such paths based on static analysis of machine code and the feasibility test of conditions using Satisfiability Modulo Theory (SMT) solvers. The experimentation shows promising results although the expected precision was slightly lowered due to clamping operations needed to cope with complexity explosion. An important point is that the implementation has been performed in the OTAWA framework and is independent of any instruction set thanks to its semantic instructions.

1998 ACM Subject Classification J.7 Real time

Keywords and phrases WCET, infeasible paths, SMT, machine code

Digital Object Identifier 10.4230/OASIScs.WCET.2015.95

1 Introduction

Temporal properties verification is utterly important for critical embedded systems like avionics, automotive or any system involving real-time constraints. Such verification is usually performed in two steps: first, the Worst-Case Execution Time (WCET) of each task composing the system is computed and, second, this WCET is used to check if the set of tasks is schedulable according to the system real-time constraints. Hence, the WCET determination must be (a) safe (higher than the real worst-case execution) to ensure soundness of the scheduling and (b) tight or precise to minimize the hardware required to run the system.

This paper addresses the computation of WCET by static analysis that ensures safety of WCET by construction at the price of tightness: an overestimation of the WCET is produced without guaranty about precision. A successful and widely used approach to compute statically WCET is Implicit Execution Path Technique or IPET. It models the execution paths of the program and the host hardware (that induces execution time) as an Integer Linear System, ILP, whose maximization function is the estimated WCET. Lots of work has been devoted to model the hardware and, for some classes of microprocessor parts (LRU caches, pipeline, etc.), provides precise results.

Yet, modelling the execution paths seems to remain a challenge and could be an important source of precision improvement. Basically, IPET models the execution paths by a Control Flow Graph, CFG, where vertices represent blocks of instructions¹ and edges, execution flow between blocks. The CFG is usually derived from a program by a basic analysis examining

* This work was partially supported by the french ANR project, W-SEPT.

¹ Usually machine instructions to be as close as possible to the hardware.



© Jordy Ruiz and Hugues Cassé;

licensed under Creative Commons License CC-BY

15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015).

Editor: Francisco J. Cazorla; pp. 95–104

OpenAccess Series in Informatics



OASIS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the branch targets of control instructions. Therefore, the set of paths represented by the CFG (potentially unbounded if the program contains loops) includes (a) all actual paths of the program and (b) other paths that are infeasible because of the program internal logic.

Applying the IPET approach to the CFG model of execution paths is consistent (as long as a maximum bound is provided for loops) because it consists in computing the maximum of execution paths. Adding infeasible paths to the set of actual execution paths will, in the worst case, produce an estimated WCET higher than the actual WCET. Yet, the added execution paths may cause a loss of tightness (distance between real WCET and estimated WCET) that is difficult to evaluate. In practice, separating feasible and infeasible paths faces the complexity issue of the number of paths in the program and in the CFG.

This paper proposes a new approach to detect infeasible paths in machine language programs based on (a) the representation of program states as labelled sets of predicates and (b) the detection of unsatisfiable conditions using a Satisfiability Modulo Theories (SMT) solver. The result is a mostly minimal list of infeasible paths or families of paths that may be used in WCET computation like IPET. The experimentation shows that our approach on the Mälardalen benchmark [4] is promising. It does not allow capturing all infeasible paths of the program but it should help improving precision of the estimated WCET.

The next section presents works related to infeasible path detection or execution path validation. Section 3 gives an overview of the proposed method and of its context while section 4 enters into details of the proposed analysis. Section 5 exposes the experimentation results and we conclude and propose future development in the last section.

2 Related Works

Several works already exist on this topic and are surveyed here.

Suhendra et al. [6] present an integrated approach to compute the WCET while excluding infeasible paths. The program is expressed as a Direct Acyclic Graph (DAG) and traversed in a bottom-up way. Throughout the analysis, the worst-case execution time is computed by accumulating block time and selecting the maximum time. During this traversal, predicates are built reflecting assignments and conditions, and paths with unsatisfiable set of predicates are discarded. This approach uses a very limited set of predicates that reduces analysis time but also the amount of infeasible paths found. In addition, there is no plan to cope with hardware effects.

Henry et al. [5] propose another integrated solution that represents as an SMT the semantic of execution paths and the time consumed by a particular path. To determine the WCET, one has to propose an upper bound, to add the matching predicate and to check the feasibility of the system. Main drawbacks of this approach are the computation time, the need for unrolling loops and the lack of support for non-scalar memory structures like arrays. In addition, there is no clear solution to cope with hardware effects except adding constant access time to instruction blocks. Unlike our approach, this approach determines the set of feasible paths and uses it to compute WCET.

The PathCrawler [7] tool generates disjunctions of conjunctions for any control point of the CFG, then proceeds to do constraint solving in order to rigorously verify properties. However, it runs on source code, and while we use a similar approach, our final goal differs: the detection of infeasible paths. In the next section, we present our approach that (a) is applied to machine code and (b) tries to preserve precision of predicates.

3 Proposed Method

3.1 Source of infeasible paths

3.1.1 Infeasible path patterns

We have identified three common types of infeasible paths found by our analysis. This list is not exhaustive and we sometimes find more complex infeasible paths but these are rarer. The first type of infeasible path simply comes from two mutually exclusive conditions in the source code, tested in sequential order:

```
if(x == 1)
    /*...*/
if(x == 2)
```

This can look like a mistake from the programmer, but there may be a lot of code between the two conditions, making the infeasible path much less obvious to see, and/or difficult to remove.

Some other infeasible paths are due to the restriction of the function parameters domain at a particular call and require interprocedural analysis to be identified. The `if(x)` condition will never be taken in the case of a call from `main()` in the following example:

```
void icrc(int x) {
    if(x)
        /*...*/
}
int main() {
    icrc(0);
```

The last type of common infeasible path is due to short-circuit condition evaluation during the compilation:

```
if(x && a)
    /* ... */
if(x && b)
```

In this case, the binary program may include an infeasible path because `if(x && a)` will be broken down into `if(x) { if(a)` by the compiler, unless smart optimizations are performed.

3.1.2 Analysis on Machine Language

Working on machine code allows precisely understanding the work of the program in the hardware and improving the WCET tightness. Yet, the analysis of the machine code suffers from (a) the lack of expressivity of machine instructions, (b) the size of the program and (c) the implementation complexity of data flow analyses. For example, registers are loosely typed and the structure of data in memory is not explicit.

Even though finding infeasible paths is easier on programs in high-level languages, an additional pass is required to map and exploit this information on binary programs. This pass is even more difficult if the optimizations of the compiler are activated. Even worse, analysis of the high-level language requires different analysers for each language, the complete sources to be available and prevents the processing of low-level subprograms written in assembly.

Hence, we choose to work at the machine code level but, as the real-time embedded domain uses lots of different microprocessors, we have to cope with as many instruction sets. Fortunately, the OTAWA C++ framework provides an abstraction of the different

instruction sets through an independent *semantic language* [3]. It consists in the translation of the machine instruction into a sequence of semantic instructions that try to capture as precisely as possible its semantics. When the instruction cannot be expressed, a special **SCRATCH** instruction allows marking the result register as modified in an unspecified way. This approach has been successfully applied to translate several instruction sets like PowerPC, ARM, Sparc and Tricore.

Basically, the *semantic language* looks like a RISC-instruction set and is made of:

- three-operand computation instructions: **ADD**, **SUB**, **CMP**, **SHL**, **SHR**, etc;
- dedicated instructions to load a literal in a register: **SETI**;
- memory access instructions: **LOAD**, **STORE**;
- branches or assignments to program counter: **BRANCH**;
- conditional execution of the sequence: **IF**.

Most instructions work only on registers but temporary registers may also be used to let instructions exchange information without modifying the actual program state. In the following, we show how the *semantic language* can be used to perform infeasible path analysis.

3.2 Analysis overview

The lookup of infeasible paths is performed by top to bottom traversal of the program; all non-cyclic paths are explored, each machine instruction is broken down into one or several semantic instructions, and an abstract representation of the program state is updated accordingly for each path. Sets of possible program states are represented as lists of predicates, initially empty (\top), meaning that any state is possible.

3.2.1 Predicate generation

Figure 1 shows several examples of analysis results. (a) is an example of predicate generation, upon parsing an ARM instruction `mov r4, r0`. The first bold line is the ARM instruction, semantic instructions are capitalized, and italic lines are modifications to the list of predicates. Two predicates are generated (noted with \oplus), and t_1 is a temporary register introduced by OTAWA. While it may look purposeless here, OTAWA automatically chooses to use this intermediate to handle more complex uses of the `mov` instructions such as, for example, `mov r0, r1, LSR #2`. Here, the instruction `mov r4, r0` is translated into two semantic instructions, `SET t1, r0` and `SET r4, t1`. In turn, their interpretation generates two predicates, $t1 = r0$ and $r4 = t1$.

3.2.2 Predicate update

The predicates can be modified either because the instruction we are parsing requires us to or because we wish to and believe it will be beneficial.

Extending the example (a), there are cases where modifying predicates is unnecessary but useful (b) because of the ephemerality of the temporary register t_1 , whereas sometimes updating the predicate is required by the instruction (c). In (b), `[r4 replaces t1]` means that every occurrence of the temporary register t_1 will be replaced by register r_4 . The predicate $r4 = t1$ becomes identity ($r4 = r4$), so we only keep $r4 = r0$ as a result of this `mov`.

In the example (c) initially containing “ $r_3 = [SP - 12]$ ”, in order for our predicate list to remain a valid abstract representation of the program state, this instruction that adds 1 to r_3 requires replacing every r_3 occurrence with $r_3 - 1$, hence the `[r3 - 1 replaces r3]` line.

<pre> mov r4, r0 SET t1, r0 ⊕ t1 = r0 SET r4, t1 ⊕ r4 = t1 </pre> <p>(a) An addition.</p> <pre> { r3 = [SP-12] } add r3, r3, #1 SETI t1, 1 ⊕ t1 = 1 ADD r3, r3, t1 [r3 - 1 replaces r3] ⊖ r3 = [SP-12] ⊕ r3 - 1 = [SP-12] </pre> <p>(c) A necessary update.</p>	<pre> mov r4, r0 SET t1, r0 ⊕ t1 = r0 SET r4, t1 ⊕ r4 = t1 [r4 replaces t1] ⊖ t1 = r0 ⊕ r4 = r0 </pre> <p>(b) A useful update.</p> <pre> [...] SET r13, t3 ⊖ r13 = SP - 4 ⊕ r13 = SP + 0 ⊖ tempvars t1, t2, t3 </pre> <p>(d) A removal.</p>
---	---

■ **Figure 1** Example of operations on the predicates.

3.2.3 Predicate removal

Instructions may force us to remove predicates from our list. For instance, if we set to r_{13} a new value, we must remove any predicate mentioning r_{13} as it will no longer be valid. In the example (d), the previously generated $r_{13} = SP - 4$ predicate must be removed (noted with \ominus). In this case, the analysis identifies t_3 to the constant $SP + 0$ (value of the stack pointer at the start of the program), so it chooses to generate $r_{13} = SP + 0$ in place of $r_{13} = t_3$.

Also, temporary registers are always deleted at the end of every semantic instruction block (matching an assembly instruction), thus we should remove all predicates containing them, hence the \ominus tempvars t_1, t_2, t_3 in the example.

4 Analysis Definition

4.1 Abstract domain

The goal of this analysis is to build a function $\mathbb{L} \rightarrow \mathbb{M}^\#$ that gives, at any control point \mathbb{L} , an abstract state $\mathbb{M}^\# : \bigvee_i (\bigwedge_j (\phi_{i,j}(R, M_h, M_s)))$ where:

- $R \cong \mathbb{Z}^n$ is the set of registers, n being the amount of available registers on the architecture;
- M_h represents the heap memory, $M_h \cong \mathbb{Z}^{\mathbb{Z}}$;
- M_s represents the stack memory, $M_s \cong \mathbb{Z}^{\mathbb{Z}}$.
- $\phi_{i,j}$ are predicates defined as:

$$\begin{aligned}
 \phi : & \quad \text{Operand} \times \text{Opr} \times \text{Operand} \\
 \text{Opr} : & \quad = | \neq | \leq | < \\
 \text{Operand} : & \quad \text{Operand} \omega \text{Operand} | - \text{Operand} \\
 \omega : & \quad + | - | \times | / | \text{mod}
 \end{aligned}$$

Each conjunction of predicates represents a set of possible program states for one or several execution paths up to a given control point. A control point may contain information coming from several paths: we do not merge the conjunctions of predicates but instead keep them in a set, this way $\mathbb{M}^\#$ is actually a disjunction of conjunction of predicates, and overestimates the set of possible program states.

In order to show this, we define \mathbb{M} as a concrete program state, that is, the set of values of the registers and of the memory, and the following concretisation function $\gamma : \mathbb{M}^\# \rightarrow 2^{\mathbb{M}}$:

$$\gamma\left(\bigvee_i \bigwedge_j \phi_{i,j}(R, M_h, M_s)\right) := \bigcup_i \bigcap_j \gamma'(\phi_{i,j}(R, M_h, M_s))$$

where for any $p : \text{Predicate}$, $\gamma'(p) := \{m \in \mathbb{M} \mid p(m)\}$ and the analysis is sound if

$$\gamma(f^\#(s)) \supseteq f(\gamma(s))$$

with $f : \mathbb{M} \rightarrow \mathbb{M}$ being any semantic instruction and $f^\# : \mathbb{M}^\# \rightarrow \mathbb{M}^\#$, also later noted $U(f)$, the corresponding modifications to be applied on our abstract state.

4.2 Update function

Our update function takes a semantic instruction and an abstract program state as parameters to return a new abstract state:

$$U : I_{\mathbb{M}} \times \mathbb{M}^\# \rightarrow \mathbb{M}^\#$$

where $I_{\mathbb{M}} = \mathbb{M} \rightarrow \mathbb{M}$ is the set of semantic instructions, which modify the machine state. Let Var be the set of variables, including R , M_h , M_s and the temporary registers $Temp$.

In the first place, we define the following functions to operate on $\mathbb{M}^\#$:

$$\text{invalidate} : Var \times \mathbb{M}^\# \rightarrow \mathbb{M}^\#$$

This function removes any *Predicate* containing a reference to the provided *Var*. Before removing, it computes the transitive closure of the current predicates to avoid unnecessary loss of information. Importantly, we observe for any $s : \mathbb{M}^\#$,

$$\gamma(s \setminus \{p\}) \supseteq \gamma(s)$$

that is, by removing a predicate we lose precision but the result remains sound. It is a convenient property to handle unsupported operations. We also use variable replacement mechanics on predicates: for any $x : Var$, arithmetic expression e and program state $s : \mathbb{M}^\#$, the program state in which every occurrence of x has been replaced by e is noted $s [e / x]$.

For any instruction modifying a variable d , if the computation of the new value of d is independent of its previous value, we remove all predicates containing d as they become obsolete. If, at the contrary, this computation depends on the previous value of d , we try to apply to the affected predicates the inverse operation, but this inverse only exists if the original operation is bijective. For instance, $f : d \mapsto 3d$ is not surjective in \mathbb{Z} , therefore f^{-1} does not exist and we will not be able to update our predicates. In this case, we often have to throw away information about the variable f is applied to, although we can sometimes keep some properties: for example f preserves the parity in this case.

For any variables d, a , constant k :

$$U[\text{SETI } d, k] s := "d = k" \cup (\text{invalidate } d s)$$

$$U[\text{SET } d, a]_{d=a} s := s$$

$$U[\text{SET } d, a]_{d \neq a} s := "d = a" \cup (\text{invalidate } d s)$$

For any variables d, a, b such that $d \neq a, d \neq b$:

$$U[\text{ADD } d, a, b] s := "d = a + b" \cup (\text{invalidate } d s)$$

$$U[\text{ADD } d, a, d] s := s [d - a / d]$$

$$U[\text{ADD } d, d, b] s := U[\text{ADD } d, b, d] s$$

$$U[\text{ADD } d, d, d] s := "d \% 2 = 0" \cup (s [d/2 / d])$$

In the ADD d, d, d case, we lose information when replacing d with $d/2$, thus we add a predicate that says that d is even, to strip away the ambiguity.

$$U[\text{MUL } d, a, b] s := "d = a * b" \cup (\text{invalidate } d s)$$

$$U[\text{MUL } d, a, d] s := "d \% a = 0" \cup (s [d/a / d])$$

$$U[\text{MUL } d, d, b] s := U[\text{MUL } d, b, d] s$$

$$U[\text{MUL } d, d, d] s := "0 \leq d" \cup (\text{invalidate } d s)$$

Again, we add a predicate to avoid the loss of information in the MUL d, a, d case. We cannot express the square root, so there is not a whole lot to be done for MUL d, d, d . There is no support for the binary instructions AND, OR, XOR, thus for any d, a', b' :

$$U[\text{[AND/OR/XOR] } d, a', b'] s := U[\text{SCRATCH } d] s := \text{invalidate } d s$$

This list is incomplete and there are many other semantic instructions to be interpreted (namely logical arithmetic shifts, memory accesses...), but these are handled in a similar way.

4.3 Flow analysis

Our analysis is applied on a **Control Flow Graph** (CFG), a directed rooted graph $G = (V, E, \epsilon)$ composed of lists of sequential instructions grouped in basic blocks which are the vertices (V) of said graph, and edges ($E = V \times V$) between basic blocks that represent the possible execution paths of the program. These edges can be conditional or not, and the CFG always includes one entry ($\epsilon \in V$) as a virtual basic block.

We parse the CFG with a working list algorithm, which general idea is to only process basic blocks once all the sources of the incoming edges have been processed. Below is a first version of the flow analysis algorithm for loopless programs:

```
wl <- {ε};
While wl != ∅
  pop bb from wl;
  If allIncomingEdgesAreAnnotated(bb)
    For edge in bb.ins
      sl <- sl ∪ edge.getAnnotation();
    For state in sl
      state.processBasicBlock(bb);
    For edge in bb.outs
      new_sl <- ∅;
      For state in sl
        new_state <- state.processEdge(edge);
        new_sl <- new_sl ∪ new_state;
      edge.annotate(new_sl);
      wl <- wl ∪ edge.target;
  End If
End While
```

We first initialize our working list wl to the root, then pop basic block elements from it until it is empty. For each basic block the algorithm is asked to work on, all its incoming edges are checked for *annotations* that give an abstraction of the program state at that control point. If any edge is missing an annotation, we cannot process the basic block yet and postpone it. Otherwise, we put all the annotations into a list of abstract states named sl above. Then, we update the states of sl to represent possible program states at the end of the basic block. Lastly, we annotate all the outgoing edges with this state list, update them

accordingly if it is a conditional edge and add the block the edge points to the working list if it is not already in it.

The improved version of this algorithm that supports loops uses one more type of annotation: on loop headers, we remember the last abstract program state we have computed. We also keep information in our enhanced program states on whether we have reached a fixpoint in the most inner loop or not. If not, we follow all the edges except for the loop exit ones. If we have found a fixpoint, we follow all the edges except for back edges.

We merge the abstract program state list into one state on every loop header. When the state list becomes too big, we may also choose to merge all paths into one to save analysis time and prevent combinatorial explosion: this is a tradeoff between execution time and efficiency of the algorithm. Our merge algorithm is a trivial intersection with a few simple enhancements. For any conjunctions of predicates $p := \bigwedge_i(\phi_i)$ and $p' := \bigwedge_j(\phi'_j)$ our merge operator $\sqcap : \mathbb{M}^\# \rightarrow \mathbb{M}^\#$ is defined as:

$$p \sqcap p' := \bigwedge_i(\phi_i) \sqcap \bigwedge_j(\phi'_j) := \bigwedge_{i,j}(\phi_i \sqcap \phi'_j)$$

where $\phi_i \sqcap \phi'_j := \phi_j$ if $\phi_i \equiv \phi'_j$ and \top otherwise. “ \equiv ” is a custom equivalence relation that is slightly less strict than the syntactical equality and supports predicate commutativity. There are many possibilities to improve this semantical equality and keep as much information as possible. The idea behind using this slightly improved intersection to approximate the disjunction of conjunctions of predicates is that for any set A, B , $(A \cap B) \subseteq (A \cup B)$.

4.4 Infeasible path identification

We now have to exploit the information we have on the state of the program at the different control points to find infeasible paths. In order to do this, we use a SMT solver, that is, a SAT solver enhanced with several theories including integers. This kind of tool allows us to find inconsistencies in our list of predicates via its C++ API. It may return **SAT** when it is able to exhibit a solution to the problem, or **UNSAT**, what guarantees that the problem has no solution. The SMT solver is systematically called at the end of the analysis of each basic block to test the consistency of the predicate list.

We chose CVC4 [1] for its high performance at recent contests such as SMT-COMP [2], its very open license and its rich API. CVC4 also supports (although partially) “unsat cores”, a technique that finds a minimal subset of predicates that caused unsatisfiability. For instance, for such a conjunction of predicates:

$$(x \neq 2) \wedge (y > z) \wedge (x = y + z) \wedge (z = 1) \wedge (y \leq 0)$$

the unsat core module of CVC4 would return $\{y > z, z = 1, y \leq 0\}$.

There may be several minimal unsatisfiable subsets of predicates, in this case CVC4 will return only one of them. This feature is still being worked on, and we have observed great improvements in the success rate of this feature in the recent development versions of this solver. Still, the interface with CVC4 is separate from the rest of our tool and it can be enhanced with the ability to call other SMT solvers.

Let ϕ_i be the set of predicates returned as an unsat core and E_{ϕ_i} the edges of each predicate ϕ_i . Any path traversing $F_p = \cup E_{\phi_i}$ should be infeasible, but because of side-effects on parallel paths, F_p may include actually feasible paths, so we only consider F_p valid when there is no feasible path traversing F_p (and we manually check for that). In the worst case, we will have to use a full, unminimized path, but this will not cause us to miss infeasible paths, only to get a bigger and less “factorized” output.

■ **Table 1** Results on the Mälardalen benchmarks.

Benchmark	BB (#)	Time (s)	Inf. paths found with minimization			w/o minimization
			1 edge	Minimized	Non-minimized	Non-minimized
SMALL BENCHMARKS (NO MERGING REQUIRED)						
ndes	57	0.267	0	0	0	0
expint	70	0.748	4	5	5	34
edn	75	0.537	2	0	0	2
prime	118	4.368	2	8	12	43
compress	122	1.801	2	8	0	19
select	136	45.598	0	4	0	8
qsortexam	155	28.201	2	4	2	12
adpcm	323	0.074	3	0	0	3
LARGE BENCHMARKS (MERGING REQUIRED)						
ud	153	17.477	3	3	0	23
minver	449	188.339	4	0	0	16
statemate	453	193.849	0	16	0	22
ludcmp	632	143.088	5	6	0	510
nsichneu	754	250.385	0	1352	3234	8620
qurt	2777	773.805	3	0	0	3
lms	3098	915.434	26	22	211	2376
fft1	6123	2223.125	0	25	0	815

5 Experimental Results

We have run our analysis program on the Mälardalen benchmarks, compiled into ARM binaries by `gcc -O1` (minimal optimizations) with non-constant global memory to ensure multiple paths, performed on a 2.90GHz i7-4600M CPU, 4GB memory. Measuring the efficiency of such an analysis is tough for multiple reasons: (a) we do not know how many actual infeasible paths the program contains, (b) since our analysis is targetted to exhibit infeasible paths, but not to exploit them, any measured improvement on the computed WCET estimation is also due to the efficiency of the tool that will exploit these infeasible paths, (c) our infeasible paths are actually sets of infeasible paths, and we do not know how many paths they include. Even if we stop trying to find minimal sets of edges, the analysis cuts paths once they are identified as infeasible, and we do not know how many paths have been cut this way.

Table 1 showcases both the performance of the analysis on this set of benchmark and the powerfulness of the minimization algorithm. The benchmarks are sorted by number of basic blocks (after virtual inlining of function calls). The 17 smallest benchmarks with a size ranging from 2 to 50 basic blocks have been removed, 10 of which gave no results.

The first three result columns show the amount of unique infeasible paths found by the analysis split in three kinds: paths made of only one edge, successfully minimized paths, and paths for which the minimization failed. The last column gives the amount of infeasible paths found when path minimization is disabled. For example, running on “prime” with minimization, the analysis finds $2 + 8 + 12 = 22$ infeasible paths, including 10 successfully minimized ones. These 10 short-length paths translate into $43 - 12 = 31$ lengthy infeasible paths when no minimization work is done. “fft1” is an extreme: 815 paths have been minimized into 25 paths of an average length of 2.84 edges. We have observed that most of the computing time is spent on SMT solving.

The results look very promising for the bigger benchmarks and the occasional merges of

predicates seem enough to tone down the combinatory explosion without hurting results too much, yet the actual impact on the precision of the computed WCET remains unknown.

6 Conclusion and Future Works

This article proposes a new approach to discover infeasible paths in a binary program. Our solution is a static analysis of (a) a CFG whose blocks are made of machine instructions (abstracted by semantic instructions) and (b) of program data states represented by predicates on registers and memories. Unsatisfiability by SMT of predicates allows identifying infeasible paths. The result is a list of edges of the program CFG that are forbidden on a feasible execution path. This information is typically used to tighten the precision of the WCET computation.

Although the proposed approach gives promising results, we feel that some infeasible paths remain undiscovered because of (a) too coarse states join operator and (b) of time calculation explosion. Issue (a) is the most important although it means that we have to find smart fixpoints for our predicates. Polyhedra could be an efficient and well-known alternative but we would be bound to linear predicates.

The second issue may be solved by reducing the amount of produced predicates. For example, code slicing would be a good technique to keep only code involved in conditions and therefore to produce only predicates involved in conditions. As a significant part of analysis time is spent in SMT solving, another solution might be to reduce the number of calls to the solver. We have to find a tradeoff during fixpoint computations between the frequency of SMT calls and the minimization of found paths: an SMT call finds infeasible paths but also allows reducing the number of states.

Finally, the infeasible paths found by our approach only concern mutual exclusivity between CFG edges. Our feeling is that quantitative relations, induced by the program semantics, exist between CFG edges and we wish to investigate this domain deeper afterwards.

References

- 1 C. Barrett, C. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In *23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *Lecture Notes in Computer Science*. Springer, 2011.
- 2 C. Barrett, M. Deters, L. de Moura, A. Oliveras, and A. Stump. 6 Years of SMT-COMP. *Journal of Automated Reasoning*, 50(3):243–277, 2013.
- 3 H. Cassé, F. Birée, and P. Sainrat. Multi-architecture value analysis for machine code. In *WCET'13*, pages 42–52. OASICs, Dagstuhl Publishing, 2013.
- 4 J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET Benchmarks: Past, Present And Future. *WCET*, 15:136–146, 2010.
- 5 J. Henry, M. Asavoae, D. Monniaux, and C. Maïza. How to compute worst-case execution time by optimization modulo theory and a clever encoding of program semantics. *SIGPLAN Not.*, 49(5):43–52, June 2014.
- 6 V. Suhendra, T. Mitra, A. Roychoudhury, and Ting C. Efficient detection and exploitation of infeasible paths for software timing analysis. In *Design Automation Conference, 2006 43rd ACM/IEEE*, pages 358–363, 2006.
- 7 N. Williams, B. Marre, P. Mouy, and M. Roger. PathCrawler: Automatic generation of path tests by combining static and dynamic analysis. In *Dependable Computing – EDCC 5*, volume 3463 of *Lecture Notes in Computer Science*, pages 281–292. Springer, 2005.