Replacing Conjectures by Positive Knowledge: Inferring Proven Precise Worst-Case Execution Time Bounds Using Symbolic Execution

Jens Knoop

Institute for Computer Languages, Compiler and Languages group,
Vienna University of Technology, Argentinierstraße 8/4, 1040 Vienna, Austria

Laura Kovács

Department of Computer Science and Engineering,
Chalmers University of Technology, Rännvägen 6B, 41296 Gothenburg, Sweden

Jakob Zwirchmayr

TTTech Computertechnik AG, Schönbrunner Straße 7,
1040 Vienna, Austria
This research was carried out while affiliated with the
Institute for Computer Languages, Compiler and Languages group,
Vienna University of Technology,
and is not related to TTTech.

Abstract

Embedded real-time software systems (ESS) play an important role in almost every aspect of our daily lives. We do rely on them to be functionally correct and to adhere to timing-constraints ensuring that their computational results are always delivered in time. Violations of the timing-constraints of a safety-critical ESS, such as an airplane or a medical control device, can have disastrous economic and social consequences. Identifying and correcting such violations is therefore an important and challenging research topic. *In this article we address this challenge and describe a rigorous approach for the timing analysis of programs and for proving its results precise*.

In practice most important is the worst-case execution time (WCET) of an ESS, that is, the maximal running time of the system on a specified hardware. A WCET analysis needs to provide a formal guarantee that a system meets its timing-constraints even in the worst case. This requires to compute a safe and tight bound for the execution time of a program. Existing WCET tools, however, are usually not able to guarantee that there is a feasible system trace that takes indeed as long as stated by the computed execution time bound: often, due to the employed abstractions during static analysis a computed WCET bound overestimates the actual WCET bound, since loop bounds and other timing-relevant program properties their computation is based on are computed for spurious infeasible system traces that can be ruled out by a path-sensitive program abstraction and analyzing program resources.

In this article we present an approach for *inferring and proving WCET bounds precise*. This approach guarantees that the WCET bound is computed for a feasible system trace. This is achieved by combining WCET computation with path-wise symbolic execution in an abstraction refinement loop. This way, symbolic execution can be targeted and limited to relevant program parts thereby taming and avoiding the usually prohibitive computational costs of symbolic execution with a full path coverage. Moreover, our approach improves the quality of the underlying WCET analysis, since it automatically tightens the bound until it is proven precise, thereby improving the precision of the initially computed WCET bound.

Our overall approach is an *anytime algorithm*, i.e., it can be stopped at any time without violating the soundness of its results. If run until termination, the WCET bound is proven precise, by automatically inferring additional constraints from spurious traces and using these constraints in the abstraction refinement.

We implemented our approach in the r-TuBound WCET toolchain and tested it on challenging benchmarks from the WCET community. Our experimental results underline the advantage of using symbolic methods for proving WCET bounds precise, at a moderate cost.

Key words: Worst-Case Execution Time (WCET) Analysis, Static Symbolic Execution, Symbolic Computation

1. Introduction

Embedded (real-time) software systems (ESS) are ubiquitous and used on a daily basis these days. They are usually composed from some application-specific hardware device (i.e. a microprocessor) that is connected to the world and receives data from sensors or some input system, and processes them by executing software tasks. There are certain demands that one can impose

Email addresses: knoop@complang.tuwien.ac.at (Jens Knoop), laura.kovacs@chalmers.se (Laura Kovács), jakob@complang.tuwien.ac.at (Jakob Zwirchmayr).

on such systems: first of all, they shall be *functionally correct*; intuitively that is they shall operate as expected, in particular, they shall not crash.

Ensuring functional correctness and providing the expected quality is a challenging research topic, both in academia and industry. Rigorous approaches based on formal methods have been developed to address and solve this challenge, including extensive testing of systems until one is sufficiently confident that no errors exist or formally verifying the underlying hardware and software system, mathematically proving the absence of errors.

Real-time embedded software systems must additionally satisfy *timing-constraints*. These are constraints on the execution time of a system requiring that computational results must be delivered within the required time limit. They are usually dependent on the underlying hardware, for example, the operating frequency or the memory cache and pipeline configuration of the underlying processor.

Depending on the application context, different levels of the criticality of the timing behavior of a real-time embedded software system can be distinguished. For example, a Voice-Over-IP application, that is an audio or video stream, where the order in which video or audio packets arrive is important, is a system of low(er) timing criticality: packets arriving too late or out of order result initially only in a degradation of the quality of the service, though ultimately the system might be rendered unusuable. For hard real-time systems, however, results are useless once the expected timing behavior is violated.

Verifying the compliance to timing-constraints is particularly challenging and crucial for *safety-critical real-time systems* upon which safety and even human lives can depend on. For example, fly-by-wire and drive-by-wire systems used in the avionics and automotive industries, with no mechanical links between the control column and the steering gear of an aircraft or car (Cary R. Spitzer, 2001), are safety-critical real-time systems. Missing a deadline is not an option for safety-critical real-time ESS, since this can have tragical economic and social consequences, resulting possibly even in the loss of human lives. As an example, consider the system controlling the airbags in a car. In case of an accident, the system needs to act within a time limit of around 40ms. When the sensor data registers an accident condition, the system must compute the required actions and supply the appropriate commands to the respective actuators, opening the airbag within this 40ms time frame. If under some circumstances the system requires longer to fully inflate the airbag, the system must be considered erroneous: the driver could already have been hurt severely, while the airbag is still inflating after more than 40ms. Therefore, for safety-critical real-time systems, ensuring functional correctness as well as verifying their timing behavior is equally important.

One of the most critical tasks in the timing analysis of safety-critical ESS is the *worst-case execution time (WCET) problem*. The WCET problem is to find an upper bound on the maximum execution time of a system. Any execution of the system is then guaranteed to be below the inferred WCET bound. We refer to this WCET problem as the *classical WCET problem*.

The need for sound and precise analysis methods for safety-critical ESS systems is growing at a fast pace. This is due to the fact that more and more functionality in such systems is realised within software that runs on less application-specific hardware (Milinkovich, 2014): for example, on Airbus aircrafts, the code size was less than a megabyte (i.e. 0.02MB) in 1975; in 1993 it grew to 12MB, and nowadays it has a size of more than 100MB in modern airplanes. Modern cars contain between 40-80 electronic control units and up to 90% of new technologies are nowadays realised in software, with the infotainment system of some cars consisting of around 20 millions lines of code. Due to the enormous increase in usage and size of software over the last decades, automated approaches to verify the safety, including the temporal safety, of such systems are essential.

State-of-the-art WCET analysis methods offer an automated approach to verify and guarantee a proper timing behavior of an ESS. WCET analysis can be performed either statically or dynamically. Dynamic WCET analysis needs concrete input data to run and measure system executions. Finding a safe WCET bound by dynamic analysis requires to measure all program executions, which often is a practically infeasible task. Therefore, dynamic WCET analysis usually yields an under-estimation of the WCET bound. It does not give any guarantees on the safety and precision of the returned WCET. By contrast, static WCET analyzers inspect a program without running it, by using abstraction techniques to exclude the possibility of under-estimation and to also ease the analysis. Typically, this results in an over-estimation of the actual bound, but guarantees its safety. However, neither dynamic nor static approaches are able to conclude a precise timing behavior of the system, or even more importantly, to prove or disprove that the computed WCET bound is precise. Whenever an insufficient timing behavior is reported by static analysis, it is unclear whether this behavior results from an actual but overly high WCET, or whether the timing limits are actually met but the reported WCET bound is over-estimated. In such a situation, *a proof of precision of the WCET bound would be especially useful*. With such a proof at hand, one could either (i) establish the precision of an overly high WCET bound, and hence conclude that a required smaller WCET bound cannot be reached; or (ii) prove that the current WCET bound is not precise, and hence might conclude system safety only when an improved bound is derived.

In this article we address the WCET problem by means of static WCET analysis. We describe an *automated framework for inferring WCET bounds and proving them precise*. This is achieved by combining techniques from symbolic execution, automated reasoning and WCET analysis. Our method proves or disproves that a computed WCET bound is precise, extending thus significantly the state-of-the-art in static WCET analysis.

Given an initial WCET bound computed by a state-of-the-art WCET analyzer, our framework iteratively tightens this bound until it proves it precise, as follows. As the (initial) WCET bound is computed on an abstraction of the concrete program, program paths exhibiting the WCET bound of the abstract program model might turn out infeasible once the actual behavior of the program is considered, identifying thus spurious WCET program traces. Our method maps the WCET bound to one or more traces in the abstract program model exhibiting this bound. Mapping the current WCET bound back to a trace in the concrete program allows us to check the feasibility of this trace. If the trace inferred from the program abstraction is feasible in the concrete program, the computed WCET bound of the trace is precise and the timing behavior is indeed exhibited by the program. The proof of precision is given by a satisfiable path feasibility check of the decision procedure, formulated as a satisfiability modulo theory (SMT) problem. On the other hand, if the decision procedure reports the infeasibility of the trace, we conclude that the current WCET bound describes a spurious program behavior which is exhibited only in the abstract program model. In this situation, the WCET bound is a WCET over-estimation and hence imprecise. Our proof of this imprecision is given by the path infeasibility result of the decision procedure.

Our approach iteratively tightens the WCET bound while searching for a proof of precision. If the WCET bound reported by a WCET analyzer is over-estimated, our method tightens the bound until the WCET bound is proven precise. If the initial bound is highly over-estimated then the computational costs of our algorithm might be high, too, assuming that then the number of necessary symbolic executions might also be high. However, due to the maturity of state-of-the-art WCET analyzers, and indicated also by our practical experiments, such a scenario is quite unlikely to happen as the over-estimation of state-of-the-art WCET analyzers is low, around 25%

higher than measured execution times even for complex architectures (Souyris et al., 2005). It is thus reasonable to assume that the over-estimation of the initial WCET bound is low or almost none. As demonstrated by our experimental findings, our method for proving inferred WCET bounds precise comes at a moderate computational cost in practice.

Contributions. We study the WCET problem in the timing analysis of embedded software systems. The main contribution of our work comes with a novel and automated framework for proving WCET bounds precise. We call this method WCET Squeezing. Unlike existing approaches, WCET Squeezing not only derives WCET bounds, it also proves a derived WCET bound precise.

For making WCET Squeezing applicable to WCET analysis, we develop the following new results:

(1) We present a decision procedure that allows to *decide whether a WCET bound reported by a WCET analyzer is precise*. This procedure combines symbolic execution with the implicit path enumeration (IPET) technique of WCET analysis. The result of our method is either a proof of precision or a counter-example, that is a proof of the infeasibility of the program path exhibiting the current WCET bound. This is out-of-scope of traditional static WCET analysis tools, which are not able to infer whether a computed WCET bound is precise or over-estimated on a particular program and hardware model.

This method is called *WCET Squeezing*. WCET Squeezing automatically tightens an imprecise WCET bound by inferring additional constraints from infeasibility proofs of program paths. The constraints derived by WCET Squeezing can be used in further WCET computations, tightening the bound until it is precise.

(2) Based on the WCET Squeezing algorithm, we introduce and define the *pragmatic WCET problem* as the following decision problem. The pragmatic WCET problem is the problem to decide whether a WCET bound reported by a static analyzer can be tightened and proved to be below some required threshold. Solving the pragmatic WCET problem requires to decide whether a WCET bound is precise: if the bound is precise and above the threshold, the answer to the pragmatic WCET problem is “no.” If the bound can be tightened and hence proved to be below the threshold, the answer to the pragmatic WCET problem is “yes."

We propose WCET Squeezing as a *decision procedure solving the pragmatic WCET problem*. As an anytime algorithm WCET Squeezing can be run until the WCET bound is below the required threshold, or it can be shown that this can not be achieved. Note that even a single iteration of WCET Squeezing might allow to verify a (user-supplied) WCET bound, and even terminating WCET Squeezing at an arbitrary iteration guarantees still an improved and more precise WCET bound. Moreover, WCET Squeezing can also be run until some time budget for the analysis is exhausted.

(3) Our overall approach with WCET Squeezing as its core component for proving WCET bounds precise is implemented in the *r-TuBound tool*. We have evaluated r-TuBound on a variety of WCET benchmarks and compared our results against the TuBound WCET analyzer (Prantl et al., 2008). Our experiments show that WCET Squeezing in conjunction with the automated computation of tight loop bounds significantly improves the tightness of inferred WCET bounds.

Structure of the article. Section 2 describes our programming model and overviews the relevant theoretical background in WCET analysis and symbolic execution. In Section 3, we demonstrate WCET Squeezing on an illustrating example. Subsequently, we formally introduce the WCET Squeezing algorithm for proving WCET bounds precise in Section 4.

Our implementation and experimental results are described in Section 5, followed by a discussion and ideas for future improvements in Section 6. We overview related work in Section 7 and draw our conclusions in Section 8. Initial results of this article have been published in our conference proceedings papers (Knoop et al., 2011, 2012, 2013). The current article, however, extends these previous works by introducing WCET Squeezing as a proof procedure for proving WCET bounds precise and solving both the classical and pragmatic WCET problem.

2. Preliminaries

Our WCET Squeezing method combines techniques from symbolic execution, WCET analysis, symbolic computation and formal verification. This section introduces the notation used throughout the paper and summarizes the relevant theoretical background.

Programming model. We represent programs as standard control flow graphs (CFG), $\mathit{CFG} := ((V, E), S, X)$, where V is the set of nodes describing program locations, 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 (Nielson et al., 1999). For each edge $e \in E$ an edge weight $w(e)$ is assigned denoting the execution time of e, and we hence have $w : E \to \mathbb{N}$. Every node n, different than S and X, has incoming $\mathit{inc}(n)$ and outgoing edges $\mathit{out}(n)$. The node S has only outgoing edges and no incoming ones, whereas the node X has only incoming edges 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. We sometimes write C instead of $c(C)$. 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 the ith bit represents the result of evaluating the i^{th} branch condition in the trace (1 if the condition holds when executing the `true`-edge or 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 $\mathit{lh} * \ell$ times, each time *lh* is contained. The number of times an edge e is taken in a path is given by its execution frequency $\mathit{freq}(e)$, where $\mathit{freq} : E \to \mathbb{N}$.

Implicit Path Enumeration Technique - IPET. We use the IPET approach of (Puschner and Schedl, 1997); a similar method is also discussed in (Li, Y.-T. S. and Malik, S. and Wolfe, A., 1995). Following (Puschner and Schedl, 1997), the IPET technique first translates the CFG of a program into an Integer Linear Program (ILP) problem. The ILP solution corresponds to the trace with the highest edge-weight. For doing so, the following constraints on the CFG are used: (i) the program is entered and exited once, i.e. $\sum \mathit{out}(S) = \sum \mathit{inc}(X) = 1$; (ii) the execution frequency of incoming edges is equal to the execution frequency of outgoing edges, i.e. $\sum \mathit{in}(n) = \sum \mathit{out}(n)$; (iii) for each loop, the loop body is executed ℓ times the loop header, i.e. $\sum \mathit{out}(lb) \leq \ell * \sum \mathit{in}(lh)$. All the frequency variables must be assigned values greater than zero and each edge e is assigned an edge-weight c_e, i.e. the execution time for the edge. The maximum solution to the above system of ILP constraints corresponds to the WCET estimate for the program, WCET $= \mathit{max} \sum_{e \in E} \mathit{freq}(e) * c_e$. In the following, we will omit edge-weights when listing ILP problems. Note that the ILP solution fixes the execution frequencies of program blocks, resulting in an *induced ILP branching behavior*, or simply just *ILP branching behavior*, that encodes one or more execution traces in the CFG. The execution traces resulting from the ILP branching behavior are called *WCET trace candidates*, or simply *WCET candidates*. If a WCET trace candidate is a feasible program execution trace, then it exhibits 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.

Consider 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 $\{(f^1 t^2 t^3 \ldots t^{\ell-1}), (t^1 f^1 t^2 \ldots t^{\ell-1}), \ldots, (t^1 t^2 t^3 ... t^{\ell-1} f^1)\}$, 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 the resulting ILP can be formulated over program blocks, edges or a combination of blocks and edges. Our ILP formulation is inspired by the employed WCET analyzer.

Symbolic Execution. A symbolic execution engine, e.g. (Biere et al., 2013), models program executions by using symbolic values instead of concrete values. This allows to execute a program on symbolic instead of concrete input data. In absence of concrete values, symbolic execution forks on branching points, following only one branch at a time, and assumes the proposed evaluation of the respective path condition. Executing a path condition thus constrains the set of concrete values for a symbolic value, as does executing program statements. For example, symbolic execution of an assignment statement constrains possible values of the left hand side of the assignment to possible values of the right hand side expression.

Runtime evaluation of conditions can be simulated on the symbolic representation: conjoining the set of path conditions assumed on a path together with the symbolic program state (i.e. symbolic values for variables) allows to check feasibility of the path. Deciding the satisfiability of these formulas can be formulated as a satisfiability modulo theory (SMT) problem and hence solved by SMT solvers – see e.g. (Brummayer and Biere, 2009).

Program verification tasks, such as proving that an assertion cannot fail, need guarantees that the property holds on any path. As the number of paths through a program increases exponentially with the number of conditionals in the program, symbolic execution of the whole program space is often an infeasible task, especially in the presence of dynamic data structures and/or unbounded loops. To overcome the computationally expensive parts of symbolic execution, we use path-wise symbolic execution. The same notions of path, execution trace, branching behavior and path expression defined for CFGs also apply to path-wise symbolic execution: a sequence of CFG branching decisions at the same time encodes a symbolic execution trace. In path-wise symbolic execution, we model symbolic program states in the theories of bit-vectors and arrays and use SMT solving to decide satisfiability of path- and loop-conditions. Our SMT representation models addresses and values as bit-vectors, program memory is represented as an array of bit-vectors. Unless a path is specified to be symbolically executed by supplying a branching behavior, paths are exhaustively extracted and executed one after the other.

3. Illustrative Example

In this section we highlight the main steps of WCET Squeezing on an illustrating example.

Consider the C program listed in Figure 1. Note that the initialization of the array element `a[0]` to 1 in Figure 1 enforces the `if`-condition to evaluate to `false` in the first iteration of the loop. Therefore, a precise WCET bound of the program must not include the costs of executing `expensive()` in the first loop iteration.

The computation of a WCET estimate for the program of Figure 1 proceeds in two steps: an architecture-independent high-level WCET analysis inferring so-called flow facts about the program, and a subsequent architecture-aware low level analysis of the program. Typical flow facts are loop bounds, information on infeasible paths or other constraints on (in)valid executions of the program. We next overview these two steps of WCET analysis in our work.

```
int n = 10;
int main ()
{
  int i, a[10];
  a[0] = 1;

  for (i = 0;
       i < n; i = i * 2 + 1)
    if (a[i] != 1)
      expensive();
    else
      cheap();
}
```

Figure 1. An illustrating C program.

High-level WCET analysis. A high-level WCET analyzer performs the following steps on the program source of Figure 1. After parsing the source, its control flow graph representation is constructed, as shown in Figure 2. To ease redability, CFG nodes list program statements. We denote the static initialization of `n` as `enter` and the (implicit) return statement as `exit`. For each node corresponding to a program block, we also list the execution time of the program block. For example, $< 0 >$ denotes that the `enter` node has execution time of 0 unit.

After the CFG construction, an interval and points-to dataflow analysis is performed, inferring value ranges and points-to relations for variables at each program location. Intervals for variables allow to effectively bound loops with concrete bounds, while points-to relations guarantee that certain variables, for example loop iteration variables, are not modified by pointer accesses. As there are no pointer expressions in the program of Figure 1, the results of the points-to analysis can be neglected for the further WCET analysis. Interval analysis infers the value 10 for the variable `n` at the beginning of the loop.

Next, the number of loop iterations, that is the loop bound, of the program of Figure 1 is derived. To this end, we model the loop by a system of algebraic recurrences describing the relation between program variables values at arbitrary loop iterations and use pattern-based recurrence solving as described in (Knoop et al., 2011) to compute a tight upper bound on the number of loop iterations. As a result, we derive the exact loop bound of 4 for the program of Figure 1. This loop bound is illustrated by the edge annotated with **4x** in Figure 2, describing that the loop-body is executed 4 times.

Low-level WCET analysis. The second step of the WCET analysis is to perform the low-level analysis of all program instructions. This step depends on the target architecture, therefore it is a necessity to perform the low-level analysis on the compiled program on the target processor. In our work, we use a modified C167 port of the WCET-aware compiler GCC 2.7.2 with the following features: optimizations violating flow fact annotations are not allowed and flow fact annotations are updated to be valid for the optimized assembly code. Block execution times can then be computed for each basic block, summing up the execution times of the instruction in each block. In complex architectures, features like data and instruction cache configuration and pipeline layout need to be considered in this step.

For the sake of simplicity, in this article we use fixed execution times for each program statement on the source level. For the program of Figure 1, let us assume that the static initialization of `n` to 10 takes 0 time unit, as this initialization takes place before the `main` function of the program is actually executed. Further, we assume that the execution times of all basic program blocks and expressions, i.e. assignments and test conditions, take 1 time unit. The call to `expensive`, respectively `cheap`, is assumed to cost 10, respectively 2 time units.

Initial WCET bound computation. After inferring the execution times of each program block, we next derive the WCET estimate of the program of Figure 1. We rely on the IPET approach and compute a WCET bound by solving the ILP problem encoding the longest program path of Figure 1.

The ILP problem obtained from applying IPET on the CFG of the program of Figure 1 shown in Figure 2 is given in Figure 3(a). We discuss Figure 3 in some detail, by relating it to the CFG of Figure 2. We denote the initialization of array element `a[0]` as `init_a0`, initialization of loop iteration variable `i` as `init_i`, and the loop condition `i < n` as `lCnd`. We write `¬lCnd` to denote the negation of the loop condition. Further, the `then`-branch of the `if`-statement is denoted as `true`, the `else`-branch of the `if`-statement as `false`, and the loop update expression incrementing the loop variable i as `incr`.

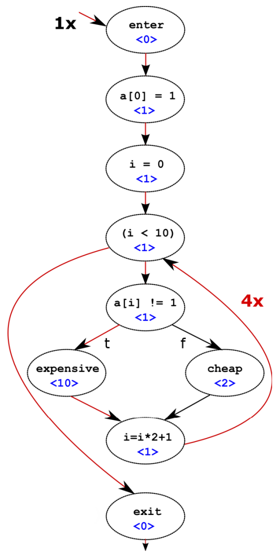
Supplying this ILP problem together with the cost function that maximizes the sum of block execution times their frequencies, the block execution frequencies for each block are derived. In our example, the ILP solution fixes the execution frequencies of program blocks as given in Figure 3(b). For solving the ILP problem, we use the ILP solver `lp_solve` of (Berkelaar et al., 2004). From this ILP solution, by summing the execution frequency of each program block multiplied by the corresponding execution time, the WCET estimate of the program is:

$$\texttt{WCET} = 1*0+1*1+1*1+4*1+4*1+4*10+0*2+4*1+1*0 = 55 \text{ time units.}$$

WCET Squeezing for inferring a proven precise WCET bound. Using the computed initial WCET bound, WCET Squeezing is next applied to tighten this bound and prove it precise. As a first step of WCET Squeezing, execution traces in the CFG need to be computed from the ILP solution from which the (initial) WCET bound was derived. This is done by using the original ILP problem and mapping back the ILP solution to ILP induced branching behavior (see Section 2).

The ILP solution of Figure 3(b) fixes the execution frequency of the `then`-branch of the conditional inside the loop to 4. As the loop bound is also 4, the ILP branching behavior is derived as the sequence $\texttt{t}^1\texttt{t}^2\texttt{t}^3\texttt{t}^4$, specifying that the `true`-edge of the CFG was taken at each loop iteration of Figure 1. For the sake of simplicity, in the sequel we write `tttt` instead of $\texttt{t}^1\texttt{t}^2\texttt{t}^3\texttt{t}^4$.

Next, symbolic execution is performed to check the feasibility of the WCET trace candidate corresponing to the ILP branching behavior `tttt`. To this end, we symbolically execute the concrete program execution trace corresponding to `tttt`, that is at each path-condition the condition is assumed to evaluate to *true*. As a result, symbolic execution infers infeasibility of the



```
enter = 1;
init_a0 ≤ enter;
init_i ≤ init_a0;
lCnd ≤ 4 * init_i;
lCnd + ¬lCnd ≤ incr + init_i;
true + false ≤ lCnd;
incr ≤ true + false;
exit ≤ ¬lCnd;
```
(a)

```
enter = 1, init_a0 = 1,
init_i = 1, lCnd = 4,
true = 4, false = 0,
incr = 4, ¬lCnd = 1
exit = 1
```
(b)

Figure 2. CFG representation of the program.

Figure 3. ILP formulation of Figure 2, (a), and its ILP solution, (b).

WCET trace candidate `tttt` and, even more, it concludes that `true`-edge of the CFG cannot be taken in the first iteration of the loop.

Hence, a counterexample to the high-level precision of the computed WCET bound of 55 is derived. The infeasible WCET trace candidate `tttt` is further used to refine the set of feasible WCET trace candidates, ultimately yielding a tighter WCET bound than the initially computed WCET bound of 55. To exclude the infeasible trace `tttt`, we add additional constraints to the initial ILP problem of Figure 3, as shown in Figure 4 and explained further.

Loop iterations up to the unsatisfiable path-condition are peeled, introducing additional ILP variables and expressions, as shown in the CFG of Figure 5. Loop peeling means to peel off the first iteration of a loop, replacing the first iterations loop condition by a conditional statement that guards a copy of the loop body (Muchnick, Steven S., 1997). The peeled iteration is executed before the original loop, whose loop bound is decreased (the ILP solution specifies a loop bound). As the `true`-edge of the conditional statement of Figure 1 in the first loop iteration is infeasible, the first iteration of the loop is peeled and added to the new ILP problem of Figure 4: the ILP variable `lCnd_P` models the loop condition, `cond_ai_P` the `if`-condition, `true_P` the `true`-edge and `false_P` the `false`-edge of the `if`-statement in the peeled first loop iteration. Note that, due to peeling, the loop bound is reduced by 1. Excluding the infeasible WCET trace candidate is then guaranteed by an additional ILP constraint that restricts the execution frequency of the `then`-branch of the `if`-statement inside the loop to be at most 3. The solution of the new ILP problem is given in Figure 6:

The WCET bound resulting from Figure 6 is then computed as:

$$\texttt{WCET} = 1 + 1 + \mathbf{1} + \mathbf{1} + \mathbf{2} + \mathbf{1} + \mathbf{3} * \mathbf{1} + \mathbf{3} * \mathbf{1} + 3 * 10 + 0 * 2 + 3 * 1 + 0 = \mathbf{46} \text{ time units},$$

which clearly improves the initial WCET bound of 55.

In the next iteration of WCET Squeezing, the ILP induced branching behavior is extracted again from the ILP solution of Figure 6, yielding the WCET trace candidate of `fttt` exhibiting

```
enter = 1
init_a0 ≤ enter
init_i ≤ init_a0

/* peeled loop iteration */
lCnd_P ≤ init_i
true_P + false_P ≤ lCnd_P
incr_P ≤ true_P + false_P
lCnd ≤ 3 * incr_P
lCnd + ¬lCnd ≤ incr * incr_P
true + false ≤ lCnd
incr ≤ true + false
exit ≤ ¬lCnd

/* clause excluding the infeasible WCET trace
candidate */
true_P + true ≤ 3
```

Figure 4. Refined ILP problem of Figure 1, by peeling the first loop iteration and excluding the infeasible WCET trace candidate `tttt`.

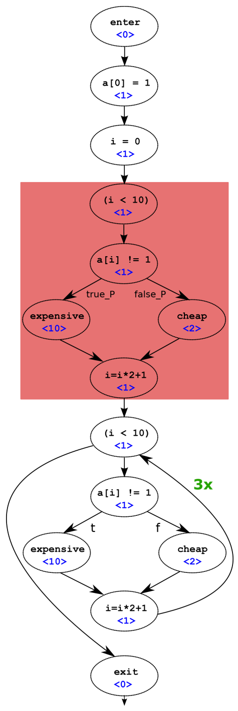


Figure 5. Refined CFG representation of Figure 1, with the the peeled first loop iteration.

```
enter = init_a0 = 1,
init_i = 1, lCnd_P = 1,
true_P = 0, false_P = 1
incr_P = 1,
lCnd = 4,
true = 3, false = 0,
incr = 3, ¬lCnd = exit = 1
```

Figure 6. A valid ILP solution that encodes a feasible program execution.

the WCET bound of 46. The ILP branching behavior is then symbolically executed and its feasibility is inferred. Therefore, WCET Squeezing terminates and reports that the WCET bound 46 is a precise bound of the running example of Figure 1.

4. WCET Squeezing: Replacing Conjectures by Positive Knowledge

We now describe WCET Squeezing in detail for proving WCET bounds precise. Given a program, WCET Squeezing solves the *classical WCET problem* by closing the precision gap between the actual WCET of the program and its usually over-approximated WCET computed by existing WCET analyzers. This way, WCET Squeezing does not only derive a precise WCET bound but also returns a witness, i.e. a feasible program execution exhibiting the WCET bound.

In a nutshell, WCET Squeezing applies on-demand WCET feasibility refinement, as follows. It takes as input the result of an a priori WCET analysis of the program and tightens, that is

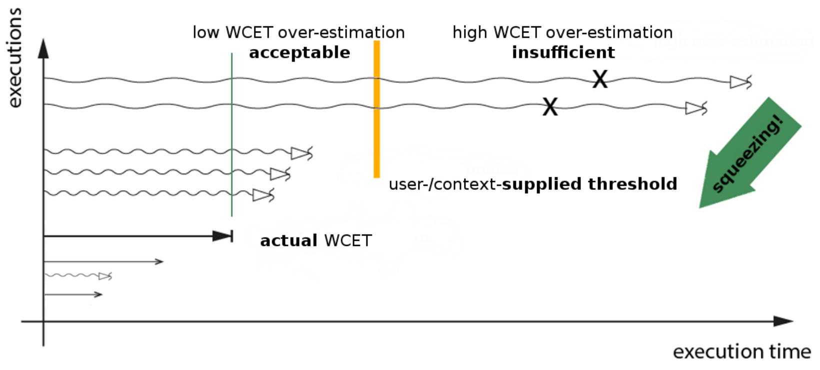


Figure 7. Straight arrows represent concrete executions, assuming some hardware-model. Waved arrows are infeasible executions with over-estimated WCET.

squeezes, this WCET estimate, by refining the program model. To this end, WCET Squeezing combines the IPET approach with symbolic execution in a novel way. To squeeze the computed WCET bound of program, the results of the IPET analysis are mapped back to a trace that can be symbolically executed 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 ILP problem, is extended by new constraints excluding the infeasible trace. The new ILP problem is then solved, resulting in a tighter WCET bound and a new program execution trace exhibiting the tigthened WCET bound. This execution trace is used in the next iteration of WCET Squeezing. Upon termination, the initial WCET bound is tightened and 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, i.e. any over- estimation due to infeasible branching decisions is eliminated.

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, as illustrated in Figure 7: infeasible executions with a high WCET (i.e. the crossed waved arrows) are ruled out by symbolic execution until the WCET estimate is below a required threshold (i.e. the waved arrows between the actual WCET and the supplied threshold) or the given time budget is exhausted or until a feasible execution is found. Usually, there is no need for any of these variants to fully symbolically execute the whole program.

Moreover, the computational effort of WCET Squeezing can be controlled by the user: WCET Squeezing can be run repeatedly until some time budget or threshold value is exhausted, improving the WCET bound in each iteration. In the case when WCET Squeezing terminates reporting a feasible program execution whose WCET is above the required threshold, WCET Squeezing proves that the program cannot meet the constraints imposed by the current threshold. WCET Squeezing hence also solves the *pragmatic WCET problem*, proving that a program meets some given timing constraints. Thus, WCET Squeezing can be used in the following three application scenarios of WCET analysis:

(1) *Precision-controlled WCET Squeezing* runs until a feasible trace is found. Since the feasible trace exhibits the actual WCET of the program, WCET Squeezing proves the precision of the derived WCET bound and solves the classical WCET problem.

(2) *Budget-controlled WCET Squeezing* tightens the WCET bound until an a priori specified running time budget is exhausted, solving thus a variant of the pragmatic WCET problem. While the WCET bound computed in this case is usually still improved over the initial one, it is usually not tight: the program trace exhibiting the WCET bound might still be proven infeasible when the running time budget is enlarged.

(3) *Limit-controlled WCET Squeezing* aims at proving that the program is fast enough, that is, its WCET is below some a priori given limit, solving another variant of the pragmatic WCET problem. In this case, one of 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.

The above detailed features of our method turn WCET Squeezing into a non-conventional new approach in WCET analysis, which is, by design, out-of-scope of 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 lead to better performance of WCET Squeezing. If a static WCET analyzer supplies tight initial bounds, a proof of precision can be obtained fast and the computational effort for WCET Squeezing is minimized. 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 minimize the over-estimation of WCET bounds.

In the rest of this section, we formally introduce our WCET Squeezing algorithm (Section 4.1). Next, we detail the main ingredients of WCET Squeezing: selection and refinement of WCET trace candidates (Section 4.2), selective symbolic execution (Section 4.3), and ILP constraint encoding and refinement (Section 4.4).

4.1. The WCET Squeezing Algorithm

The WCET Squeezing algorithm, presented in Algorithm 1, iteratively refines the WCET estimate of programs with reducible control flow. Reducibility restricts the allowed control flow, for example by excluding constructs like multi-entry loops. 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. Optional parameters can also be supplied to guarantee termination of WCET Squeezing within a certain time-limit `time` or below a WCET threshold `BL_value`. If during the supplied time-limit a WCET trace candidate is excluded by WCET Squeezing, the WCET bound of the program is improved unless another WCET trace candidate exhibits the same initial WCET bound (*Budget-controlled WCET Squeezing*). Alternatively, when running WCET Squeezing with a pre-defined *threshold*-value, our method answers whether an a priori fixed WCET limit can be met by the program: WCET Squeezing is run until the improvement in the WCET bound reaches the required value, reporting *yes* in this case, or until it terminates due to a feasible WCET trace candidate, reporting *no* (*Limit-controlled WCET Squeezing*).

Note however that WCET Squeezing is always guaranteed to terminate, even without using a pre-defined time-limit and/or threshold. 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 bound is improved at every iteration of Algorithm 1, and hence the WCET bound reported upon the termination of Algorithm 1 is improved compared to the input of Algorithm 1.

Algorithm 1. The WCET Squeezing Algorithm
Input: ILP problem `ilp_problem`, Program p
Output: ILP solution `ilp_solution`
Optional Input Parameter: time-budget `time` or WCET threshold limit `BL_value`

```
1  begin
2  do
3    $ilp\_solution := ILPsolve(ilp\_problem)$
4    $wcet\_candidates := extractCandidates(p, ilp\_problem, ilp\_solution)$
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 (Berkelaar et al., 2004). Based on the computed ILP solution, the corresponding ILP branching behavior is mapped back to the CFG of the program `p` and WCET trace candidates are extracted (line 4), as detailed in Section 4.2. These WCET trace candidates are next symbolically executed (line 5), as presented in Section 4.3. 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 the WCET bound estimate of the program under study. If all WCET trace candidates are infeasible, the ILP branching behavior is infeasible as well. The WCET bound 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 4.4. A further iteration of WCET Squeezing is applied on the new ILP problem, yielding tighter WCET bound estimates of the program (line 3). Algorithm 1 for WCET Squeezing terminates when a tightened and precise WCET bound is derived (line 6).

4.2. WCET Trace Candidates

To construct WCET trace candidates (line 4 of Algorithm 1), a mapping from the ILP branching behavior to program execution traces is needed. 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. Note that the ILP solution yields bounds for all loops and allows us to peel-off single loop iterations. Peeling-off iterations allows to determine the branching behaviour for single iterations.

From the ILP branching behavior ilp_bb, WCET trace candidates are constructed by specifying their branching behavior bb. Recall that 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 $\mathit{freq}(e)$, no interleaving among branch-conditions is possible. Therefore, a single WCET trace candidate is constructed whose $\mathit{freq}(e)$ positions are set either to `t` or `f`. 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:

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

Example 1. Consider the program given in Figure 8, $\mathit{freq}(t_{C1}) = 1$ and $\mathit{freq}(t_{C2}) = 1$. The ILP branching behavior *ilp_bb* given by `tt` can then only result in a single branching behavior bb, which is `tt`.

```
if (C1) // t_C1
  ...
if (C2) // t_C2
        ...
```

Figure 8. Program whose *ilp_bb* results in a single bb.

```
for (i = 0; i < 4; i++)
  if (C1) // t_C1
    ...
  else // f_C1
    ...
```

Figure 9. Program whose *ilp_bb* can result in multiple bb.

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:

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

That is, a sequence of `t` (respectively, `f`) of length $\mathit{freq}(t_{C_i})$ (respectively, $\mathit{freq}(f_{C_i})$) is set starting from $\mathit{ilp\_bb}[i]$. 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.

Example 2. Consider the program shown in Figure 9, with $\mathit{freq}(t_{C1}) = 4$ and $\mathit{freq}(f_{C1}) = 0$. The execution frequency of t_{C1} is greater than 1 but the execution frequency of f_{C1} is 0. Therefore the *ilp_bb* given by `tttt`, can only result in a single bb, which is `tttt`.

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 $\mathcal{S} = \{ \underbrace{\texttt{t...t}}_{\mathit{freq}(t_{C_i}) \text{ times}} , \underbrace{\texttt{f...f}}_{\mathit{freq}(f_{C_i}) \text{ times}} \}$.

Using the results of (Skiena, 2008), the number of permutations over S, and thus the number of loop branching behaviors is: $p = \frac{\left(\mathit{freq}(t_{C_i})+\mathit{freq}(f_{C_i})\right)!}{(\mathit{freq}(t_{C_i})!) * (\mathit{freq}(f_{C_i})!)}$.

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:

$$bb_x[i+j] = \mathit{permutation}_x\{ \underbrace{\texttt{t},\ldots,\texttt{t}}_{\mathit{freq}(t_{C_l}) \text{ times}} , \underbrace{\texttt{f},\ldots,\texttt{f}}_{\mathit{freq}(f_{C_l}) \text{ times}} \}$$

$$\text{with} \quad 0 \leq j \leq \mathit{freq}(t_{C_i}) + \mathit{freq}(f_{C_i}) \quad \text{and} \quad 0 \leq x \leq p,$$

where $\mathit{permutation}_x\{S\}$ gives the xth permutation over S.

Example 3. Consider again the program of Figure 9, with $\mathit{freq}(t_{C1}) = 2$ and $\mathit{freq}(f_{C1}) = 2$. The execution frequency of both t_{C1} and f_{C_1} is greater than 1, therefore $p = \frac{(2+2)!}{2!*2!} = \frac{24}{4} = 6$ different bb are constructed: `ttff`, `fttf`, `fftt`, `tftf`, `ftft`, `tfft`.

Note that for a branching behavior 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.

4.3. Selective Symbolic Execution

Selective symbolic execution supports the analysis of a computable or measurable property, such as the WCET, of a program under study, while exploring only the relevant parts for analyzing the property. The goal of selective symbolic execution is to minimize the number of symbolic executions required in order to improve the analysis results. Our WCET Squeezing algorithm combines a symbolic execution engine with WCET analysis (line 5 of Algorithm 1) and uses WCET estimates to guide selective symbolic execution by symbolically executing only those traces that might exhibit the WCET bound.

Using the branching behavior bb of the WCET trace candidates, the symbolic execution in 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.

Symbolic execution in WCET Squeezing 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 an SMT representation 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 constructed SMT representation. That is, if the specified evaluation of the path-condition is unsatisfiable at some point, the trace $\pi(bb)$ is infeasible. We conclude that $\pi(bb)$ is infeasible iff one of the involved conditions, C_i, cannot symbolically evaluate to the outcome specified by the ILP solution, $bb[i]$.

If every condition C_i in the branching behavior bb can evaluate to the value specified by $bb[i]$, 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 refinement and WCET tightening is possible (line 6 of Algorithm 1).

Otherwise, if the symbolic execution of a trace candidate fails, some path-condition C_i cannot evaluate to the value $bb[i]$ specified in the ILP solution. The symbolically evaluated conditions can be mapped to the corresponding 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 further detailed in Section 4.4.

4.4. 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 new ILP constraint to the ILP problem.

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

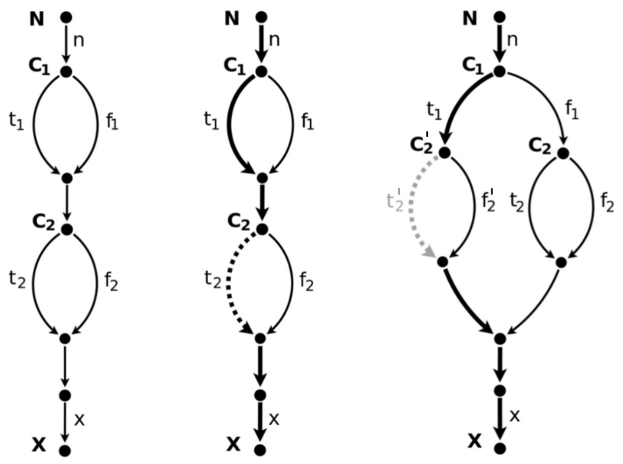
Figure 10. Program with mutually exclusive conditions `C1` and `C2`.

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 constructed ILP constraint involves all conditional-edges corresponding to all C_i that were symbolically evaluated. Using the notation from Section 4.3, recall that $bb[i]$ gives the conditional-edge (`t` or `f`) from the ith position of bb. We denote by $bb[i]_{C_i}$ the conditional edge of C_i in the ILP; for simplicity, the value of $bb[i]_{C_i}$ is going to be denoted by t_i or f_i. Then, the conditional-edges 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 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 4. Consider Figure 10, where conditions C_1 and C_2 are mutually exclusive. The (abstracted) CFG representation of Figure 10 is given in Figure 11(a). The initial ILP solution, given in Figure 11(d), yields a WCET trace candidate with branching behavior `tt`, i.e. an execution frequency that enables the execution of both t_1 and t_2 corresponding to C_1 and C_2 (both with execution frequency 1). However, as the conditions C_1 and C_2 are mutually exclusive, only one of the conditions can be *true*. Therefore, symbolic execution sets C_1 to `true`, executes t_1, infers that the evaluation of C_2 to `true` is unsatisfiable, and thus the execution of t_2 is infeasible.

There are two ways to construct a constraint from the infeasible symbolic execution: the first corresponds to the transformation of the ILP, as listed in Figure 11(e) and illustrated by Figure 11(c). It explicitly excludes the trace by removing it from the abstraction. The transformation duplicates a conditional and removes the infeasible edge, by using the constraint `f2'` $\leq$ `c2'` instead of `f2'` `+` `t2'` $\leq$ `c2'` (for loops, this requires peeling off loop iterations). Alternatively, the constraint can be constructed by specifying in the ILP that either C_1 or C_2 (but not



(a) (b) (c)

```
n = 1
c1 ≤ n
t1 + f1 ≤ c1
c2 ≤ t1 + f1
t2 + f2 ≤ c2
x ≤ c2
```
(d)

```
n = 1
c1 ≤ n
t1 + f1 ≤ c1
c2′ ≤ t1
f2′ ≤ c2′
c2 ≤ f1
t2 + f2 ≤ c2
x ≤ c2′ + c2
```
(e)

Figure 11. (a) CFG representation of Figure 10; (b) Infeasible WCET trace candidate (bold) with an unsatisfiable conditional edge (dotted); (c) Transformed CFG after symbolic execution, with two copies of the conditional-statement C_2 for excluding the infeasible trace of (b); (d) The original ILP of (a); (e) The modified ILP of (c).

both) is valid, thus implicitly excluding the trace from the ILP, by decreasing the combined execution frequency of all executed conditionals edges executed, i.e. of t_1 and t_2.

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 11(a) is executed at most once,[1] that is $n \leq 1$ and $x \leq 1$; (ii) the conditional-edges of Figure 11(a) are executed at most once, that is $t_1 + f_1 \leq n$ and $t_2 + f_2 \leq n$; (iii) due to the new constraint, the combined execution frequency of the `true`-edges of C_1 and C_2 is restricted to 1 (the execution frequency of C_1, that is $t_1 + t_2 \leq 1$.

As illustrated in Example 4, 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.

Let us finally note, that in the presence of loops, 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 ILP constraint is generated: $\mathit{freq}(e) \leq \ell * m$. Our framework handles the ILP refinement of programs with loops by using this ILP constraint when computing the combined execution frequency for edges inside loops. Otherwise, the ILP problem refinement for program loops is performed similarly to the already discussed cases of this section. Therefore, to ease readability, here we do not further address the treatment of loops, but refer the interested reader to (Knoop et al., 2013) for more details.

[1] To ease readability, in Figure 11 we refer to $freq(n)$ as n, and similarly for the other CFG nodes and expressions.

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

Figure 12. `lcdnum.c`, simplified.

Example 5. We conclude this section by highlighting WCET Squeezing on the example of Figure 12, taken from the `lcdnum.c` benchmark of the Mälardalen WCET suite (Gustafsson et al., 2010).

The program of Figure 12 contains 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 bound of 24320 time units (cycles), where the execution frequency of the `true`-edge of the conditional is set to 10. By mapping back this WCET bound 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 and concludes the infeasibility of this trace by symbolic execution. An ILP constraint excluding this infeasible WCET trace candidate is then derived and used in the IPET analysis. The new ILP problem yields the new WCET bound of 23420 and sets the execution frequency of the `true`-edge of the conditional to 9. Note that the new WCET bound tightens the initially computed WCET bound by 3.7%.

In the second iteration of WCET Squeezing, the new WCET bound of 23420 is used and the feasibility of WCET trace candidates are checked. This time, there are multiple (namely 11) WCET trace candidates since the `false`-edge of the conditional can be also taken in a loop iteration. Symbolic execution, however, establishes infeasibility of each WCET trace candidate, and thus a third iteration of WCET Squeezing is applied.

In the third iteration of WCET Squeezing, the execution frequency of the `true`-edge of the conditional is now set to 8 and the new WCET bound of 22520 is derived, yielding a WCET improvement of 7.4% when compared to the initial WCET bound. The number of WCET trace candidates executed by symbolic execution increases in the following iterations of WCET Squeezing. WCET Squeezing terminates in its sixth iteration, where a feasible WCET trace candidate with execution frequency of 5 for the `true`-edge of the conditional is found. This feasible WCET trace candidate yields the precise WCET bound of 19820 cycles, tightening the initial WCET bound by 18.5%.

5. Implementation and Experimental Results

Our overall approach of WCET Squeezing for proving WCET bounds precise is *implemented in the r-TuBound WCET tool chain* (Knoop et al., 2012). In this section we overview our implementation and report on our experimental results obtained by using WCET Squeezing on challenging WCET benchmarks.

Our implementation of WCET Squeezing in r-TuBound is available at:

`www.complang.tuwien.ac.at/jakob/tubound`.

5.1. WCET Squeezing and r-TuBound

r-TuBound uses the general WCET framework of (Prantl et al., 2009), but significantly extends this work by combining traditional WCET analysis with automated techniques for proving WCET bounds precise. In addition, r-TuBound implements the loop bound computation technique of (Knoop et al., 2011) for computing tight loop bounds. The main steps of r-TuBound are illustrated in Figure 13 and described below.

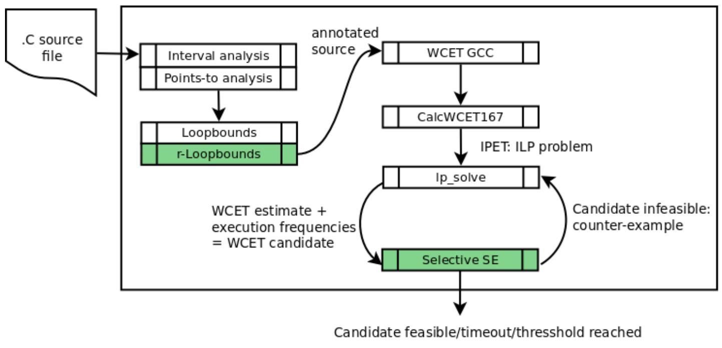


Figure 13. Overall workflow of WCET Squeezing in r-TuBound.

Overview of r-TuBound. Our implementation in r-TuBound takes as input a C program and returns either a proven precise WCET bound of this program or a WCET bound that fulfills an a priori given time/threshold constraint. For doing so, the C program source is first parsed by r-TuBound using the Edison Design Group (EDG) C/C++ frontend (Edison Design Group, 1992). The CFG representation of the program is next transformed by means of the Static Analysis Tool Integration Engine – SATIrE framework of (Schordan et al., 2007). To this end, r-TuBound reuses a number of C/C++ source-to-source transformers based on the ROSE compiler framework (LLNL, 2000). It also uses the static analysis libraries of SATIrE and TERMITE (Prantl, 2008) for implementing a forward-directed data flow interval analysis, a points-to analysis, and a simple constraint-based analysis for loop bound computation (Prantl et al., 2009). However, r-TuBound extends (Prantl et al., 2009) by using pattern-based recurrence solving in conjunction with SMT solving and deriving loop bounds for a more general class of programs than the ones handled by (Prantl et al., 2009).

Further, r-TuBound relies on a WCET-aware C compiler, based on the GNU C compiler 2.7.2.1 ported to the Infineon C167CR architecture with added WCET analysis functionality, and the CalcWCET167 low level WCET analysis tool (Kirner, 2012) which supports the Infineon C167CR as target processor and applies IPET to generate the ILP problems from which WCET bounds can be derived. Finally, r-TuBound applies WCET Squeezing to tighten and prove precision of the (initial) WCET bound, returning a proven precise WCET bound for its input program.

Loop bound computation in r-TuBound. For computing loop bounds, r-TuBound implements the following steps: rewriting while-loops into equivalent for-loops, rewriting if-statements into if-else statements, translating multi-path loops with abrupt termination into loops without abrupt termination, approximating non-deterministic variable assignments, and pattern-based recurrence solving for computing tight loop bounds. More details on the loop bound computation step of r-TuBound can be found in (Knoop et al., 2011, 2012).

Initial WCET bound computation. After the loop bound computation step, a WCET bound is derived by r-TuBound using the CalcWCET167 low level analyzer. Running CalcWCET167 on the annotated assembly of a function produces an IPET problem specification formulated as an ILP problem. In r-TuBound we use the ILP solver `lp_solve` (Berkelaar et al., 2004) to compute ILP solutions from which an initial WCET bound and execution frequencies of program blocks are derived.

WCET Squeezing. WCET Squeezing is next applied to prove or disprove the initially inferred WCET bound precise: the ILP problem specification and solution and the annotated assembly are supplied to a modified version of CalcWCET167. The modified CalcWCET167 then extracts the branching behavior from the assembly and the ILP. The branching behavior is used to construct WCET candidates.

The symbolic execution engine of r-TuBound, which is used in WCET Squeezing for proving precise WCET bounds, is implemented in the standalone symbolic engine SmacC (Biere et al., 2013). SmacC applies path-wise symbolic execution of a supplied C program that lies in the supported subset of (ANSI) C and uses the Boolector SMT solver (Brummayer and Biere, 2009) to check feasibility of WCET trace candidates in the quantifier-free logic of bit-vectors with arrays.

5.2. Experimental Results

We evaluated our WCET Squeezing implementation in r-TuBound on a number of challenging examples taken from the Mälardalen WCET benchmark suite of the WCET community (Gustafsson et al., 2010).

In our experiments, we used those examples from the Mälardalen WCET benchmark suite that can be handled by the symbolic execution engine. The symbolic execution engine does not support full ANSI C (for example, there is no support for floating point operations or structs), therefore the selection of benchmarks was chosen accordingly. Additionally, we selected benchmarks where WCET analysis came up with precise bounds (i.e. WCET Squeezing proved their precision without tightening them further) and benchmarks where the WCET estimate could be improved by WCET Squeezing.

Our results are summarized in Table 1. Column 1 (BM) of Table 1 lists the benchmark's name and Column 2 (function) lists the analyzed functions in the benchmark. Column 3 (WCET) shows the initial WCET estimate and Column 4 (squeezed) the bound obtained after running WCET Squeezing. Column 5 (#iters) lists how many iterations of WCET Squeezing were performed and Column 6 (#π) shows the number of excluded execution traces. Columns 7 (%imp) and 8 report on the obtained WCET improvements, as follows: %imp describes the achieved improvement and Column 8 (%prec) the maximum improvement, i.e., the improvement necessary for a precise estimate. Column 9 (note), finally, states whether precision could be achieved and proved.

The initially computed WCET trace candidate is feasible for the functions `prime`, `cl_block`, and `icrc1`, hence the initial bound is proved to be precise by r-TuBound, which was unknown before. No further improvements in the WCET bounds are possible for these examples. For the functions `adpcm`, `duff`, `expint`, and `fibcall` continuous improvement is achieved until WCET Squeezing terminates, resulting in a precise squeezed WCET bound. For the functions `expint` and `janne_complex`, both can be improved by more than 90%, we report the impact of a single iteration of WCET Squeezing to demonstrate the different effects 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 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. Since WCET Squeezing was not run until termination for the benchmark programs in the lower part of the table, columns %imp and %prec differ for those benchmarks (the actual, feasible, WCET path exhibiting a reduced estimate of %prec is easy to find for those examples). This illustrates the often dramatic impact of even only running a few iterations of WCET Squeezing.

BM	**function**	**WCET**	**squeezed**	**#iters**	**#π**	**%imp**	**%prec**	**note**
prime	prime	784860	784860	1	0	0	0	PPO
compress	cl_block	8440	8440	1	0	0	0	PPO
crc	icrc1	18060	18060	1	0	0	0	PPO
adpcm	logsch	5560	5380	1	1	3.24	3.24	PPW
	uppol2	12260	12040	2	2	9.87	9.87	PPW
duff	duffcopy	85940	79949	5	5	7.5	7.5	PPW
fibcall	fib	79840	75040	2	2	6.39	6.39	PPW
expint	expint	1.24E7	1.22E7	1	1	0.94	93	I
janne-cmp	complex	694980	653380	1	1	5.9	93.3	I
lcdnum	main	24320	22520	2	11	7.4	18.5	I
nsichneu	main	4.95E6	4.94E6	2	2	0.28	-	*I

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. PPO stands for "precision proved without tightening", PPW denotes "precision proved with tightening", "I" denotes an imprecise WCET bound and "*" denotes that the actual WCET is unknown.

Summarizing, our experiments show that WCET Squeezing can effectively prove the precision of WCET bounds, and significantly tighten the WCET bounds computed by other WCET techniques. It is for example not unusual, that in a single iteration of WCET Squeezing, the precision of WCET bound is improved by 3%!

6. Discussion

In this section we discuss some limitations of WCET Squeezing and point out further applications of WCET Squeezing to WCET analysis.

As noted, the precision of WCET Squeezing depends on the underlying hardware model. We are currently investigating the use of WCET Squeezing for improving the precision of dynamic, that is measurement-based, WCET analysis. Dynamic WCET analyzers run the program on the real hardware and measure hardware-dependent execution times, but cannot identify the actual WCET execution of the program. Nevertheless, by encoding feasibility of program traces as satisfiability questions in SMT solving, one could enforce program executions along feasible WCET trace candidates during dynamic WCET analysis. This way the derived program execution times by a dynamic analyzer will be more precise and closer to the actual WCET of the program. WCET Squeezing can thus help to improve the precision of dynamic WCET analysis.

We are also interested in extending WCET Squeezing with recent results in refining cache analysis, either by means of program verification (Chattopadhyay and Roychoudhury, 2011) or path-sensitive symbolic execution (Chattopadhyay and Roychoudhury, 2013). Cache analysis typically classifies memory accesses into cache-hits (the memory access will always successfully hit the cache), cache-misses (the memory access will always fail to hit the cache and needs to access memory, which is costly), and unclassified accesses that are assumed to be cache-misses during WCET analysis. Refining unclassified memory accesses during cache analysis is therefore of critical importance for WCET analysis. Existing approaches to cache analysis refinement annotate the source code with additional assertions describing numerical properties over the memory and are in particular useful for analyzing memory accesses inside program loops.

Validity of the introduced assertions can, for example, be checked based on model checking or symbolic execution. We are interested in using such a cache analysis refinement in the context of WCET Squeezing, and apply the path feasibility analysis of WCET Squeezing to refine unclassified memory accesses only on feasible program paths, improving thus the precision and performance of cache analysis.

On the implementation side, WCET Squeezing in r-TuBound currently relies on a manually constructed mapping between the assembly and the source of the program. In the future we are interested to verify the feasibility of ILP solutions by performing symbolic execution on the assembly. This way, arbitrary compiler optimizations performed before low-level WCET analysis will naturally be respected in the symbolic execution step of WCET Squeezing. As various hardware features can be modelled on the assembly but not on the source level, SMT reasoning in the quantifier-free theories of bit-vectors and arrays could also turn out to be more efficient on the assembly code than on the source code. A natural disadvantage of running symbolic execution on the assembly code is that symbolic execution becomes platform-dependent and hence less generic. However, current approaches, such as (Cassé et al., 2013), address partially this problem, for example by using a low level virtual machine intermediate representation (Lattner and Adve, 2004).

7. Related Work

To the best of our knowledge, our WCET Squeezing method is the first approach which tightens and improves the WCET bound of a program after an initial WCET analysis in an anytime-manner. WCET Squeezing makes use of both symbolic execution and WCET analysis, and is different from other existing approaches for the following reasons.

Symbolic execution originally was used for test-case generation but recently found more and more applications in program verification or automatic exploit generation for applications. Symbolic execution has been successfully applied for test-case generation for programs in (Cadar et al., 2008) and (Cadar et al., 2006), where, for example, the work of (Cadar et al., 2006) describes a symbolic execution engine for bug-hunting. In these approaches the focus lies on speeding up symbolic execution by optimization and caching of queries discharged to the underlying solver or tracking only currently relevant information. In contrast, symbolic execution in WCET Squeezing offers little optimizations and tracks as much information as possible. In general, precise information needs high run-times, a problem we counteract by only applying selective symbolic execution.

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. (Knoop et al., 2012; Ballabriga et al., 2010; Gustafsson, 2013), typically rely on the IPET technique (Puschner and Schedl, 1997) to calculate a WCET estimate. IPET usually overestimates the WCET bound, as the constructed ILP problem encodes numerous spurious program executions that are infeasible in the program. WCET Squeezing is used in addition to IPET-based WCET analysis, overcoming this deficiency.

The refinement loop of WCET Squeezing approach has similarities with the counterexample guided abstraction refinement (CEGAR) method of (Clarke et al., 2000). WCET Squeezing is an anytime-algorithm (Boddy, 1991) that allows to refine or prove precise a WCET estimate after an initial IPET-based WCET analysis. A related idea is presented in (Bang et al., 2007), where an ILP encoding of the program is used to check whether partial solutions of a specific size to the ILP problem yield infeasible program paths. Feasibility of solutions in (Bang et al., 2007) is checked using model checking. In contrast to our approach of path-wise symbolic execution, (Bang et al., 2007) encodes the feasibility check as program assertions in the original program, thereby losing the advantage of path-local reasoning.

In general, inferring precise program information comes with high computational costs, a problem which we avoid by using selective symbolic execution: WCET Squeezing applies symbolic execution only when information about the program is too coarse or when other analysis methods fail. A similar idea is presented in (Bodík et al., 1997) where symbolic execution is used to refine spurious def-use results via a path feasibility analysis. In (Bodík et al., 1997) branching decisions are determined at compile time and used to identify and remove infeasible paths. This method can be seen as a light-weight on-demand symbolic execution of conditional nodes, whereas symbolic execution in WCET Squeezing always executes single paths.

Symbolic execution for WCET analysis is also used in (Kebbal and Sainrat, 2006) and avoids some typical pitfalls of symbolic execution. For example, loops are not unfolded and hence multiple executions of the same block are omitted. We note that (Kebbal and Sainrat, 2006) analyses each program block whereas our selective symbolic execution approach only analyses relevant program blocks and paths.

Measurement-based timing analysis techniques can be seen complementary to static WCET analysis tools. Measurement-based tools require test inputs that cover a sufficient portion of the program executions to infer results with high confidence. In (Zolda and Kirner, 2011) test data is systematically generated test using model checking and additional heuristics . In contrast to this technique that generates test-cases for arbitrary executions of the program, WCET Squeezing generates test-cases that lead to executions along the WCET trace candidate(s).

A different approach to WCET analysis is described in the quantitative abstraction refinement work of (Cerny et al., 2013), which relies on segment- and state-based abstract interpretation (Cousot and Cousot, 1977). The state-based approach has some similarities to counterexample encoding in WCET Squeezing, which makes control-flow decisions explicit in loop iterations. Related, though conceptually different is the approach of (Brandner and Jordan, 2014). 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 bound of the program.

Eliminating the irrelevant parts from the program allows usage of a more precise but computationally more expensive WCET analyzer that then might come up with a more precise WCET bound. Another related approach is the abstract execution framework of (Gustafsson, 2001), where context-sensitive abstract interpretation is applied to analyse loop iterations and function calls in separation. Instead of applying a fix-point analysis, abstract operations on abstract values are used in (Gustafsson, 2001), where an abstract value can, e.g. , be represented as an interval. When abstract values prevent the evaluation of a conditional, both branches need to be followed. Abstract states can be merged at join points to prevent the path explosion problem. As a result, a single abstract execution can represent the execution of multiple concrete paths. This is not the case in the traditional use of symbolic execution. Compared to WCET Squeezing, abstract execution in (Gustafsson, 2001) analyzes the entire program, whereas our method applies more costly symbolic execution only to relevant parts of the program.

8. Conclusion

We presented WCET Squeezing, a new technique that allows to replace conjectures, that is WCET bounds, inferred by a typically imprecise WCET analysis by positive knowledge in the form of a proof of precision of derived (and typically tightened) WCET bounds.

Precision. We described a decision procedure that is able to prove whether the WCET bound inferred by a static WCET analyzer is precise, and if not, to tighten the WCET bound until eventually the bound is proved to be precise. Our procedure, called WCET Squeezing, iteratively refines the program model by excluding infeasible program paths until the computed WCET bound of the refined model is proven precise.

WCET Squeezing combines symbolic execution together with the implicit path enumeration technique, and uses SMT reasoning together with integer linear programming techniques for computing precise WCET bounds. Symbolic execution offers a precise framework for program analysis and tracks complex program properties by analyzing single program paths in isolation. This path-wise program exploration of symbolic execution is, however, computationally expensive, which often prevents full symbolic analysis of larger applications: the number of paths in a program increases exponentially with the number of conditionals, a situation denoted as the path explosion problem. Therefore, for applying symbolic execution for WCET Squeezing, we used WCET analysis as a guidance for symbolic execution allowing us to focus symbolic execution to a set of relevant program paths thereby effectively avoiding full symbolic coverage of the program. The resulting symbolic execution framework is thus selective in the sense that it explores only those program paths that (possibly) exhibit the WCET bound of the program. Selective symbolic execution in WCET Squeezing carefully balances the precision and computational cost of symbolic execution making WCET Squeezing efficient in practice.

Such a balance between precision and efficiency is also present in other approaches to the WCET analysis of programs: a successful WCET analyzer usually requires a balance between the speed and the precision of the deployed analysis. Precision of the analysis is gained by applying powerful program analysis techniques that gather information about the program and pass it to further analysis and computation steps. Precision of the analysis enables the computation of tight WCET bounds, however, at the expense of high computational costs; this sometimes prevents the analysis to terminate within a given time-limit. Therefore, precision of the analysis is often traded for its speed, that is faster analysis at the cost of typically less precise WCET bounds instead of more precise ones but less efficient analyses. Automated methods for improving imprecise WCET estimates into tight WCET bounds are therefore crucial in making WCET analysis practically useful.

We demonstrated that combining selective symbolic execution with traditional WCET analysis in WCET Squeezing is such an automated method for deriving precise WCET bounds. We showed that, when using selective symbolic execution in WCET analysis, a partial symbolic coverage of the program is sufficient to tighten and, eventually, prove the WCET bound of the program precise. Selective symbolic execution comes thus with the advantage of avoiding the path explosion problem of traditional symbolic execution, as it applies costly symbolic execution only for those program parts that influence the WCET bounds.

Experiments. We implemented WCET Squeezing in the fully automatic WCET tool r-TuBound. We evaluated r-TuBound on challenging examples coming from the WCET community. Our experiments of using r-TuBound for WCET Squeezing, selective symbolic execution and loop bound computation highlight the practical benefits of our work. It is not unusual, for example, that in a single iteration of WCET Squeezing, the precision of WCET bound is improved by 3%.

In closing, WCET Squeezing demonstrates the benefit of using symbolic methods, including symbolic execution, symbolic computation, and theorem proving, in the WCET analysis of programs. Our results empower the strength of existing WCET analysis techniques, solving problems no other WCET analyzer could so far solve. We conclude, that symbolic methods, if applied efficiently as proposed in this article, are a useful addition to program analysis techniques and particularly suited for WCET analysis of programs. We therefore believe that extending our results with more sophisticated symbolic methods is a challenging but rewarding task to be further investigated. In particular, we are interested in designing new methods for deriving precise WCET bounds for programs with complex control flow and data structures, including floating points, arrays, and pointers.

Acknowledgements. We acknowledge funding from the Vienna University of Technology, the Austrian research project FWF S11409-N23, the ERC Starting Grant 2014 SYMCAR 639270, the Wallenberg Academy Fellowship 2014, and the Swedish VR grant GenPro D0497701. We also acknowledge support of this research by the EU FP7 COST Action IC 1202 Timing Analysis on Code-Level (TACLe). The third author was supported partly by the Institute de Recherche en Informatique de Toulouse (IRIT), France.

References

Ballabriga, C., Casse, H., Rochange, C., Sainrat, P., October 2010. OTAWA: an Open Toolbox for Adaptive WCET Analysis. In: Proc. of IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS). Springer, Austria.
URL `ftp://ftp.irit.fr/IRIT/TRACES/11771_seus2010.pdf`

Bang, H. J., Kim, T. H., Cha, S. D., 2007. An Iterative Refinement Framework for Tighter Worst-Case Execution Time Calculation. In: Proc.of ISORC. pp. 365–372.

Berkelaar, M., Eikland, K., Notebaert, P., 2004. lp_solve. Software, `http://lpsolve.sourceforge.net/5.5/`.

Biere, A., Knoop, J., Kovács, L., Zwirchmayr, J., 2013. SmacC: A Retargetable Symbolic Execution Engine. In: Proc. of ATVA.

Boddy, M. S., 1991. Anytime Problem Solving Using Dynamic Programming. In: Proc. of AAAI. pp. 738–743.

Bodík, R., Gupta, R., Soffa, M. L., Nov. 1997. Refining Data Flow Information Using Infeasible Paths. SIGSOFT Softw. Eng. Notes 22 (6), 361–377.

Brandner, F., Jordan, A., 2014. Subgraph-Based Refinement of Worst-Case Execution Time Bounds. Tech. Rep. hal-00978015 (version 1), ENSTA.

Brummayer, R., Biere, A., 2009. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In: Lecture Notes in Computer Science (LNCS). Vol. 5505. Springer, pp. 174–177, TACAS'09.

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

Cadar, C., Ganesh, V., Pawlowski, P. M., Dill, D. L., Engler, D. R., 2006. EXE: Automatically Generating Inputs of Death. In: Proc. of CCS. pp. 322–335.

Cary R. Spitzer, 2001. The Avionics Handbook. The Electrical Engineering Handbook Series. CRC Press Llc.
URL `http://books.google.fr/books?id=RukamQEACAAJ`

Cassé, H., Birée, F., Sainrat, P., 2013. Multi-architecture Value Analysis for Machine Code. In: Proc.of WCET. pp. 42–52.

Cerny, P., Henzinger, T., Radhakrishna, A., 2013. Quantitative Abstraction Refinement. In: Proc. of POPL. pp. 115–128.

Chattopadhyay, S., Roychoudhury, A., 2011. Scalable and Precise Refinement of Cache Timing Analysis via Model Checking. In: Proc.of RTSS. pp. 193–203.

Chattopadhyay, S., Roychoudhury, A., 2013. Scalable and precise refinement of cache timing analysis via path-sensitive verification. Proc.of RTS 49 (4), 517–562.

Clarke, E. M., Grumberg, O., Jha, S., Lu, Y., Veith, H., 2000. Counterexample-Guided Abstraction Refinement. In: Proc. of CAV. pp. 154–169.

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

Edison Design Group, 1992. The C++ Front End. `http://www.edg.com/index.php?location=c_frontend`.

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

Gustafsson, J., 2013. SWEET Manual.
URL `http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/out/webhelp/index_frames.html`

Gustafsson, J., Betts, A., Ermedahl, A., Lisper, B., 2010. The Mälardalen WCET Benchmarks: Past, Present And Future. In: Proc. of WCET. pp. 136–146.

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

Kirner, R., 2012. The WCET Analysis Tool CalcWcet167. In: Proc. of ISoLA (2)'12. pp. 158–172.

Knoop, J., Kovács, L., Zwirchmayr, J., 2011. Symbolic Loop Bound Computation for WCET Analysis. In: Proc. of PSI. pp. 116 – 126.

Knoop, J., Kovács, L., Zwirchmayr, J., 2012. r-TuBound: Loop Bounds for WCET Analysis. In: Proc. of LPAR. Vol. 7180 of LNCS. pp. 435 – 444.

Knoop, J., Kovács, L., Zwirchmayr, J., 2013. WCET Squeezing: On-demand Feasibility Refinement for Proven Precise WCET-bounds. In: Proc. of RTNS. pp. 161–170.

Lattner, C., Adve, V. S., 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proc.of CGO. pp. 75–88.

Li, Y.-T. S. and Malik, S. and Wolfe, A., Dec. 1995. Efficient microarchitecture modelling and path analysis for real-time software. Proceedings of the 16th IEEE Real-Time Systems Symposium, 254–263.

LLNL, 2000. The Rose Compiler: an Open Source Compiler Infrastructure. http://www.rosecompiler.org/.

Milinkovich, M., Feb. 6th 2014. ERTS 2014 Keynote speech: What Can Industry Learn From Open Source Development. Slides available at `http://www.erts2014.org/Site/0R4UXE94/Fichier/ERTS_OpenSourceProcessesv2.pdf`, last accessed: 2014-03-04.

Muchnick, Steven S., 1997. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers.

Nielson, F., Nielson, H. R., Hankin, C., 1999. Principles of Program Analysis. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Prantl, A., 2008. The Termite Library. `http://www.complang.tuwien.ac.at/adrian/termite/Manual/`.

Prantl, A., Knoop, J., Schordan, M., Triska, M., 2009. Constraint Solving For High-level WCET Analysis. CoRR abs/0903.2251.

Prantl, A., Schordan, M., Knoop, J., 2008. TuBound - A Conceptually New Tool for Worst-Case Execution Time Analysis. In: In Proc.of WCET'08. Vol. 8.

Puschner, P. P., Schedl, A. V., July 1997. Computing Maximum Task Execution Times – A Graph-Based Approach. Real-Time Systems 13 (1), 67–91.

Schordan, M., Barany, G., Prantl, A., Pavlu, V., 2007. SATIrE - The Static Analysis Tool Integration Engine. `http://www.complang.tuwien.ac.at/satire/`.

Skiena, S. S., 2008. The Algorithm Design Manual, 2nd Edition. Springer Publishing Company, Incorporated.

Souyris, J., Pavec, E. L., Himbert, G., Jégu, V., Borios, G., 2005. Computing the Worst Case Execution Time of an Avionics Program by Abstract Interpretation. In: Proc. of WCET. pp. 21–24.

Zolda, M., Kirner, R., 2011. Compiler Support for Measurement-based Timing Analysis. In: Proc. of WCET. pp. 62–71.