Expressing and Exploiting Conflicts over Paths in WCET Analysis*

Vincent Mussot[1], Jordy Ruiz[2], Pascal Sotin[3], Marianne de Michiel[4], and Hugues Cassé[5]

1 **IRIT, University of Toulouse, Toulouse, France**
`mussot@irit.fr`
2 **IRIT, University of Toulouse, Toulouse, France**
`ruiz@irit.fr`
3 **IRIT, University of Toulouse, Toulouse, France**
`sotin@irit.fr`
4 **IRIT, University of Toulouse, Toulouse, France**
`Marianne.De-Michiel@irit.fr`
5 **IRIT, University of Toulouse, Toulouse, France**
`casse@irit.fr`

Abstract

The presence of infeasible paths in a program is a source of imprecision in the Worst-Case Execution Time (WCET) analysis. Detecting, expressing and exploiting such paths can improve the WCET estimation or, at least, improve the confidence we have in estimation precision. In this article, we propose an extension of the FFX format to express conflicts over paths and we detail two ways of enhancing the WCET analyses with that information. We demonstrate and compare these techniques on the Mälardalen benchmark suite and on C code generated from Esterel.

1998 ACM Subject Classification B.2.2 Performance Analysis and Design Aids, C.3 Special-Purpose and Application-Based Systems, C.4 Performance of Systems, D.2.4 Software/Program Verification

Keywords and phrases WCET analysis, Infeasible paths, Path conflicts, IPET, CFG transformation

Digital Object Identifier 10.4230/OASIcs.WCET.2016.3

1 Introduction

The Worst-Case Execution Time (WCET) analysis of a program takes into account all the finite (with loop bounds) paths of its Control Flow Graph (CFG). The data manipulated by the program might make some of these paths infeasible. If the WCET analyser is unaware of these infeasible paths, or is not able to exploit them, the analysis *may* suffer from:

1. Direct over-approximation when the Worst-Case Execution Path (WCEP) is infeasible,
2. Indirect over-approximation when infeasible paths pollute the timing of hardware analyses.

The function shown on Listing 1 illustrates these two points. When called with a value less than 64 the heavy computation `comp` is infeasible and should not be taken as WCEP (Item 1). When called with a value greater than or equal to 64 the array `precomp` is not accessed and should not alter the abstract state of the cache (Item 2).

* This work is supported by the French ANR W-SEPT.

© Vincent Mussot, Hugues Cassé, Jordy Ruiz, Pascal Sotin, Marianne de Michiel, and Hugues Cassé; licensed under Creative Commons License CC-BY

16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016).
Editor: Martin Schoeberl; Article No. 3; pp. 3:1–3:11

Open Access Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
int f(int n) {
  if (n < 64) return precomp[n];
  else return comp(n);
}
```

Listing 1 A function that either computes its result or returns a precomputed result.

```
<conflict>
  <!-- Edge or block identifier 1 -->
  <!-- ... -->
  <!-- Edge or block identifier N -->
</conflict>
```

Listing 2 General form of a conflict.

In this paper we focus on a specific class of infeasible paths called *conflicts*[1]. A conflict is defined by a set of CFG edges that cannot all appear in the same execution trace. As soon as one of these edges can be executed several times, there is no straightforward translation of the conflict into the Integer Linear Program (ILP) usually built to compute an upper bound on the WCET – Implicit Path Enumeration Technique (IPET). If ILP could handle efficiently disjunctions then we could translate a conflict between the edges `A`, `B` and `C` by $n_{\mathtt{A}} = 0 \vee n_{\mathtt{B}} = 0 \vee n_{\mathtt{C}} = 0$ where $n_{\mathtt{X}}$ is the number of executions of edge `X`.

Our contributions are the following:

- We present an extension of the FFX format [2] for expressing conflicts. We can represent simple conflicts, conflicts that are valid in a given context and conflicts involving specific instances of an edge, with or without relevant order (Section 2).
- We propose two ways of integrating conflicts in the WCET analysis, through CFG transformation (based on [8], Section 3) or with additional ILP constraints (based on [9], Section 4).

In Section 5 we experiment these approaches on the Mälardalen benchmark suite [4] and two C programs generated from Esterel, considering conflicts inferred by a SMT-based tool at binary level [10]. Gains can be significant (but do not exceed 10%).

The issue of infeasible paths in WCET analysis being an active topic since many years, we compare our expression format and integration process with some previous work inferring, expressing and exploiting conflicts [3, 5, 11, 6] in Section 6.

2 Expressing Conflicts in FFX

The open format FFX (Flow Facts in XML) is a flowfact annotation language [2] aimed to be portable, expandable and easy to write, understand and process. Annotations in FFX are stored in an XML file rooted by a `flowfacts` element. Inside, a hierarchy of elements represents:

- Facts (e.g. loop bounds)
- Context of validity for inner elements (e.g. call context, loop iteration)

2.1 The `conflict` Element

We enrich the FFX format with the element `conflict` illustrated in Listing 2. This element means that a valid program trace cannot contain all blocks/edges mentioned inside. In other terms, only paths that go through at most N-1 of the N blocks/edges mentioned in the element are valid. A block/edge is identified by the `block/edge` FFX element and we make the assumption that it can be mapped to a block/edge of the CFG.

[1] The term is borrowed from [11].

```
<loop loopId="L">
  <iteration number="*">
    <conflict>
      <edge "A" />
      <edge "B" />
    </conflict>
  </iteration>
</loop>
```

Listing 3 Conflicts for each iteration of a loop.

```
<conflict>
  <edge "A" />
  <call name="C1" ...>
    <edge "B" />
    <edge "C" />
  </call>
</conflict>
```

Listing 4 Conflict with edges in a call.

```
<conflict ordered="yes">
  <edge "A" />
  <edge "B" />
  <edge "C" />
</conflict>
```

Listing 5 `A` then `B` then `C` is infeasible, but `CAB` is allowed for example.

2.1.1 Context of Validity

We inherit from FFX the notion of context of validity. A `conflict` element inside a given context applies to each sub-trace defined by the context. For example, Listing 3 describes a conflict that occurs in each iteration of loop `L`. It means that in one iteration we can see no `A` but some `B` and in another iteration no `B`, but some `A` (no `A` and no `B` is also allowed).

2.1.2 Specific Instances of Edge/Block

The use of contextual elements in FFX allows stating that a property holds in that context (e.g. for the last iteration, when the function is called from a given call site). We employ the same contextual elements *inside* the `conflict` tag to restrain it to specific instances of a given edge or block. For example, Listing 4 describes a conflict between `A` and instances of `B` and `C` belonging to a specific call.

2.1.3 Ordered Conflict

We experienced that a conflict as described in Section 2.1 is a strong property in the sense that it holds regardless of the order of its elements. To allow the expression of a weaker property, we introduced an attribute named `ordered` for the `conflict` element. This attribute can be given the value `yes` or `no`, the default being `no`. If a conflict is ordered, it only states that its constituents cannot appear altogether in that order.

2.2 Formal Semantics

In this section, we give a formal semantics to the `conflict` element. This semantics is the foundation of the properties of Section 2.3 but it can be skipped if reading Section 2.1 was clear enough. The semantics takes the form of a predicate indicating whether the execution path π is accepted or not. We *extend* the FFX acceptance semantics $\mathcal{FFX}\left[\!\left[\cdot\right]\!\right]$ as follows:

$$\mathcal{FFX}\left[\!\!\left[\begin{matrix}\texttt{<conflict ordered=}o\texttt{>}\\ elems\ \texttt{</conflict>}\end{matrix}\right]\!\!\right](\pi) = \mathcal{C}\left[\!\left[elems\right]\!\right](o,\pi)$$

$$\mathcal{C}\left[\!\left[elem_1 \dots elem_n\right]\!\right](\texttt{no},\pi) = \bigvee_{1\le i\le n} \mathcal{C}\left[\!\left[elem_i\right]\!\right](\texttt{no},\pi)$$

$$\mathcal{C}\left[\!\left[elem_1 \dots elem_n\right]\!\right](\texttt{yes},\pi) = \langle\sigma_1,\dots,\sigma_n\rangle \in \text{split}_n(\pi) \Rightarrow \bigvee_{1\le i\le n} \mathcal{C}\left[\!\left[elem_i\right]\!\right](\texttt{yes},\sigma_i)$$

$$\mathcal{C}\left[\!\left[\texttt{<edge }e\texttt{/>}\right]\!\right](o,\pi) = e \notin \pi$$

$$\mathcal{C}\left[\!\left[\texttt{<block }b\texttt{/>}\right]\!\right](o,\pi) = b \notin \pi$$

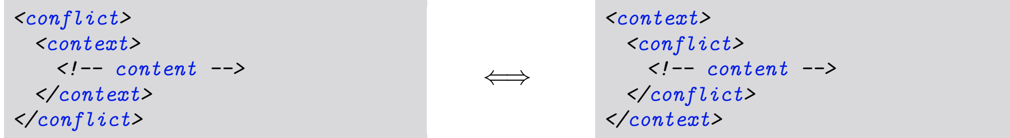
$$\mathcal{C}\left[\!\left[\texttt{<}ctx\texttt{>}elems\texttt{</}ctx\texttt{>}\right]\!\right](o,\pi) = \sigma \in \text{sub}_{ctx}(\pi) \Rightarrow \mathcal{C}\left[\!\left[elems\right]\!\right](o,\sigma)$$

where sub_{ctx} is a function returning the sub-traces matching the contextual element *ctx* and where $\sigma_1.\cdots.\sigma_n = \pi$ means that the σ_i form a sequential decomposition of π.

2.3 Properties of the `conflict` Element

▶ **Property 1.** *A* `conflict` *element can be flipped with:*

- *Its internal context, if it is the only child of the* `conflict`*,*
- *Its external context, if the* `conflict` *is its only child.*

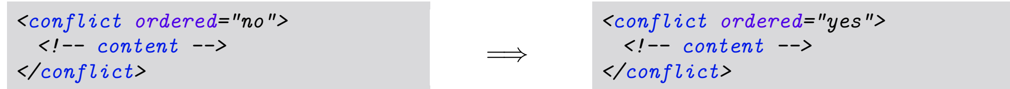


```
<conflict>
  <context>
    <!-- content -->
  </context>
</conflict>
```

$\Longleftrightarrow$

```
<context>
  <conflict>
    <!-- content -->
  </conflict>
</context>
```

▶ **Property 2.** *A conflict having its* `ordered` *attribute set to* `no` *entails the same conflict with this attribute set to* `yes` *since the paths that go through an ordered list of edges is a subset of the paths that go though the same list of edges in any order.*



```
<conflict ordered="no">
  <!-- content -->
</conflict>
```

$\Longrightarrow$

```
<conflict ordered="yes">
  <!-- content -->
</conflict>
```

3 Integrate Conflicts by CFG Transformation

In this section we propose to turn a `conflict` FFX tag into an equivalent automaton to integrate it in the analysis process through a CFG transformation. This idea is based on [8] where we present a method to turn the semantic information of an annotation language (such as FFX) into a hierarchical automaton enriched with constraints. We then perform a product operation between the CFG of a program and this automaton and feed the result to IPET to obtain a WCET.

3.1 Example of CFG Transformation

If we reduce a CFG to an equivalent Deterministic Finite Automaton (DFA), we obtain the set of paths that are structurally possible in a program. If we want to remove an edge of this DFA, we can simply perform an automata product with an automaton that would forbid that specific edge. More precisely, this automaton would accept any edge except this one.

The general idea is to rely on the formal operations that exist on automata to integrate additional semantic information in the analysis process.

Figure 1 illustrates an automata product between an automaton that represents a conflict between two edges A and B and the DFA that corresponds to a CFG.

The *Conflict* automaton works as follows: in the state `x`, all edges are accepted through the default transition *, except edges A and B that go respectively in `y` (resp. `z`) where any edges different from B (resp. A) is accepted.

The result of this product is an unfolded CFG where there exists no path that can go through A and B. The semantic information of the conflict has been carried by the automaton and integrated as a structural restriction in the CFG.

Two limitations appear in this process:

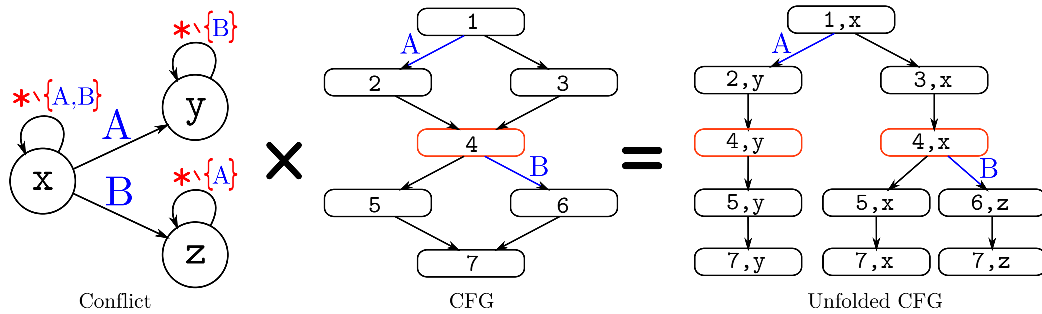


Figure 1 Product between a conflict automaton and a CFG.

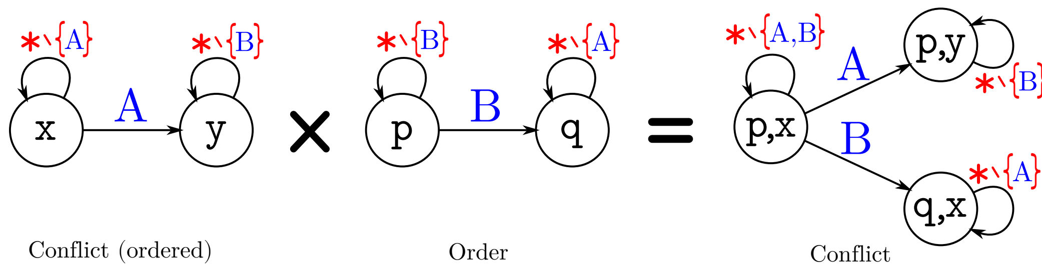


Figure 2 Product between an ordered conflict and the automaton that describes the order.

- Turning a semantic property into an equivalent automaton is not straightforward. However we presented in [8] a strong formalism of hierarchical automata enriched with constraints that can handle most of the annotations that are used in the WCET analysis (e.g. contexts, loop bounds, conflicts...).
- This approach suffers from scalability problems: in Figure 1, some blocks are duplicated due to the product (e.g. block 4,5,7). Also, blocks `7,x` and `7,y` are now separated due to the `z` state, even if there exists no path that could go through A after B. And if the CFG had more blocks after block `7`, they would all appear in the three branches. Moreover this is only a 2-edge conflict. A 3-edge conflict has seven states and a product could almost multiply the number of blocks of a CFG by seven. We propose to use the "ordered" property to reduce the number of states of the conflict automaton and then to reduce the number of replications.

3.2 Simpler Automaton with the Ordered Property

When the ordered property of a conflict is set to "yes", the only paths removed are those that go through the elements of a `conflict` in the right order. This is weaker than the non-ordered conflict. However, if we add the information that the order specified in the `conflict` is the only one possible, we obtain an equivalence to the non-ordered conflict.

The first automaton in Figure 2 represents an ordered conflict that only excludes paths where B is taken after a A.

If we perform a product with the second automaton which ensures that no A can occur after a B, we obtain the previous non-ordered conflict.

It turns out that when our conflict detection tool finds a conflict over a list of edges of a CFG in a specific context, it also ensures that there exists no other possible orders for these edges in this context. In other terms, the order of these edges is already a structural constraint of the CFG. In these conditions, we can carry the property with the weaker but simpler ordered conflict automaton, since it is the only structurally feasible path in the CFG.

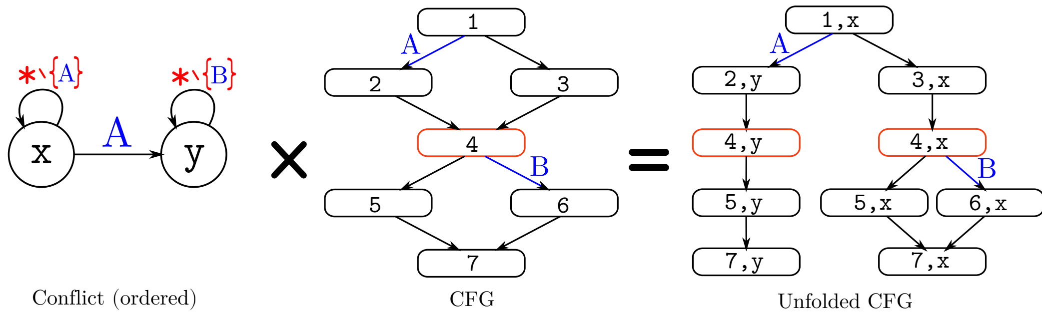


Figure 3 Product between an ordered conflict and a CFG.

3.3 Benefits of the Ordered Property

Figure 3 illustrates how a simpler conflict automaton avoids the separation of the CFG in three distinct branches while still removing the conflicting path from the structure.

With this ordered version of the conflict, we can consider performing the product using a conflict over n edges, even when n is greater than three since it results in an automaton with $n-1$ states. It was impossible with the non-ordered conflict which results in an automaton with 2^n-1 states and almost as many potential replications of each block of the CFG.

4 Integrate Conflicts with ILP Constraints

In this section, we present a method based on [9] for translating FFX conflicts into ILP constraints. These constraints are meant to be added to the constraints produced by an IPET-based WCET analysis tool (with potential pessimism).

4.1 Framework

In [9] Raymond proposed a method for turning any set of conflicting edges into linear constraints. For example, consider a CFG where an edge $\mathtt{A}$ is located inside a loop (bound n) and an edge $\mathtt{B}$ outside. Consider a conflict between the edge $\mathtt{A}$ (in any iteration) and the edge $\mathtt{B}$. If we unroll the loop, the complete set of conflicts is $\{(\mathtt{A}_1,\mathtt{B}),\dots,(\mathtt{A}_n,\mathtt{B})\}$. Note that $\mathtt{A}_1,\dots,\mathtt{A}_n$ are called *avatars* of $\mathtt{A}$.

The method proposed turns this set of conflicts into the ILP constraint $\mathtt{A}+n.\mathtt{B}\leq n$.

The key idea of this framework is that it works on an acyclic unfolding of the original CFG where we can turn a conflict like $(\mathtt{A}_1,\mathtt{B})$ into $\mathtt{A}_1+\mathtt{B}\leq 1$. From sets of such inequalities, clever summation recovers linear constraints on the original CFG. This leads to a general formula that is valid for any set of CFG edges X and any set of conflicting edge avatars S built upon X:

$$\sum_{x\in X} p_x x \leq (|X|-1)|S| + \sum_{x\in X} l_x$$

where:
- p_x is the maximum of the counts of each avatar of x in S (called multiplicity),
- l_x is $p_x m_x - |S|$ (called lack),
- m_x is the number of avatars of x in the unfolding.

This formula does not reflect exactly the conflict, but it is a safe approximation.

```
<conflict ordered="yes">
  <edge "a" />
  <edge "b" />
</conflict>
```

Listing 6 Conflict without context.

```
<conflict ordered="yes">
  <loop address="0x...">
    <iteration number="*">
      <edge "a" />
      <edge "b" />
    </iteration>
  </loop>
</conflict>
```

Listing 7 Conflict on each iteration of a loop.

```
<conflict ordered="yes">
  <loop address="0x...">
    <iteration number="-1">
      <edge "a" />
      <edge "b" />
    </iteration>
  </loop>
  <edge "c" />
</conflict>
```

Listing 8 Conflict on the last iteration of a loop and after.

4.2 Translations

Our contribution is a prototype OTAWA [1] plug-in to translate FFX conflicts into ILP constraints. We illustrate the translation performed on several examples.

The conflict presented in Listing 6 is a simple conflict with no contexts. Under the hypothesis that $m_a = m_b = 1$ (a and b appear only once in the acyclic unfolding) the method results in: $S = \{(a, b)\}$ and $|S| = 1, p_a = p_b = 1, l_a = l_b = 0$. Therefore the generated constraint is $a + b \leq 1$.

The Listing 7 presents a conflict for each iteration of a loop. Under the assumption that $m_a = m_b = n$, we obtain the system:

$$\begin{cases} S = \{(a_1, b_1), \ldots, (a_n, b_n)\} \\ |S| = n, p_a = p_b = 1, l_a = l_b = 0 \end{cases}$$

The resulting ILP constraint is then $a + b \leq n$. Note that the right-side of the inequality can be generalized to $n(|X| - 1)$.

Finally, the conflict presented in Listing 8 illustrates a conflict between two edges in the last iteration of a loop and a third one after. Under the assumption that $m_a = m_b = n \land m_c = 1$, the following system is obtained:

$$\begin{cases} S = \{(a_n, b_n, c)\} \\ |S| = 1, p_a = p_b = p_c = 1, l_a = l_b = n - 1, l_c = 0 \end{cases}$$

The generated ILP constraint is then $a + b + c \leq 2n$. Again, note that the right-side of the inequality can be generalized to $|X| + |In|.(n - 1) - 1$ where $In \subseteq X$ is the subset of edges belonging the loop.

5 Experiments

The PathFinder tool presented in [10] runs an abstract interpretation top-to-bottom analysis on binary programs, looking for semantic conflicts. It acts as a pre-analysis for the WCET analysis, aiming to provide information about infeasible paths in the most factorized, exploitable way possible for other analyses. It relies on the OTAWA framework.

This infeasible path analysis tool models the program state for a set of paths in the CFG as a conjunction of predicates on registers and memory cells. It checks the feasibility of a set of paths by checking the satisfiability of this abstract program state in an SMT[2] solver.

When an infeasible path is detected, PathFinder attempts to express the detected conflicts in a *minimal* number of infeasible paths, each written as a list of CFG edges that cannot all

[2] Satisfiability Modulo Theory

Table 1 Results on the Mälardalen benchmarks.

	Nb. of conflicts found		WCET gain simple arch.		WCET gain arm9 + cache	
Program	Total	After minim.	Constraints	Unfolded	Constraints	Unfolded
SMALL MÄLARDALEN BENCHMARKS						
adpcm	174	28	0.00 %	0.00 %	CE	CE
cnt	118	5	0.00 %	0.00 %	0.00 %	0.00 %
cover	3	3	6.95 %	6.95 %	**0.01 %**	**0.25 %**
crc	8	8	0.50 %	0.50 %	**4.10 %**	**9.70 %**
edn	7	6	0.03 %	0.03 %	CE	CE
expint	8	5	0.00 %	0.00 %	**0.00 %**	**0.09 %**
fibcall	1	1	0.72 %	0.72 %	0.32 %	0.32 %
fir	1	1	0.00 %	0.00 %	**3.37 %**	**7.45 %**
select	18	11	0.16 %	0.16 %	0.09 %	0.09 %
sqrt	407	10	0.40 %	0.40 %	0.04 %	0.04 %
LARGE MÄLARDALEN BENCHMARKS						
statemate	1118	71	2.77 %	CE*	1.00 %	CE*
ud	13	1	1.17 %	1.17 %	1.08 %	1.08 %
nsichneu	13648	7684	0.00 %	CE*	0.00 %	CE*
minver	10	9	1.40 %	1.40 %	CE	CE
ludcmp	29	3	0.00 %	0.00 %	0.00 %	0.00 %
lms	2097	141	CE	CE	CE	CE
fft1	830	149	CE	CE	CE	CE
qurt	797	41	CE	CE	CE	CE
ESTEREL BENCHMARKS						
runner	5618	185	9.84 %	CE*	9.12 %	CE*
abcd	4949	274	3.01 %	CE*	5.17 %	CE*

be taken on a single path associated with a context (the scope of a loop, of a function for any or a particular function call point).

This list of edges expressing each infeasible path must also be as small as possible, in order to minimize the complexity of WCET analysis that will use these results. This is achieved by retrieving from the SMT solver a kernel of predicates, named an unsatisfiable core, and hook one or several edges to each predicate. Other refinements are performed, namely using (post-)dominance properties on the CFG in order to remove superfluous items from the list of conflicting edges.

Once the analysis completes and the infeasible paths have been minimized as best we can, we output them using the FFX format (Section 2).

Results of our experiments are presented in Table 1. Programs were compiled for the *armv5t* architecture without any optimization (-O0). The second and third columns show the total number of conflicts found by Pathfinder, and the number after minimization. In the next two columns, we analyzed the benchmarks with a trivial architecture that applies a simple metric to compute low-level timings and without any cache. In the last two columns, we used an architecture model derived from ARM9 with a 1 KB instruction cache and a 256 KB data cache.

For several programs we were not able to compute a WCET due to *Capacity Exceeded* (CE) because of scalability problems occurring either during the unfolding of the CFG (CE*), or during the WCET computation itself. Other benchmarks of the suite are not listed here either because they contain recursion that is not supported by Pathfinder or because Pathfinder did not detect any conflict.

In the *Constraints* columns we fed ILP with the constraints derived from the conflicts (Section 4) while we applied the automata product (section 3) in the *Unfolded* columns. The values are the percentage of gain between the WCET computed with the integration of conflicts over the WCET estimation without.

Thanks to the integration of conflicts in the WCET analysis, we observed significant improvements of the precision in some cases, even for a trivial architecture without cache: for these benchmarks, the original WCEP was infeasible.

Unsurprisingly, the table also shows that for a trivial architecture without cache, the WCET precision improvement is exactly the same if we add ILP constraints or if we unfold the CFG. In other terms, gains only appear when the WCEP is infeasible.

On the other hand we highlighted an important result in the column where cache is used: significant precision improvements are noticeable (in bold) when we unfold the CFG. Indeed, at some points in the WCET analysis, abstract cache states that represent the possible states of a cache are merged. When the constraint method is used, the merge points remain unchanged, but unfolding the CFG allows to separate paths which avoids some of these merging operations. It results in more precise abstract cache states and eventually a more precise WCET estimation.

6 Related Work

Engblom et al. [3] present an IPET-based framework for WCET analysis together with an annotation language based on scopes. These scopes correspond to the contexts enclosing our conflicts. We rely on their technique for scoping out local ILP constraints to global ones.

Kirner et al. [6] survey the annotation languages used by the WCET analysis tools (including FFX). They identify categories of dynamic control flow information but the notion of conflict is absent. In their section 9, they use four languages to encode flow information, one of which ($\mathbf{B}_1$) is a contextual conflict. Only PL and IDL are able to encode faithfully the conflict using a regular expression (linked with Section 3).

Suhendra et al. [11] present an algorithm[3] for inferring pairwise-conflicts in a given context (function or iteration). They then use these conflicts to improve a path-based WCET analysis. There tool was not meant to export this knowledge.

Knoop et al. [7] refine the WCET by disproving the feasibility of the WCEP. This approach has the clear interest of focusing the flow analysis on relevant paths. The price to pay is an increased WCET resolution technique complexity. Note that ignoring the infeasibility of paths that are not WCEP might lead to indirect over-approximation of the WCET (Item 2 in Section 1).

Ruiz and Cassé [10] retrieve conflicts from a binary program. Unlike the work previously mentioned these conflicts can involve edges in a loop and edges outside of this loop. The analysis can deliver contextual conflicts over a single function call. It was a motivation for extending the FFX format so to encompass all these subtleties.

[3] It turns out that Algorithm 1 of [11] is wrong. If the program of Figure 1 is run with $\mathtt{x} = \mathtt{y} = \mathtt{z} = 5$ the path taken goes through edges that are reported as branch-branch conflict. This is due to a bad hypothesis in the second item of Definition 3.2: *all* paths between the two branches must not modify the involved variables.

7 Conclusion

Throughout the paper, we presented an extension to the FFX format that allows representing a class of infeasible paths called conflicts (Section 2). Finding these conflicts and taking them into account enables an improvement of the WCET analysis precision of up to 10% on some programs (Section 5). We presented two methods to integrate the conflicts in the WCET analysis. One creates additional ILP constraints reflecting the conflicts (Section 4) and scales well. The other one is an automatic transformation of the CFG that removes the conflicting paths (Section 3); the size of the CFG may explode, but this method can affect the low-level analyses and yield additional WCET precision.

We noted that benchmark suite programs offer a field of improvement through detection and integration of conflicts, while source codes generated from Esterel compiler are slightly more promising. We plan to look for other classes of programs to illustrate the various conflicts that our tools are able to detect and take into account. We also plan to address the issue of scalability, both in the WCET analysis and with the unfolding method, in order to support bigger and more complex programs.

References

1. Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: an open toolbox for adaptive WCET analysis. In *8th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS 2010)*, 2010. doi:10.1007/978-3-642-16256-5_6.
2. Armelle Bonenfant, Hugues Cassé, Marianne De Michiel, Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. FFX: a portable WCET annotation language. In *20th International Conference on Real-Time and Network Systems (RTNS 2012)*, 2012. doi:10.1145/2392987.2392999.
3. Jakob Engblom and Andreas Ermedahl. Modeling complex flows for worst-case execution time analysis. In *21st IEEE Real-Time Systems Symposium (RTSS 2000)*, 2000. doi:10.1109/REAL.2000.896006.
4. Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The mälardalen WCET benchmarks: Past, present and future. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, 2010. doi:10.4230/OASIcs.WCET.2010.136.
5. Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *27th IEEE Real-Time Systems Symposium (RTSS 2006)*, 2006. doi:10.1109/RTSS.2006.12.
6. Raimund Kirner, Jens Knoop, Adrian Prantl, Markus Schordan, and Albrecht Kadlec. Beyond loop bounds: comparing annotation languages for worst-case execution time analysis. *Software and System Modeling*, 10(3), 2011. doi:10.1007/s10270-010-0161-0.
7. Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. WCET squeezing: on-demand feasibility refinement for proven precise wcet-bounds. In *21st International Conference on Real-Time Networks and Systems (RTNS 2013)*, 2013. doi:10.1145/2516821.2516847.
8. Vincent Mussot and Pascal Sotin. Improving WCET analysis precision through automata product. In *21st IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2015)*, 2015. doi:10.1109/RTCSA.2015.11.
9. Pascal Raymond. A general approach for expressing infeasibility in implicit path enumeration technique. In *International Conference on Embedded Software (EMSOFT 2014)*, 2014. doi:10.1145/2656045.2656046.

10 Jordy Ruiz and Hugues Cassé. Using SMT solving for the lookup of infeasible paths in binary programs. In *15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015)*, 2015. `doi:10.4230/OASIcs.WCET.2015.95`.

11 Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. Efficient detection and exploitation of infeasible paths for software timing analysis. In *43rd Design Automation Conference (DAC 2006)*, 2006. `doi:10.1145/1146909.1147002`.