WCET Squeezing: On-demand Feasibility Refinement for Proven Precise WCET-bounds*

Jens Knoop
TU Vienna
Vienna, Austria
knoop@complang.tuwien.ac.at

Laura Kovács
Chalmers
Gothenburg, Sweden
laura.kovacs@chalmers.se

Jakob Zwirchmayr
TU Vienna
Vienna, Austria
jakob@complang.tuwien.ac.at

ABSTRACT

The Worst-Case Execution Time (WCET) computed by a WCET analyzer is usually not tight, leaving a gap between the actual and the computed WCET of a program. In this article we present a novel on-demand WCET feasibility refinement technique, called *WCET Squeezing*, for minimizing this gap.

WCET Squeezing provides conceptually new means for addressing the classical problem of WCET computation, by deriving a WCET bound that comes as close as possible to the actual one. WCET Squeezing is an anytime algorithm, that is, it can be stopped at any time without violating the soundness of its results. This anytime property allows to apply WCET Squeezing not only for deriving precise WCET bounds but to also prove additional timing constraints over the program. Namely, WCET Squeezing can be used to guarantee that a program is fast enough by ensuring that the WCET of the program is below some required limit. If the initially computed WCET of the program is above this limit, WCET Squeezing can be stopped as soon as the squeezed WCET of the program is below the limit (proving the program meets the required timing constraint), or if the squeezed WCET is tight but above the given limit (proving the program cannot meet the timing constraint). WCET Squeezing can also be used until a given time budget is exhausted to compute a tight(er) WCET bound for a program. These new applications of WCET Squeezing are out of the scope of traditional WCET analyzers.

WCET Squeezing combines symbolic program execution with the Implicit Path Enumeration Technique (IPET) for computing a precise WCET bound. WCET Squeezing is applicable as a post-process to any WCET analyzer which encodes the IPET problem as an Integer Linear Program (ILP). We implemented our method in the r-TuBound toolchain and evaluated our implementation on a set examples taken from the Mälardalen WCET benchmark suite. Our experiments demonstrate that WCET Squeezing can significantly tighten the WCET bounds of programs. Moreover, the derived WCET bounds are proven to be precise at a moderate computational cost.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
RTNS 2013 , October 16 - 18 2013, Sophia Antipolis, France
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
Copyright 2013 ACM 978-1-4503-2058-0/13/10 ...$15.00.
http://dx.doi.org/10.1145/2516821.2516847.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time systems and embedded systems

1. MOTIVATION

In order to ensure soundness of safety-critical real-time systems it is crucial to verify their temporal properties. Such systems are composed of tasks which must finish within a given time period. A task scheduling analysis is therefore performed to guarantee safety of the system. Task scheduling requires an a-priori known Worst-Case Execution Time (WCET) for each of the involved tasks, which is usually obtained by an architecture-dependent timing analysis and an architecture-independent flow fact analysis.

In this article we present a new method, called *WCET Squeezing*, for tightening the WCET provided by some off-the-shelve WCET analyzer. WCET Squeezing applies on-demand WCET feasibility refinement. It takes as input the result of an a-priori WCET analysis of the program and tightens, that is *squeezes*, this WCET estimate, by refining the program model. To this end, WCET Squeezing applies a new kind of symbolic execution, called *selective symbolic execution* [6, 5], in combination with the Implicit Path Enumeration Technique (IPET) of [26]. To squeeze the computed WCET bound for a program, we map the result of the IPET analysis to a trace in the program and symbolically execute this trace to decide whether it is feasible or not. If it is feasible, the computed WCET bound is tight and WCET Squeezing terminates by reporting a precise WCET bound: any remaining WCET overestimation is due to a conservative hardware model. If it is infeasible, the original IPET problem, encoded as an Integer Linear Program (ILP), is extended by a new constraint excluding the infeasible trace. The new IPET problem is then solved, resulting in a tighter WCET bound. The new ILP problem yields a new trace that is used in the next iteration of WCET Squeezing. By iteratively applying these steps until termination, WCET Squeezing refines the initial WCET bound derived by a static WCET analyzer. Moreover, upon termination of WCET Squeezing, a precise WCET bound is obtained. Precision of a WCET bound guarantees that the bound is computed for an actual execution trace of the program.

This brief description of WCET Squeezing has many similarities with the Counter-example Guided Abstraction Refinement (CeGAR) approach of [12]. In the CeGAR method, an initial abstrac-

*This research is partly supported by the FP7-ICT Project 288008 Time-predictable Multi-Core Architecture for Embedded Systems (T-CREST), the EU FP7 Cost Action no. IC1202 Timing Analysis on Code-Level (TACLe), the FWF National Research Network RiSE (S11410-N23), the FWF Hertha Firnberg Research grant (T425-N23), the WWTF PROSEED grant (ICT C-050), and the CeTAT project of the TU Vienna.

tion of the program is analyzed for reachability of error states. If an error state is spurious, that is reachable in the abstraction but not in the program, the abstraction is refined to exclude reachability of the error state. The refined abstraction is used in the next iteration of CeGAR. Similarly, in WCET Squeezing, we start with an initial abstraction of the program, where the abstraction is encoded as an ILP problem. A solution of the ILP problem represents a WCET execution trace of the program. This trace might however only be feasible in the considered abstraction and not in the concrete program. Therefore, we next symbolically execute the trace and, if found infeasible, the abstraction is refined to exclude the trace. This refined abstraction is used in the next iteration of WCET Squeezing. Note that the used program abstraction gets more precise in each iteration of our WCET Squeezing algorithm, yielding a more precise WCET estimate for the program. Let us however emphasize that WCET Squeezing avoids the short-comings of IPET and symbolic execution, namely the lack of program knowledge beyond flow facts for IPET and the non-scalability of symbolic execution for a fast growing number of paths. This is illustrated in Figure 1: infeasible executions with a high WCET (i.e. the crossed long-waved arrows) are ruled out by symbolic execution until the estimate is below a required threshold (i.e. the short-waved arrows) or the time budget is exhausted or until a feasible execution is found.

As solving the *classical WCET problem* means to minimize the gap between the actual and the estimated WCET bound of a program, we therefore conclude that WCET Squeezing solves the classical WCET problem. However, WCET Squeezing can be also used to prove or disprove the so-called *pragmatic WCET problem*, where we define the pragmatic WCET problem as the problem of proving that a program meets some given timing constraint. When using WCET Squeezing for solving the pragmatic WCET problem, we summarize the following three applications.

1. *Precision-controlled WCET Squeezing* runs until a feasible trace is found. Since the feasible trace exhibits the actual WCET of the program, in this mode WCET Squeezing proves the precision of the derived WCET bound.

2. *Budget-controlled WCET Squeezing* tightens the WCET bounds until an a-priori specified running time budget is exhausted. While the WCET bound computed in this case is derived to be tight within the given running time, the program trace exhibiting the WCET bound might still be proven infeasible when a bigger running time budget is used.

3. *Limit-controlled WCET Squeezing* aims at proving that the program is fast enough, that is, its WCET below is below some a-priori given limit. In this case, one of the two scenarios might happen. If WCET Squeezing computes a WCET bound below the specified limit, it proves that the program is fast enough. Otherwise, WCET Squeezing derives a precise WCET bound which is greater than the required; in this case, WCET Squeezing proves that the program fails to meet its required timing constraint.

We implemented our WCET Squeezing algorithm in the r-TuBound tool chain [22], by integrating and adjusting the symbolic execution engine of [5] in r-TuBound. When evaluated our implementation on examples coming from the Mälardalen benchmark suite [17], our experimental results show that WCET Squeezing is both effective and efficient. For instance, it is not unusual that the precision of WCET bounds were improved by up to 9% after only two iterations of WCET Squeezing; in some cases these bounds were also proven precise after two WCET Squeezing iterations.

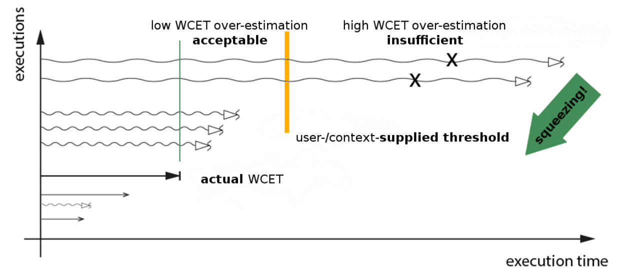

Figure 1: Straight arrows represent concrete executions, assuming some hardware-model. Long waved arrows are infeasible executions with over-estimated WCET.

By running WCET Squeezing until termination, we observed that for some examples the precise WCET bounds improved the initial WCET bound by up to 90%.

Our WCET Squeezing approach is, by design, out-of-scope of any current WCET analyzer. The proof of precision of WCET bounds makes WCET Squeezing unique and empowers the strength of other WCET tools as well. WCET Squeezing can nicely be integrated with other WCET methods, where tighter initial WCET bounds yield better performances of WCET Squeezing. If a static WCET analyzer supplies tight initial bounds, a proof of precision can be obtained fast and hence the computational effort for WCET Squeezing is reduced. WCET Squeezing can therefore be used as an additional tool to improve the quality of WCET analyzers: instead of replacing current WCET methods, WCET Squeezing can be used in conjunction with other approaches to minimized the over-estimation of WCET bounds.

Contributions. The main contributions of this paper are as follows.

- We present the first method to *automatically tighten and prove precise* the WCET bound computed by a state-of-the-art WCET analyzer, called *WCET Squeezing*. To this end, it combines in a unique novel fashion traditional IPET-based WCET analysis techniques with (selective) symbolic execution.

- WCET Squeezing works on-demand and can be used in three different modes that are all out of the scope of traditional WCET analyzers. (1) Precision-controlled: Computing a WCET that comes as close as possible to or even coincides with the actual WCET of a program (classical WCET problem). (2) Limit-controlled: Squeezing the WCET as much as necessary in order to prove or disprove that a program meets a pre-defined required WCET limit (pragmatic WCET problem). (3) Budget-controlled: Tightening a WCET estimate within a pre-defined time budget allowed for WCET Squeezing (pragmatic WCET problem).

- WCET Squeezing is the first technique that systematically addresses limit-controlled and budget-controlled WCET analysis. We consider these two variants instances of a new WCET problem that we denote as the pragmatic WCET problem in distinction to the classical WCET problem. It is worth noting that WCET Squeezing addresses the classical WCET problem in a fashion that is out of the scope of traditional WCET analyzers, namely by tightening and proving precise some initially computed WCET bound.

- The WCET refinement constraints are fully automatically derived by WCET Squeezing, without using or relying on

a-priori specified WCET templates and predicates. The approach is implemented as an extension of the r-TuBound toolchain and evaluated on a set of the Mälardalen benchmarks suite [17]. These experiments indicate that WCET Squeezing is most effective in practice at moderate costs.

The rest of the paper is structures as follows. Section 2 illustrates our approach on an example. Section 3 overviews relevant terminology from WCET analysis and symbolic execution. Section 4 presents WCET Squeezing and Section 5 summarizes our experimental findings. Section 6 discusses related work and Section 7 concludes.

2. OVERVIEW OF THE APPROACH

We illustrate WCET Squeezing on the example given in Figure 2, taken from the `lcdnum.c` benchmark of the Mälardalen WCET suite [17].

This example consists of a for-loop with a conditional statement calling the function `num_to_lcd`. An initial WCET analysis of this program infers a loop bound of 10, and yields a WCET estimate of 24320 cycles with the execution frequency of 10 for the `then`-branch (`true`-block) of the conditional inside the loop. By mapping back this WCET result to a WCET program trace candidate, we only obtain the program execution trace calling `num_to_lcd` in each iteration. WCET Squeezing uses this initial WCET trace candidate as a starting point for the first symbolic execution and concludes the infeasibility of this trace. In the next step of WCET Squeezing, an IPET constraint excluding this infeasible trace is derived and added to the IPET-based WCET analysis. This yields a new WCET estimate of 23420 cycles with an execution frequency of 9 for the `true`-block in the loop, which constitutes a tightening of 3.7%.

```
    ...
for(i = 0; i < 10; i++) {
  if(i < 5) {
    a = a & 0x0F;
    OUT = num_to_lcd(a);
  }
}
    ...
```

Figure 2: `lcdnum.c`, simplified.

In the second iteration of WCET Squeezing, this new WCET estimate is used and feasibility of the WCET trace candidates is checked again. This time, there are multiple WCET trace candidates since the `else`-branch (`false`-block) of the conditional can be also taken in a loop iteration. By using symbolic execution, none of these WCET trace candidates are found feasible. A third iteration of WCET Squeezing is therefore taken, ruling out all WCET trace candidates with a `true`-block frequency of 8. This way, the WCET estimate of the program is tightened to 22520 cycles. Note that, compared to the initial WCET estimate, an accumulated WCET improvement of 7.4% is obtained, after excluding 11 WCET trace candidates.

The number of WCET trace candidates that require symbolic execution increases in the following iterations until WCET Squeezing terminates in the sixth iteration, when it finds a feasible WCET trace candidate with execution frequency of 5 for the `true`-block. This feasible trace results in a WCET estimate of 19820 cycles. Compared to the original WCET estimate this is a tightening of 18.5% and the new bound is proven precise.

3. PRELIMINARIES

Our work combines techniques from symbolic execution, WCET analysis and program verification in a unique fashion. In this section we present the necessary ingredients of our method.

Programming model. As usual, we represent programs as control flow graphs (CFG), $CFG := ((V, E), S, X)$, where V is the set of nodes, E is the set of directed edges representing program blocks, $S \in V$ is the start-node, and $X \in V$ is the end-node [25]. For each edge $e \in E$ an edge weight $w(e)$ is assigned denoting the execution time of e, and we hence have $w : E \to \mathbb{N}$. To ease readability, we will omit edge weights wherever possible. Every node n, different than S and X, has incoming $inc(n)$ and outgoing edges $out(n)$; the node S has only outgoing edges $out(S)$ and no incoming ones; whereas the node X has only incoming edges $inc(X)$ but no outgoing ones. Conditional nodes C split the flow depending on the runtime evaluation of a boolean condition $c(C)$, where we refer to $c(C)$ as the path-condition. For simplicity, we sometimes write C instead of $c(C)$. Execution times for $c(C)$ are assumed to be added to the successor edge weights. Edges taken when the condition C evaluates to `true` are called `true-edges` (`true-blocks`) and are denoted by t. Similarly, edges taken when the condition evaluates to `false` are called `false-edges` (`false-blocks`) and are denoted by f. To make explicit that t and f result from the evaluation of C, we write t_C and f_C to mean that these are the true-, respectively `false`-edges of C. Further, t_C and f_C are called the *conditional-edges* of C.

A path in the CFG is a sequence of nodes and edges and a program execution trace is a path from S to X. A path condition in a trace is the evaluation of a condition at a branching point that forces execution along the trace. If a branching point is guarded by a condition `c`, then the path condition for the trace that follows the `true`-branch of the conditional is *eval*(`c`) = `true`. The evaluation of all path conditions in an execution trace defines the *branching behavior* of the trace, i.e. the evaluation of all conditions along the trace. It is thus a sequence of branching decisions that can be encoded as a sequence of bits, where each bit b_i represents the result of evaluating the ith branch condition in the trace (1 if the condition holds, when executing the `true`-edge, 0, when executing the `false`-edge). A program loop in the CFG is modeled by a loop header *lh*, a loop condition *lc* and body *lb* and a loop exit *le* node, with an edge from *lb* to *lc*. Each loop is annotated in the CFG with a loop bound ℓ. A valid path including a loop therefore contains *lb* at most $lh * \ell$ times, each time *lh* is contained. The number of times an edge e is taken in a path is given by its frequency $freq(e)$, where $freq : E \to \mathbb{N}$ and $freq(e)$ gives the sum of executions of the edge e in an execution trace.

Implicit Path Enumeration Technique - IPET. The IPET method of [26] first translates the CFG of a program into an ILP problem. Next, it computes an ILP solution corresponding to the path with the highest edge-weight. For doing so, the following constraints on the CFG are used: (i) the program is entered and exited once, that is $\sum out(S) = \sum inc(X) = 1$; (ii) the execution frequency of incoming edges is equal to the execution frequency of outgoing edges, that is $\sum in(n) = \sum out(n)$; (iii) for each loop, the loop body is executed ℓ times the loop header, that is $\sum out(lb) = \ell * \sum in(lh)$. The maximum solution to the above system of ILP constraints corresponds to the WCET estimate for the program. In the following, for simplicity, we will omit edge-weights when listing ILP problems. Note that the ILP solution fixes the execution frequencies of program blocks, resulting in an *ILP branching behavior*, induced by the ILP, that encodes one or more execution traces in the CFG. The execution traces resulting from the ILP branching behavior are called *WCET candidates* and if they are feasible, they exhibit the calculated WCET.

A single ILP branching behavior can result in one or more execution traces in the CFG, as information about the exact sequence of edge executions for edges in loops is not available. Note that without additional constraints, IPET will always select the maximum execution frequency for those edges of conditional nodes with higher edge-weight. Thus, the solution of IPET in absence

of additional constraints always encodes a single execution trace.

Consider now an execution path containing a loop with a conditional in the loop body, such that the loop is executed ℓ times. Assume that the frequency of the false-edge f is m, for some $m \in \mathbb{N}$. Therefore, the frequency of the corresponding `true`-edge t is constrained to $\ell - m$. The ILP branching behavior then encodes multiple execution paths. For conditional-edges $e, e' \in \{t, f\}$, we write ee' to mean that the execution of edge e is followed by the execution of e'. Then, the set of branching behaviors for $m = 1$ (omitting branching decisions outside the loop body) is $\{(f^1 t^2 t^3 \dots t^{\ell-1}), (t^1 f^1 t^2 \dots t^{\ell-1}), \dots, (t^1 t^2 t^3 ... t^{\ell-1} f^1)\}$, where t^i (respectively, f^i) denotes that the `true`-edge t (respectively, `false`-edge f) was taken in the ith iteration of the loop. Note that paths given above are valid execution paths in the CFG. They are however not necessarily valid execution paths in the original program, where each condition is evaluated at runtime. In the sequel, we will refer to the branching behavior describing a single execution trace as a path-expression. A single element in a path-expression is a branching decision for a conditional-node C.

Symbolic Execution. A symbolic execution engine, e.g. [5], models program executions by using symbolic values instead of concrete values. This allows one to execute a program on symbolic data instead of concrete values. Executing a path condition constrains the set of concrete values for a symbolic value. Therefore, runtime evaluation of conditions can be simulated on the symbolic representation. The same notions of path, execution trace, branching behavior and path expression defined for CFGs also apply for path-wise symbolic execution: a sequence of CFG branching decisions at the same time encodes a symbolic execution trace. Note, that a symbolic execution trace usually encodes multiple concrete execution traces.

4. WCET SQUEEZING

The WCET Squeezing algorithm, presented in Algorithm 1, iteratively refines the WCET estimate of programs with reducible control flow.

It takes as input the result of an a-priori WCET analysis of the program. That is, Algorithm 1 takes as input the ILP problem `ilp_problem` resulting from applying IPET on the CFG of the program under study. Remember that a maximum solution of the `ilp_problem` gives an (initial) WCET estimate of the problem. In addition to the ILP problem, an optional parameter can be supplied to guarantee termination within a certain time-limit `time`: if during that time a WCET candidate is excluded, the WCET estimate is improved, unless the next candidate exhibits the same estimate (*Cost-controlled*). Alternatively, when running WCET Squeezing with a pre-defined *threshold*-value, our method allows to solve the pragmatic WCET problem, that is to *answer whether a pre-defined required WCET limit can be met by a program*: WCET Squeezing is run until the improvement in the WCET estimate reaches the required value, reporting *yes*, or until it terminates due to a feasible candidate, reporting *no* (*Threshold-controlled*).

Note that WCET Squeezing is guaranteed to terminate, even without using a pre-defined budget- and/or limit. This is so because the ILP problems during WCET Squeezing encode only a finite number of WCET trace candidates. In the worst-case scenario, WCET Squeezing terminates after symbolically executing all program traces. It is worth noting that the WCET estimate is improved at every iteration of Algorithm 1 (with the exception when different paths exhibit a similar WCET bound), and hence the WCET estimate reported upon the termination of Algorithm 1 will be improved.

ALGORITHM 1. ***WCET Squeezing Algorithm***
Input: *ILP problem* `ilp_problem`
Output: *ILP solution* `ilp_solution`
Optional Input Parameter: *budget or limit* `BL_value`

```
1  begin
2  do
3    ilp_solution := ILPsolve(ilp_problem)
4    wcet_candidates := extractCandidates(ilp_problem,
                                          ilp_solution)
5    counter_ex := symbolicExecution(wcet_candidates)
6    if no counter_ex then return ilp_solution
7    ilp_problem := encodeConstraint(ilp_problem,
                                     counter_ex)
8  forever or [optional] until BL_value is reached
9  return ilp_solution
10 end
```

The main steps of Algorithm 1 are as follows. First, a solution `ilp_solution` of the ILP problem `ilp_problem` is computed (line 3), by using an off-the-shelve ILP solver [4]. Based on the computed ILP solution, the corresponding ILP branching behavior is mapped back to the CFG of the program and WCET trace candidates are extracted (line 4), as detailed in Section 4.1. These WCET trace candidates are next symbolically executed (line 5), as presented in Section 4.2. The result of symbolic execution on WCET trace candidates is stored in `counter_ex`: if a candidate is feasible, the `ilp_solution` corresponding to this trace is returned (line 6) as WCET estimate of the program under study. If all WCET trace candidates are infeasible, the ILP branching behavior is infeasible as well. The WCET estimate corresponding to the *ilp_solution* is exhibited by the program and the infeasible ILP branching behavior is excluded by adding a constraint to the ILP problem (line 7), as discussed in Section 4.3. A next iteration of WCET Squeezing is further applied on the new ILP problem, yielding tighter WCET estimates of the program (line 3). Algorithm 1 for WCET Squeezing terminates when a feasible and refined WCET estimate is derived (line 6).

Next we consider the ingredients of Algorithm 1 in more detail and describe the approach to extract WCET trace candidates (Section 4.1), symbolic execution (Section 4.2), and encoding of ILP constraints (Section 4.3).

4.1 WCET Trace Candidates

To construct WCET trace candidates, a mapping from the ILP branching behavior to program execution traces is needed. WCET trace candidates can be specified by a branching behavior, i.e. a sequence of branching decisions. The ILP branching behavior, denoted by *ilp_bb*, is initially generated by mapping all executed edges, that is edges with an execution frequency greater than 0, from the first ILP solution to a trace in the CFG of the program and selecting all conditions executed in the trace. The ILP branching behavior is represented as a sequence of executed conditions C_i, where C_i is the ith condition of the trace, and it has the execution frequencies $freq(t_{C_i})$ and $freq(f_{C_i})$ of its conditional-edges associated with it. The values of $freq(t_{C_i})$ and $freq(f_{C_i})$ are as given in the first ILP solution.

From the ILP branching behavior *ilp_bb*, WCET trace candidates are constructed by specifying their branching behavior *bb*. A branching behavior *bb* forms an execution trace, where the ith element of *bb*, denoted by $bb[i]$, stores the evaluation of the executed path-condition C_i. We refer to $bb[i]$ as the ith branch decision of *bb*. Depending on the execution frequencies of the conditional-edges $e \in \{t_{C_i}, f_{C_i}\}$ of C_i in *ilp_bb*, multiple branching behaviors *bb* can be constructed from *ilp_bb*, as follows.

Case 1. One conditional-edge of C_i is executed once. If only one of the conditional-edges e of C_i is executed with $freq(e)$, no interleaving among branch-conditions is possible. Therefore, a single WCET trace candidate is constructed whose $freq(e)$ positions are set either to t or f results. Assuming that the execution frequency of t_{C_i} (respectively, f_{C_i}) is 1, the path-condition C_i is assumed to evaluate to `true` (respectively, `false`), hence the branching behavior at position i is set to *t* (respectively, f). Using our previous notation, in this case we have:

$$bb[i] = \begin{cases} t, & \text{if } freq(t_{C_i}) = 1 \\ f, & \text{if } freq(f_{C_i}) = 1 \end{cases}$$

Case 2. One conditional-edge of C_i is executed repeatedly. If the conditional-edge e of C_i is executed with a frequency higher than 1, we need to encode the multiple executions of e. To this end, multiple positions in *bb* are set to either *t* or *f*, as follows:

$$\begin{cases} bb[i+j] = t \quad \text{with} \quad 0 \leq j \leq freq(t_{C_i}), \\ \qquad \text{if } freq(t_{C_i}) > 0 \text{ and } freq(f_{C_i}) = 0 \\ bb[i+j] = f \quad \text{with} \quad 0 \leq j \leq freq(f_{C_i}), \\ \qquad \text{if } freq(f_{C_i}) > 0 \text{ and } freq(t_{C_i}) = 0 \end{cases}$$

That is, a sequence of t (respectively, f) of length $freq(t_{C_i})$ (respectively, $freq(f_{C_i})$) is set starting from $ilp_bb[i]$. Note that for multiple conditionals inside a loop, the values of *bb* must be set such that their index coincides with the corresponding branching-decision in the trace.

Case 3. Both conditional-edges of C_i are executed repeatedly. If the ILP branching behavior specifies the execution of both conditional-edges of C_i inside a loop, the branching-decisions can interleave, and hence ilp_bb encodes multiple WCET trace candidates.

The number of WCET trace candidates constructed from ilp_bb is given by the number of all possible permutations over the set of edges $\mathcal{S} = \{ \underbrace{t \ldots t}_{freq(t_{C_i}) \text{ times}}, \underbrace{f \ldots f}_{freq(f_{C_i}) \text{ times}} \}$. Using the results of [27], the number of permutations over S, and thus the number of loop branching behaviors is: $p = \frac{\left(freq(t_{C_i})+freq(f_{C_i})\right)!}{(freq(t_{C_i})!) \, * \, (freq(f_{C_i})!)}$.

In this case, we take care of the multiple branching behavior as follows. We construct p copies of the current *bb*, that is we take $bb_1, \ldots, bb_p$ branching behaviors where each bb_x is a copy of *bb*. Next, each bb_x is continued by one loop branching behavior, as given below:

$$bb_x[i+j] = permutation_x\{ \underbrace{t, \ldots, t}_{freq(t_{C_l}) \text{ times}}, \underbrace{f, \ldots, f}_{freq(f_{C_l}) \text{ times}} \}$$
$$\text{with} \quad 0 \leq j \leq freq(t_{C_i}) + freq(f_{C_i}) \quad \text{and} \quad 0 \leq x \leq l,$$

where $permutation_x\{S\}$ gives the xth permutation over S. Note that for *bb* the correspondence between index i and executed condition C_i is not one-to-one; previous conditions C_k with $k < i$ might have already set multiple positions, including $bb[i]$, in *bb*.

4.2 Selective Symbolic Execution

Selective symbolic execution supports the analysis of a computable or measurable property (i.e. WCET) of a program under study, while exploring only the relevant parts (i.e. trace candidate) for analyzing the property. The goal is to minimize the number of symbolic executions required in order to improve on analysis results. The WCET Squeezing approach combines a symbolic execution engine with a WCET analysis toolchain and use WCET estimates to guide selective symbolic execution, by symbolically executing only those traces that might exhibit the WCET estimate.

Using the branching behavior *bb* of the WCET trace candidates, the symbolic execution engine of Algorithm 1 directs the program execution along these traces and, for each trace, checks the feasibility of the conditions on each branching point. A symbolically evaluated execution trace is feasible if the conjunction of all path conditions is satisfiable, meaning that the execution trace is a feasible program execution trace. As the symbolic execution engine is precise, it serves as an oracle to decide whether the ILP branching behavior is a feasible branching behavior in the concrete program.

For doing so, our symbolic execution step in line 5 of Algorithm 1 proceeds as follows. It takes as input the source code of the program and the branching behavior *bb* of one of the WCET trace candidates.

The symbolic execution engine then constructs a satisfiability module theory (SMT) representation [3] of the program execution, according to the branching behavior together with the source. A branching behavior *bb* of length n specifies the evaluations of n path-conditions, which can be analyzed for satisfiability in the SMT representation provided by the symbolic execution engine. That is, if the specified evaluation of the path-condition is unsatisfiable at some point, the trace $\pi(bb)$ is infeasible. Using our previous notations, we conclude that $\pi(bb)$ is infeasible iff the boolean expression:

$$symbolicEval(C_0, bb[0]) \; \wedge \; symbolicEval(C_1, bb[1]) \; \wedge \; \ldots \; \wedge \; symbolicEval(C_i, bb[i])) \tag{1}$$

is unsatisfiable, for $i \leq n$.

If the branching behavior *bb* gives a satisfiable evaluation of (1), the WCET trace candidate corresponding to *bb* yields a successful symbolic execution. Hence, the WCET trace candidate is feasible and exhibits the current WCET estimate. Therefore, no further WCET refinement is possible (line 6 of Algorithm 1).

Otherwise, if the symbolic execution of a trace candidate fails, some path-condition C_i in (1) is unsatisfiable for some i. This condition C_i can be mapped to its conditional nodes, resulting in an ILP encoding of an infeasible WCET trace candidate. Hence, the encoding yields a counter-example that needs to be excluded from the ILP branching behavior in the next iteration of WCET Squeezing (line 7 of Algorithm 1). The constraint constructed from this counter-example involves all symbolically executed conditions, as detailed in Section 4.3.

4.3 ILP Constraint Encoding

If a WCET trace candidate induced by an ILP branching behavior is infeasible, it is excluded from further WCET computations by adding a derived ILP constraint to the new ILP problem.

The construction of the ILP constraint is such that it decreases the total sum of execution frequencies of all conditional-edges that were symbolically executed until infeasibility was inferred (including the unsatisfiable one). That is, for an infeasible WCET trace candidate π, the ILP constraint constructed involves all conditional-edges corresponding to C_i from Equation (1). Using the notation from Section 4.2, recall that $bb[i]$ gives the conditional-edge (t or f) from the ith position of *bb*. $bb[i]_{C_i}$ is the conditional edge of C_i, denoted t_i or f_i in the ILP. Then, the conditional-edges of (1) over which a new ILP constraint is constructed are given by $bb[0]_{C_0}$, $bb[1]_{C_1}, \ldots, bb[i]_{C_i}$. To ensure that the execution frequencies of

```
void f () {
    if (C1) ...
    if (C2) ...
}
```

Figure 3: Conditions `C1` and `C2` are mutually exclusive.

these conditional-edges is decreased, the new ILP constraint we add to the ILP problem is:

$$bb[0]_{C0} + bb[1]_{C1} + \cdots + bb[i]_{Ci} \leq$$
$$\mathit{freq}(bb[0]_{C0}) + \mathit{freq}(bb[1]_{C1}) + \cdots + \mathit{freq}(bb[i]_{Ci}) - 1.$$

EXAMPLE 1. *Consider Figure 3, where conditions C_1 and C_2 are mutually exclusive. The (abstracted) CFG representation of Figure 3 is given in Figure 4(a). The initial ILP solution yields a WCET trace candidate with branching behaviour tt, i.e. an execution frequency that enables the execution of both $t1$ and $t2$ (both with execution frequency 1). However, as conditions C_1 and C_2 are mutually exclusive, only one of the conditions can be true. Therefore, symbolic execution will set C_1 to true, execute $t1$, and infer that the evaluation of $C_2 =$ true is unsatisfiable, hence execution of $t2$ is infeasible. The constraint constructed from the result of symbolic execution will specify in the resulting ILP problem that either C_1 or C_2 (but not both) is valid, by decreasing the combined execution frequency of all conditionals edges executed, i.e. of $t1$ and $t2$.*

In more detail, the derived ILP problem that includes the new constraint specifies the following properties: (i) the entry-edge n and the exit-edge x of the CFG of Figure 4(a) is executed at most once, that is $n \leq 1$ and $x \leq 1$; (ii) the conditional-edges of Figure 4(a) are executed at most once, that is $t1 + f1 \leq n$ and $t2 + f2 \leq n$; (iii) due to the new constraint, the combined execution frequency of the `true`-edges of C_1 and C_2 is restricted to 1, that is $t1 + t2 \leq 1$. Therefore, (iii) enforces that any ILP solution will assign a frequency of at most 1 for the combined execution frequency of the two `true`-edges. Thus, either `true`-edge $t1$ or `true`-edge $t2$ is executed, but not both – executing both edges requires a combined execution frequency of at least 2.

As illustrated in the example above, the ILP constraints constructed from infeasible WCET trace candidates restrict the combined sum of execution frequencies of all conditional-edges involved in the trace. Therefore, any valid solution of the new ILP problem must deviate in at least one conditional edge from the solution of the previous ILP problem.

Alternatively, the *syntactic-based* encoding introduces additional ILP variables and corresponds to a CFG transformation (Figure 4) that removes only the infeasible trace from the CFG. On the graph representation of the CFG, the following two transformations are applied: (i) the branching decision is made explicit in the CFG by copying and pulling up the unsatisfiable condition into the predecessor conditional node; (ii) the unsatisfiable edge for the prefix of the infeasible WCET trace candidate is removed.

EXAMPLE 2. *Consider again the program code given in Figure 3 and the CFG representation in Figure 4(a). Let $t1$ and $t2$ respectively denote the `true`-edge of C_1 and C_2. Assume that the ILP branching behavior for Figure 3 is initially tt, that is a WCET trace candidate executing $t1$ followed by $t2$. This WCET trace candidate is inferred infeasible by symbolic execution because the evaluation of condition C_2 to `true` (i.e. executing conditional edge $t2$) is unsatisfiable. Therefore, the trace is removed from the CFG of the program: the conditional node of the unsatisfiable edge is copied and pulled up into the last conditional node and the true-decision edge is removed for the prefix of the candidate.*

Implementation Pragmatics in the Presence of Loops. The difference between the syntactic- and the semantic-based encoding in the above examples only effects the number of ILP variables and constraints. In the presence of loops, using the syntactic-based encoding also increases the number of ILP variables and constraints,

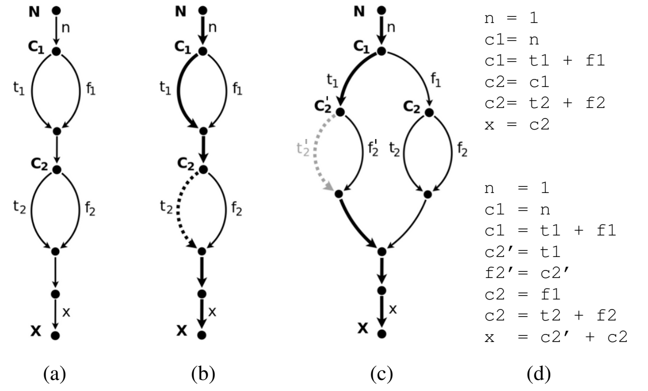

Figure 4: (a) CFG representation of Figure 3; (b) Infeasible WCET trace candidate (bold) with an unsatisfiable conditional edge (dotted); (c) Transformed CFG, excluding the infeasible trace of (b); (d) top, the original ILP of (a) and the modified ILP of (c), bottom.

while using the semantic-based encoding increases the number of WCET trace candidates.

Note that the execution frequencies of edges inside loops are constrained to their execution frequency times the loop bound. That is, for an edge that is executed m times inside a loop, the following constraint is generated: $\mathit{freq}(e) \leq \ell * m$, This needs to be taken into account for both encodings when computing the combined execution frequency for edges inside loops, as the following example illustrates:

```
void main () {
  int i;
  bool exec = false;
  if (*)
    exec = true;      // t1
  for (i = 0; i < 5; i++)
    if (exec == false) {
      expensive();    // t2
      exec = false;
    } else
      exec = true;    // f2
}
```

Figure 5: Executing the `true`-edge of the first conditional restricts the execution of the `true`-edge inside the loop. $*$ denotes non-deterministic choice, `f1` is not executing `t1`

```
       n <= 1;
      c1 <= n;
 t1 + f1 <= c1;
loopHead <= 1;
loopBody <= loopHead * 5;
loopBody <= t2 + f2;
loopExit <= loopHead;
       x <= loopExit;
```

Figure 6: ILP problem after WCET analysis of the example in Figure 5. The branching behavior imposed by the ILP solution is *tttttt*.

EXAMPLE 3. *Assume that the first true-block of Figure 5 (edge t_1, marked as `t1`) has an execution time of 1 and the second true-block (edge t_2, marked as `t2`) has 10. All other costs are ignored. The initial ILP solution selects all true-blocks to be executed, the t_1 with frequency 1 and the t_2 with frequency 5. The reported WCET estimate amounts to $1 * 1 + 5 * 10 = 51$, the branching behavior for the trace is tttttt (i.e. $t_1 t_2^1 \ldots t_2^5$).*

Symbolically executing this trace yields the unsatisfiability of the second condition (`exec == false`), in the first iteration of the loop. Thus, if the first branching decision is t, the execution frequency of the `true`-edge in the loop is only 4. Just reducing the frequency in the ILP problem to 4 yields an invalid result. The WCET estimate would be restricted to $1 * 1 + 4 * 10 = 41$, even though there exists a path exposing a higher WCET: Executing the false-branch of the first conditional (branching behavior fttttt) leads to a WCET of $0 * 1 + 5 * 10 = 50$ and is allowed in the original

problem.

We denote the situation illustrated in Example 3 as candidate-flip. If the prefix to a loop flips, the constraints about loop iterations need to be inferred again, i.e. it increases the number of symbolically executed WCET trace candidates. Both the syntactic- and the semantic-based encoding can handle candidate-flips, but based on our experience, a combination of the two encodings handles them best.

On CFG level, the syntactic-based approach peels-off a loop iteration [24] and introduces a new conditional for the loop condition in the first iteration and children for the loop body. Similar to Example 2, the last conditional is split, introducing copies for all following edges, and then the infeasible trace is removed. On the ILP level, the same technique is applied: additional variables for the copies of the condition and the loop-peel are introduced in the ILP problem. At the same time, the number of WCET trace candidates is smaller, as branching in different iterations is explicitly encoded.

The semantic-based encoding constructs, as before, a constraint that decrements the combined execution frequency of all conditional-edges up to the loop and the execution frequency of the conditional-edge in the loop. Additionally to the original ILP (Figure 6), the constraint restricts the combined execution frequency of the `true`-edge (`t1`) of condition `c1` and the `true`-edge (`t2`) of the condition in the loop body, i.e. `t1 + t2 <= 5`. (Figure 7). This constraint introduces no new ILP variables, but the ILP branching behavior of encodes more WCET trace candidates.

The combined encoding uses the syntactic-based encoding to peel one loop iteration and decrements the execution frequency of edges inside the loop. It does not split conditional nodes, instead a semantic-based encoding is used to restrict the combined execution frequency of the loop prefix and the peeled condition, such that the branching decision in the first iteration of the loop is explicit for the loop prefix. The branching decision in the first iteration of the is not restricted for other executions. Figure 8 depicts the resulting ILP problem where the solver will return the correct WCET estimate, i.e. branching behavior *fttttt*, exhibiting a WCET of 50. The advantage of the combined encoding is that it only introduces ILP variables and constraints for the peeled loop iteration and at the same time makes explicit the branching behavior, therefore reducing the number of symbolically executed WCET candidates.

The derived ILP problem in Figure 8 uses a combined constraint. Additional ILP variables are introduced to model peeling off the first loop iteration (syntactic), an additional constraint (semantic) restricts the combined execution frequency of the `true`-edge of the first conditional and the `true`-edge of the peeled loop iteration. Thus, any solution that selects the first `true`-edge $t1$ is then restricted to pick the `false`-edge *peelF* in the peeled (1st) iteration of the loop. If a solution picks the `false`-edge of the first conditional, the restriction of the semantic constraint does not apply and any valid interleaving of true and `false`-edges in the loop can be picked (including the `true`-edge in the peeled iteration).

```
       n <= 1;
      c1 <= n;
 t1 + f1 <= c1;
loopHead <= c1;
loopBody <= loopHead * 5;
loopBody <= t2 + f2;
loopExit <= loopHead;
 t1 + t2 <= 5;
       x <= loopExit;
```

Figure 7: Semantic constraint.

```
               n <= 1;
              c1 <= n;
         t1 + f1 <= c1;
        peelCond <= c1;
 peelT + peelF<= peelCond;
      t1 + peelT <= 1;
       loopEntry <= peelCond;
        loopBody <= loopEntry * 4;
        loopBody <= t2 + f2;
        loopExit <= loopEntry;
               x <= loopExit;
```

Figure 8: Combined constraint.

5. EXPERIMENTAL RESULTS

In this section we describe our implementation and report on our experimental findings on the Mälardalen WCET benchmark suite [17].

Implementation. Our WCET analysis toolchain r-TuBound [22] uses CalcWCETC167 as the low-level WCET analyzer [21], which applies IPET using the ILP solver lp_solve [4]. In order to support WCET Squeezing in r-TuBound, we made the following modifications and extensions. We changed the CalcWCETC167 engine of r-TuBound such that it extracts a WCET trace candidate and its ILP branching behavior, by relying on the ILP solution of lp_solve and the assembly. Further, we extended r-TuBound with a symbolic execution engine. To this end, we made use of the symbolic execution tool SmacC [5] and modified SmacC such that a symbolic execution of the program can be selected based on branching behaviors. If the ILP branching behavior encodes multiple concrete executions, we take all possible path permutations for a specific conditional block frequency in a loop and symbolically execute the resulting traces. The construction of additional ILP constraints can be fully automated but for this work we relied on a manual mapping between blocks in the assembly and the source: we constructed WCET trace candidates from ILP problems and solutions using this mapping. Both r-TuBound and SmacC perform intra-procedural analysis.

Results. All experiments were performed on an Intel Core i5 CPU M540@2.53GHz with 4GB of main memory. In our experiments, we restricted the running time of symbolic execution to 20 minutes, in some cases this limit was reached. We note that our symbolic execution engine does currently not perform additional optimizations for improving symbolic execution. Additionally, the calls to the symbolic execution engine are not yet done in an incremental manner, meaning that our symbolic execution approach now re-executes even those paths that were already shown infeasible.

We evaluated WCET Squeezing on 10 examples taken from the Mälardalen WCET benchmark suite. We summarize our results in Table 1. Column 1 of Table 1 reports the benchmark's name and Column 2 lists the relevant functions in the benchmark. Column 3 gives the initial WCET estimate as reported after WCET analysis by r-TuBound. Column 4 describes the WCET estimate obtained after running WCET Squeezing. Column 5 lists how many iterations of WCET Squeezing were executed and Column 6 gives the number of execution traces that were excluded. Columns 7 and 8 report on the obtained WCET improvements, as follows: Column 7 describes the achieved improvement (i.e. the new WCET bound), whereas Column 8 denotes the maximum improvement (i.e. the actual WCET bound of a feasible path derived by WCET Squeezing).

For the functions `prime`, `cl_block`, and `icrc1` the initial WCET candidate is feasible, hence the initial bound is proved to be precise. Proving this is out of scope of state-of-the-art WCET tools. For the functions `adpcm`, `duff`, `expint`, and `fibcall` continuous improvement is achieved until WCET Squeezing terminates, this time proving precision of the squeezed WCET. Functions `expint` and `janne_complex` can both be tightened by more than 90%. In Table 1 we only report the impact of a single iteration of WCET Squeezing to demonstrate the different effect of excluding a single WCET trace candidate: while the impact amounts to roughly 1% for the first program, it amounts to almost 6% for the second. This reflects the fact of how much the excluded conditional block contributes to the WCET estimate of the function. Similarly, this holds for `lcdnum` and `nsichneu`: executing

1:benchmark	2:function	3:WCET	4:SQZ	5:#iters	6:#π	7:%imp	8:%prec	9:note
prime	prime	784860	784860	1	0	0	0	precision proved w/o refinement
compress	cl_block	8440	8440	1	0	0	0	precision proved w/o refinement
crc	icrc1	18060	18060	1	0	0	0	precision proved w/o refinement
adpcm	logsch	5560	5380	1	1	3.24	3.24	precision proved w/ refinement
	uppol2	12260	12040	2	2	9.87	9.87	precision proved w/ refinement
duff	duffcopy	85940	79949	5	5	7.5	7.5	precision proved w/ refinement
fibcall	fib	79840	75040	2	2	6.39	6.39	precision proved w/ refinement
expint	expint	1.24E7	1.22E7	1	1	0.94	93	imprecise
janne-complex	complex	694980	653380	1	1	5.9	93.3	imprecise
lcdnum	main	24320	22520	2	11	7.4	18.5	imprecise
nsichneu	main	4.95E6	4.94E6	2	2	0.28	-	imprecise, actual WCET unknown

Table 1: Proving precision of WCET estimates by running WCET Squeezing until termination. The lower part focuses on the impact of only a few iterations.

two iterations of WCET Squeezing results in an improvement of more than 7% for `lcdnum` and less than 0.5% for `nsichneu`. For `expint`, the high over-estimation of the WCET is due to the fact that the WCET toolchain initially assumes the inner loop to be executed in every iteration. Squeezing reveals that the inner loop is only executed in the last iteration. Similarly, in `janne_complex` over-estimation is due to a complex interleaving of nested loops, ultimately inferred by WCET Squeezing. We evaluated WCET Squeezing on 10 examples taken from the Mälardalen WCET benchmark suite in detail.

6. RELATED WORK

To the best of our knowledge, WCET Squeezing is the first approach which tightens and proves precise the WCET bound of a program after an initial WCET analysis in the presented anytime-manner. WCET Squeezing is also the first method that applies symbolic execution in conjunction with a WCET analysis toolchain in order to improve the WCET estimates for a real processor.

As WCET Squeezing makes use of both symbolic execution and WCET analysis, in the sequel we describe the works that are the most related ones to our approach.

WCET analysis. Static WCET analysis is performed using timing analysis tools which need flow- fact information about the program under analysis. Such information may be given manually by the developer or inferred automatically by a flow fact analyzer and includes information about execution frequencies of blocks and loop bounds for program loops. Modern static WCET analyzers, see e.g. [22, 1, 16], usually rely on the IPET technique [26] to calculate a WCET estimate. IPET usually over-estimates the WCET, as the constructed ILP problem encodes numerous spurious program executions that are infeasible in the concrete program. WCET Squeezing can be used *in addition* to IPET-based WCET analysis, overcoming this deficiency.

The main difference between WCET Squeezing and other approaches that eliminate infeasible paths in the WCET computation model is that WCET Squeezing is applied as a post-process to WCET analysis and that it yields a proof of precision. The proof is found fast if the initial WCET bound is tight. Therefore, WCET Squeezing profits from tight initial bounds and can be used in conjunction with other approaches.

In [23] the authors use the WCET estimate for program optimizations improving the WCET of the program: the WCET estimate is used to effectively bound the number of loop unrollings during WCET analysis. Since the size of the program to be analyzed changes during loop unrollings, constraints on available memory and cache must be considered to guarantee improvements of the WCET bound. To this end, during a loop unrolling additional information about the hardware is used in [23]. Combining and intertwining IPET with a cache and pipeline analysis is also discussed in [14]. Similarly to [23, 14], in WCET Squeezing we also use the WCET estimates to guide program analysis, in particular to identify the relevant program parts for symbolic execution. However, our approach is not platform dependent. Nevertheless, extending WCET Squeezing with hardware-aware information is an interesting task to be investigated, especially for applying symbolic execution on the binary model and refining the low-level WCET computation model.

In [28] a method to incorporate static path exclusions into a static WCET analyzer is described. This approach is similar to the def-use refinement of [7]. To this end, conditionals are inspected and a formula that characterizes the condition is encoded in Pressburger arithmetic. Then, an off-the-shelve solver is used to check whether the evaluation of one conditional enforces a specific evaluation of a different conditional. This way, additional ILP flow facts are inferred and can be added to the ILP, excluding paths that are infeasible due to the static evaluation of the conditionals. When compared to WCET Squeezing, we note that the constraints derived by [28] can also be derived by our method: the information about the evaluation of conditionals is implicitly carried in our symbolic execution. Similarly to [28], WCET Squeezing also profits from a precise a-priori path-feasibility analysis. However, the proof of precision obtained during WCET Squeezing cannot be derived using [28].

WCET Squeezing relies on symbolically executing worst-case candidates. A traditional application of symbolic execution is test-case generation, where path constraints are solved in order to infer input data that leads to program executions along this path. To this end, our symbolic execution engine could be replaced by other techniques, for example by the method of [19], for deriving input values for executed paths. With such an extension, WCET Squeezing could also compute the input that leads execution along the worst- case candidates.

A somewhat similar approach to WCET Squeezing is also described in [15], where worst-case input values are obtained by partitioning the input space as follows. A WCET analysis is performed and a WCET estimate is calculated. The domain of possible input values is tracked and separated into smaller ranges in each iteration of the approach. For each partition, a WCET estimate is then inferred. Finally, when there is only one input combination in a partition, the WCET is returned. The approach of [15] can infer precise WCET estimates by shrinking input ranges, while keeping (possibly over-estimated) results for larger partitions. Similarly to WCET Squeezing, the algorithm of [15] is also an anytime algorithm by improving the WCET estimate at every iteration of the algorithm.

Upon termination of [15], all input values are partitioned according to their WCET bound. Thus, the partition that exhibits the highest WCET bound is the input range leading a program execution towards the WCET path. When compared to WCET Squeezing, we however note that the method of [15] re-analyzes the entire program for smaller input domains as well, while in WCET Squeezing we only re-calculate the ILP solution at every iteration of Squeezing. A drawback of [15] comes therefore from the fact it might still consider spurious traces for WCET calculation, because the analysis does not infer additional results about infeasible paths. Our notion of precision is hence out-of-scope of [15].

In contrast to the approaches presented above, the work described in [2] addresses the classical problem of WCET computation by using ILP in conjunction with model checking. Similarly to WCET Squeezing, the approach of [2] maps the ILP encoding of a program to a program trace and then checks feasibility of this trace. However, unlike our method, the feasibility constraints of program paths are encoded in [2] as program assertions in the original program, which are then verified by using a software model checker. Contrarily to our path-wise selective symbolic execution approach, the advantage of path-local reasoning is hence not explored in [2].

Related, though conceptually different is the approach of [8]. Here, irrelevant program parts are identified via the criticality of basic blocks, which denotes the relation between the longest path through the basic block and the WCET of the program. Eliminating the irrelevant parts from the program allows usage of a more precise but computationally more expensive WCET analyzer which then might come up with a more precise WCET bound.

A different approach to WCET analysis is given in [11], where a formal framework for quantitative abstraction refinement is presented using abstract interpretation [13]. The method relies on segment- and state-based abstract interpretation. The state-based approach [11] has some similarities with our syntactic-based abstraction refinement technique, i.e. counter-example encoding that makes control-flow decisions explicit in loop iterations. We are currently investigating whether a combination of [11] with WCET Squeezing would yield tighter WCET estimates and a faster termination of our algorithm.

Symbolic Execution. In contrast to techniques that analyze the program as a whole (e.g. model-checking), symbolic execution reasons in a path-local manner. Symbolic execution is especially suited for automated testing but also found applications in program verification and bug-hunting [10, 5]. It allows for precise analysis of programs but the number of program paths that need to be analyzed increases exponentially with the number of conditionals. Therefore, applications of symbolic execution often target only partial symbolic coverage of the program, e.g. generating test-cases that achieve high line coverage [9].

One of the major advantages of symbolic execution engines is the amount of information they can infer about a program, as they implicitly carry all this information along when they explore a program. Applications of symbolic execution for loop bound refinement in WCET analysis is briefly addressed in [22]. There, symbolic execution is used to infer and validate arithmetic properties about program loops, in some cases allowing to refine the computed loop bound.

Unlike the above mentioned works, our technique relies on the tight interaction between a traditional static WCET analysis toolchain applying IPET in combination with a symbolic execution engine that allows to select symbolic execution traces and to precisely reason about path-conditions. We exploit the fact that loop bounds are implicitly supplied by the initial WCET analysis. Compared to traditional flow-fact analysis, the symbolic execution component of WCET Squeezing infers precise constraints for paths that constitute to the WCET estimate. In each iteration of WCET Squeezing, solving the new ILP problem allows to refine the WCET estimate. Compared to traditional symbolic execution engines, WCET Squeezing offers a way to identify precisely which paths need to be symbolically executed in order to improve the analysis.

Symbolic execution is also exploited and adjusted in [20, 18], as follows. The method of [20] avoids unfolding program loops, and hence omits multiple executions of the same block. While [20] analyses each program block separately, in the selective symbolic execution approach of WCET Squeezing we analyse only the relevant program blocks and paths. The abstract execution framework of [18] can be seen as a specialized form of symbolic execution, combining the abstract interpretation framework with symbolic execution. While the symbolic execution engine in WCET Squeezing analyzes each trace in isolation of others, abstract execution is a whole-program analysis that applies abstract state merging, which might lead to information loss. Compared to WCET Squeezing, both approaches aim at computing a tight WCET bound, but neither of them can improve nor prove precise the computed bounds. WCET Squeezing is designed as a post-process to an initial WCET analysis and benefits from its focus on single candidate traces.

7. CONCLUSION

WCET Squeezing brings a new and powerful approach for computing precise WCET bounds for safety-critical real-time systems. It is an on-demand anytime algorithm that is applicable as a post-process to any state-of-the-art IPET-based WCET analyzer. WCET Squeezing handles the classical WCET problem by proving WCET bound precise, yielding an automated approach that is out of scope of traditional WCET analyzers. WCET Squeezing is also the first technique to handle the pragmatic WCET problem.

Conceptually, WCET Squeezing iteratively refines the program model by continuously excluding more and more infeasible execution traces from the program model. Technically, WCET Squeezing is put on top and combines a traditional WCET analysis toolchain with a symbolic execution engine that selectively analyzes only WCET-relevant parts of the program. This way, WCET Squeezing adds WCET-relevant information to the coarse program abstraction of an initial IPET analysis, but avoids the expensive computational costs of full symbolic program execution.

Our experimental data on a set examples taken from the Mälardalen WCET suite show significant improvements of WCET precisions already after a few iterations of WCET Squeezing. For example, it is not unusual that WCET bounds are improved up to 9% after only two iterations of WCET Squeezing. Even more, WCET Squeezing is often able to compute a proven tight bound, i.e., the actual WCET of the program. Note that this is out-of-the-scope of traditional WCET analyzers as they do not have means to checking the feasibility of a computed WCET path.

Currently, we are working on fully automatizing WCET Squeezing, i.e. automatizing the construction of the mapping from ILP to source as well and on porting WCET Squeezing to other WCET toolchains. Further work also includes the integration of a cache and pipeline analysis into WCET Squeezing, and hence making WCET Squeezing architecture dependent.

8. REFERENCES

[1] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. OTAWA: an Open Toolbox for Adaptive WCET Analysis. In *Proc. of IFIP Workshop – SEUS*, 2010.

[2] H. J. Bang, T. H. Kim, and S. D. Cha. An Iterative Refinement Framework for Tighter Worst-Case Execution Time Calculation. In *Proc. of ISORC*, pages 365–372, 2007.

[3] C. Barrett, A. Stump, C. Tinelli, S. Boehme, D. Cok, D. Deharbe, B. Dutertre, P. Fontaine, V. Ganesh, A. Griggio, J. Grundy, P. Jackson, A. Oliveras, S. Krstić, M. Moskal, L. D. Moura, R. Sebastiani, T. D. Cok, and J. Hoenicke. C.: The SMT-LIB Standard: Version 2.0. Technical report, 2010.

[4] M. Berkelaar, K. Eikland, and P. Notebaert. *lp_solve P*. Software, 2004. Available at `http://lpsolve.sourceforge.net/5.5/`.

[5] A. Biere, J. Knoop, L. Kovács, and J. Zwirchmayr. SmacC: A Retargetable Symbolic Execution Engine. In *Proc. of ATVA*, pages 482–486, 2013.

[6] A. Biere, J. Knoop, L. Kovács, and J. Zwirchmayr. The Auspicious Couple: Symbolic Execution and WCET Analysis. In *Proc. of WCET*, pages 53–63, 2013.

[7] R. Bodík, R. Gupta, and M. L. Soffa. Refining Data Flow Information Using Infeasible Paths. *SIGSOFT Softw. Eng. Notes*, 22(6):361–377, 1997.

[8] F. Brandner and A. Jordan. Refinement of Worst-Case Execution Time Bounds by Graph Pruning. 2013. Under submission.

[9] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. of OSDI*, pages 209–224, 2008.

[10] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically Generating Inputs of Death. *ACM Trans. Inf. Syst. Secur.*, 12(2):10:1–10:38, 2008.

[11] P. Cerny, T. Henzinger, and A. Radhakrishna. Quantitative Abstraction Refinement. In *Proc. of POPL*, pages 115–128, 2013.

[12] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Proc. of CAV*, pages 154–169, 2000.

[13] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *In Proc.of POPL*, pages 238–252, 1977.

[14] C. Cullmann and F. Martin. Data-flow based detection of loop bounds. In *Proc. of WCET*, 2007.

[15] A. Ermedahl, J. Fredriksson, J. Gustafsson, and P. Altenbernd. Deriving the worst-case execution time input values. In *Proc. of ECRTS*, pages 45–54, 2009.

[16] J. Gustafsson. SWEET: SWEdish Execution Time tool. `http://www.mrtc.mdh.se/projects/wcet/sweet.html`, 2001.

[17] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET Benchmarks: Past, Present And Future. In *Proc. of WCET*, pages 136–146, 2010.

[18] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis Using Abstract Execution. In *Proc. of RTSS*, pages 57–66, 2006.

[19] A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith. FShell: Systematic Test Case Generation for Dynamic Analysis and Measurement. In *Proc. of CAV*, pages 209–213, 2008.

[20] D. Kebbal and P. Sainrat. Combining Symbolic Execution and Path Enumeration in Worst-Case Execution Time Analysis. In *Proc. of WCET*, 2006.

[21] R. Kirner. The WCET Analysis Tool CalcWcet167. In *Proc. of ISoLA*, pages 158–172, 2012.

[22] J. Knoop, L. Kovács, and J. Zwirchmayr. r-TuBound: Loop Bounds for WCET Analysis. In *Proc. of LPAR*, pages 435 – 444, 2012.

[23] P. Lokuciejewski and P. Marwedel. Combining Worst-Case Timing Models, Loop Unrolling, and Static Loop Analysis for WCET Minimization. In *Proc. of ECRTS*, pages 35–44, 2009.

[24] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., 1997.

[25] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York Inc., 1999.

[26] P. P. Puschner and A. V. Schedl. Computing Maximum Task Execution Times – A Graph-Based Approach. *Real-Time Systems*, 13(1):67–91, 1997.

[27] S. S. Skiena. *The Algorithm Design Manual*. Springer Inc., 2nd edition, 2008.

[28] I. Stein and F. Martin. Analysis of Path Exclusion at the Machine Code Level. In *Proc. of WCET*, 2007.