The Auspicious Couple: Symbolic Execution and WCET Analysis *

Armin Biere[1], Jens Knoop[2], Laura Kovács[3], and Jakob Zwirchmayr[2]

1 **Johannes Kepler University Linz, Austria** `biere@jku.at`
2 **Vienna University of Technology, Austria** `[knoop|jakob]@complang.tuwien.ac.at`
3 **Chalmers University of Technology, Sweden** `laura.kovacs@chalmers.se`

Abstract

We have recently shown that symbolic execution together with the implicit path enumeration technique can successfully be applied for the Worst-Case Execution Time (WCET) analysis of programs. 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 in the timing analysis of programs, we propose to use WCET analysis as a guidance for symbolic execution in order to avoid full symbolic coverage of the program. By focusing only on paths or program fragments that are relevant for WCET analysis, we keep the computational costs of symbolic execution low. Our WCET analysis also profits from the precise results derived via symbolic execution.

In this article we describe how use-cases of symbolic execution are materialized in the r-TuBound tool and present new applications of WCET-guided symbolic execution for WCET analysis. The new applications of selective symbolic execution are based on reducing the effort of symbolic analysis by focusing only on relevant program fragments. By using partial symbolic program coverage obtained by selective symbolic execution, we improve the WCET analysis and keep the overhead of symbolic execution low.

1998 ACM Subject Classification C.3 Special-Purpose and Application-Based Systems

Keywords and phrases WCET analysis, Symbolic execution, WCET refinement, Flow Facts

1 Introduction

Symbolic execution can analyze a program with high precision, by using symbolic instead of concrete input values of the program. Programs are symbolically executed path-wise, and each program path is analyzed in isolation. This however comes at the price that every program path needs to be symbolically executed in order to infer results that are valid for the entire program. In other words, a *full symbolic coverage* of the program is needed for verifying program properties using symbolic execution. As the number of paths increases exponentially with the number of conditionals in the program, computing full symbolic coverage for larger applications is in practice not realistic. Applications of symbolic execution therefore only explore relevant parts of the program behavior and compute a *partial symbolic coverage* of the program.

Such a compromise between precision and computability is also present in the Worst-Case Execution Time (WCET) analysis of programs. Namely, a successful WCET analysis requires a balance between the speed and the precision of the deployed analysis. Precision of the analysis is gained by

*This research is supported by the FP7-ICT Project 288008 T-CREST, the FWF RiSE projects S11408-N23 and S11410-N23, the WWTF PROSEED grant ICT C-050, the FWF grant T425-N23, and the CeTAT project of the TU Vienna.

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

applying powerful program analysis techniques that gather information about the program and pass it to further analysis- and computation-steps. Precision of the analysis yields tight WCET estimates, however, at the cost of high computational effort; this sometimes prevents the analysis to terminate within a given time-limit. Precision of the WCET analysis is therefore often traded for its speed: faster analysis with likely imprecise WCET estimates is preferred to a precise but slow one. Following this compromise, automated methods for refining imprecise WCET results into tighter ones are needed in the WCET analysis of programs.

In this paper we argue that combining symbolic execution with traditional WCET analysis yields an efficient and precise method for computing WCET estimates. We show that, for using symbolic execution in WCET analysis, a partial symbolic coverage of the program is sufficient to tighten and, eventually, prove the computed WCET bound of the program to be precise. We do so by applying *selective symbolic execution* over program parts and avoid the path explosion problem of traditional symbolic execution. To this end, we use costly symbolic execution only for those parts of the program that influence the WCET estimate. Our WCET-guided symbolic execution is a precise selective symbolic execution for *relevant* parts of the program, and avoids the computational overhead of full symbolic execution. Our workhorse in this paper is the r-TuBound toolchain [19]. We extend r-TuBound with symbolic execution (Section 3) and present three existing applications of symbolic execution in r-TuBound for WCET analysis (Section 4):

- We use symbolic execution in r-TuBound on *reduced program fragments*, and analyze programs which could not be analyzed by r-TuBound so far due to the too restrictive programming model of [18];
- We deploy symbolic execution in r-TuBound to *compute loop bounds*. This extension allows r-TuBound to calculate loop bounds in cases where it has previously failed;
- Based on the implicit path enumeration technique (IPET) [22], we use the result of an initial WCET analysis and apply symbolic execution in r-TuBound to *tighten initial WCET estimates and eventually prove these bounds precise*, by applying the work of [20].

Based on our current use of symbolic execution in r-TuBound, we also discuss further applications of symbolic execution for the WCET analysis of programs (Section 5). These new directions rely on partial symbolic coverage of the program and include:

- inferring *precise execution frequencies* for loops with conditionals;
- generating *WCET path test-cases* used in measurement-based WCET analysis tools;
- automated support for *mode-sensitive analysis* of programs after an initial (IPET-based) WCET analysis;

We believe that the WCET applications proposed and discussed in this article encourage the further use of symbolic execution in WCET analysis. The symbolic execution extensions that are already implemented in r-TuBound were successfully applied on examples coming from the WCET tool challenge [13]: WCET estimates were tightened and the cost of symbolic execution was low. We are confident that the overhead for the proposed applications of symbolic execution in WCET analysis can be kept low, while providing valuable information about the program.

2 Preliminaries

In this section we give a brief overview of the main ingredients of symbolic execution and WCET analysis. For more details, we refer to [24, 21].

Symbolic Execution. Symbolic execution uses symbolic instead of concrete input data to symbolically execute a program. To do so, input variables of the program are assumed to be "symbolic," which means that they can have an arbitrary value (conforming to the specified data-type). If a conditional statement splits the control-flow of the program, symbolic execution follows both successor

edges of the conditional, restricting possible values of symbolic variables according to the condition. For example, if a conditional executes the *true*-edge of the condition only if a variable has a certain constant value, then symbolic execution assumes the constant value for the variable when following this edge. Thus, symbolic variable values are restricted by path conditions or assumptions involving the respective variable. This allows to track complex constraints for each variable and use solvers, such as [6], to reason about the derived constraints.

Symbolic execution of programs with conditionals (and loops) leads in many cases to the path explosion problem, as the number of paths needed to be symbolically executed increases exponentially with the number of conditionals in the program. Hence, full symbolic coverage of larger applications is infeasible in practice. The problem of path explosion can be addressed in different ways, for example, by using heuristics for computing only partial symbolic coverage of the program, for instance in the context of test-case generation and bug-hunting. A detailed overview on symbolic execution is presented in [9].

WCET Analyis. A static WCET analysis toolchain typically includes several high-level analyses that gather so-called flow fact information about the program. Essential flow facts include loop bounds and execution frequencies (i.e. worst-case execution counts) of conditional edges in the program. When computing WCET estimates, the underlying hardware architecture needs to be analyzed for inferring execution times of program blocks. Additional hardware features, such as cache-configuration and pipeline layout, also need to be taken into account. Precise WCET bounds denotes, in this article, that any over-estimation of the WCET is due to imprecise hardware modelling and not due to infeasible paths.

With the block execution times computed for the program, various techniques can be applied to find the path that exhibits the WCET of the program. One of the most common approaches is the implicit path enumeration technique (IPET) [22]. It is applied on the control flow graph (CFG) representation of the program and relies on the fact that each program execution satisfies the following flow properties: (i) a program execution executes the entry point of the program once and (ii) other program blocks are executed as often as their predecessor blocks. Therefore, any program block following the entry point is executed once, unless it appears in a conditional or loop statement. For a conditional (iii) the total sum of the execution frequencies of its conditional blocks (denoted the `true`- and `false`-block of the conditional) is the same as the execution frequency of the predecessor block of the conditional, which is the condition-block. For blocks inside loops, (iv) the execution frequencies are multiplied by the loop bound. Hence, when applying IPET, loop bounds are assumed to be supplied as flow-facts.

When using IPET, the program is represented as an integer linear program (ILP) where each program block is modeled by an ILP variable that has the block execution time associated with it. An ILP-solver is then used to solve the flow problem (i) - (iv) specified above. By using such an ILP encoding on execution frequencies, the solution of the corresponding ILP problem assigns values to the ILP variables, that is, execution counts of program blocks. A WCET estimate for the program is then obtained by maximizing the sum of the products of execution frequencies and block execution times for each block.

As flow facts about programs are not always precise (e.g. loop bounds are not exact but are over-approximated), the ILP encoding of the program usually encodes numerous spurious program paths. Some of these spurious program paths yield high execution times. Therefore, the WCET estimate computed from the ILP solution is only an over-estimation of the actual WCET: its precision crucially depends on the quality of additional flow facts supplied to IPET. In an IPET problem that contains a conditional inside a loop, the ILP solver will always assume the conditional block that exhibits the higher execution time to be executed. This might be overly pessimistic. Supplying additional flow facts that specify valid execution frequencies for the conditional block therefore

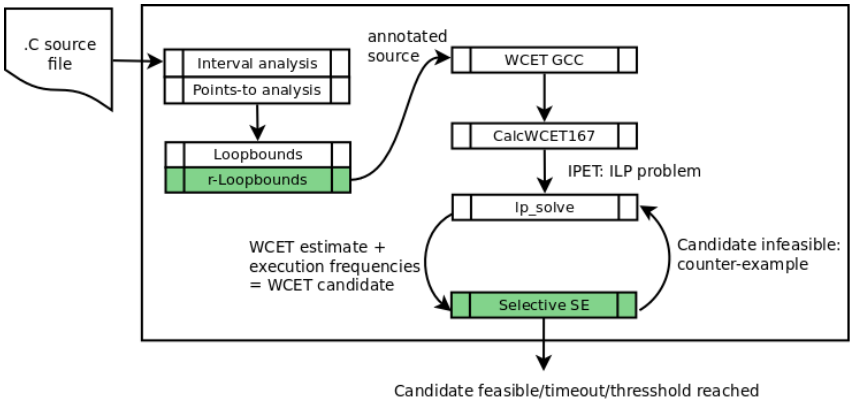

Figure 1 Architecture of the WCET toolchain. Colored parts rely on symbolic execution.

results in a tighter WCET estimate.

3 Selective Symbolic Execution in r-TuBound

In this section we describe our extensions to the r-TuBound tool [19], by integrating symbolic execution into r-TuBound. The common theme of all these extensions relies on a selective use of symbolic execution for timing analysis, instead of symbolically executing the whole program.

The r-TuBound tool is a WCET analysis toolchain described in [19]. It applies high-level analyses on the source level and calculates WCET estimates using a low-level analyzer. In a nutshell, the main steps of r-TuBound, as presented in [19], are as follows. Given a program with loops written in a restricted class of C, r-TuBound deploys interval and points-to analysis to derive bounds, called loop bounds, on the number of loop iterations. The source code, annotated with the results of these analyses, is then compiled by the WCET aware compiler of r-TuBound. The resulting assembly is analyzed by the WCET analyzer CalcWCET167 of the Infineon C167 microprocessor [17]. To this end, r-TuBound applies the IPET approach, solves the resulting ILP problems and derives WCET estimates as outputs.

Using the framework of [19], we have extended r-TuBound by using symbolic execution in WCET analysis. Figure 1 shows the current workflow of r-TuBound, where the colored components correspond to our new extensions to r-TuBound. We refined the loop bound computation step of r-TuBound by exhaustive symbolic execution for loops, implemented in the `r-Loopbounds` step of Figure 1. We also added a *selective symbolic engine* to r-TuBound for deriving tight WCET bounds, listed as the `Selective SE` step of Figure 1. In the rest of this section we describe the integration of symbolic execution in r-TuBound.

Symbolic Execution and r-TuBound. We extended r-TuBound with the symbolic execution engine of [24, 4] which uses the theories of bit-vectors and arrays. This extension is applied in r-TuBound to construct a precise memory-model of the program, as follows. Given an input program (written in C), the program is first parsed and stored as an abstract syntax tree in the code-list. The code-list is then further processed, as follows.

Program paths are extracted from the code-list generated by the parser, and are symbolically executed by writing and reading the symbolic representation of the program memory. As a result, a representation of the symbolic program execution is obtained as a set of satisfiability modulo theory (SMT) formulas, where the SMT formulas are expressed in theory of bit-vectors and arrays. Verification conditions, expressing runtime and memory-access properties, are generated to guarantee runtime- and memory-safety of the program (e.g. does not dereference `NULL`). Properties that hold in the symbolic representation of the programs are also guaranteed to hold in the actual program.

Instead of symbolically executing all paths in the program, selective symbolic execution allows to execute paths selectively by supplying a sequence of branching decisions that encode executions of the program. The branching decisions are extracted from the IPET solution. These decisions allow to iteratively select and symbolically execute WCET paths. Precise constraints are only inferred about the WCET path, reducing the costs for symbolic execution.

4 Precise WCET Analysis without Path Explosion in r-TuBound

Symbolic execution infers precise program properties that can further be used in an IPET-based WCET analysis. Nevertheless, symbolic execution comes with the cost of analyzing each program path, an unsatisfactory task for large programs with loops and conditionals. To avoid this problem, when using symbolic execution for WCET analysis in r-TuBound, our work identifies relevant program parts, that is program fragments for which symbolic execution is necessary to be applied. More precisely, we apply symbolic computation in the following three scenarios: (i) analyzing reduced program fragments in isolation of the entire program, by reasoning about single statements in loop bodies, as well as about loops and nested loop structures, (ii) deriving loop bounds on the number of loop iterations, and (iii) analyzing a low number of paths for refining and proving precise WCET estimates. For doing so, in (i) we rely on the programming model of [18], (ii) makes uses of [24, 4] and is restricted only to the programming model of the underlying symbolic execution engine, and (iii) is based on the theoretical framework presented in [20].

```
1: int main (int flag) {
2: int i;
3: for(i = 0; i < 5; i++)
4:   if(i == 4 && flag) {
5:     i = 0;
6:     flag = 0;
7:  }
8: }
```

Figure 2 Our running example. All data is assumed uninitialized.

In what follows, we overview the relevant parts of symbolic execution in each of the above scenarios of r-TuBound, and illustrate our work on the example of Figure 2. Based on the current applications of symbolic execution in r-TuBound, in Section 5 we will outline new and ongoing applications of symbolic execution for WCET analysis.

(i) Analyzing reduced program fragments. We use the symbolic execution framework of [24, 4] to verify arithmetic properties about one or more conditional updates to the loop counter of loops. Symbolic execution is appropriate in this setup as interval analysis often lacks sufficiently precise analysis results. By using symbolic execution, in the current version of r-TuBound we can verify arithmetic properties about loop counters without the need of deriving tight intervals for the values of loop counters. Even more, we are able to handle a more general programming model than the one described in [18]. Namely, we can analyze loops whose conditional updates are arbitrary expressions in the combined theory of linear arithmetic, bit-vectors and arrays, whereas [18] was restricted to the theory of linear arithmetic. If the such derived arithmetic properties are proved to be correct by using [24, 4], the loop bound computation step of r-TuBound can safely be applied. Our use of symbolic execution also allows to merge conditional updates over the loop counter into a so-called *combined minimal update*, which then yields a tighter loop bound, and hence a tighter WCET estimate.

Example. The loop analysis step of [18, 19] fails to compute a bound for the loop in Figure 2. This is so because the conditional update to the loop counter i (in line 5) violates the computed loop bound in cases when the loop counter is reset, i.e. when `flag` is true. In Figure 2, i increases in the loop header. Therefore, the `r-Loopbounds` step of r-TuBound symbolically executes the conditional update for an arbitrary (i.e. symbolic) loop iteration and verifies that the conditional update can only increase the value of i. If this property is violated, it is not safe to compute a loop bound using the techniques of [18, 19] implemented in the `Loopbounds` step of Figure 1. In the example in Figure 2, the property is false, the conditional update can decrease the loop counter, and therefore no loop bound is computed. We therefore proceed to exhaustive symbolic execution on a *reduced* version of Figure 2, as detailed below.

(ii) Loop bound computation. If the loop bound computation step of [18] in r-TuBound fails, we apply exhaustive symbolic execution over the *reduced* program in r-TuBound. By a reduced program we consider the program that only contains the loop under study together with relevant variable declarations, i.e. the variables used in the loop. Variable values are treated as symbolic, with the exception of the loop counter. Additional information, such as intervals for variable values or program slices, can also be supplied in this step to improve the precision of the loop bounds. A supplied time-limit guarantees termination of the approach. The reduced program is symbolically executed in r-TuBound, where r-TuBound initially sets the loop bound to 1. If symbolic execution reports that the negation of the loop condition is unsatisfiable on the (unwound loop) path, the loop bound is increased by one. Upon termination within the time-limit, no execution of the program exhibits a higher loop bound. Such a use of symbolic execution in r-TuBound is especially useful when bit-precise reasoning is required.

Example. The approach of [18] cannot derive a loop bound for Figure 2. Therefore, (bounded) exhaustive symbolic execution is applied to only analyze the program loop. By using symbolic execution in r-TuBound, we derive 9 as the exact loop bound of Figure 2.

(iii) Deriving precise WCET bounds. The WCET analysis approach presented in [20] relies on the tight combination of a symbolic execution engine and a WCET analyzer, as follows. An IPET-based WCET analysis is first used, yielding an ILP problem encoding constraints on the program flow and a WCET estimate. Next, symbolic execution on single program paths is applied in order to infer constraints that allow tightening and ultimately proving precise the WCET bound. For doing so, the ILP solution describing the execution frequencies of program blocks and the ILP problem is analyzed, and one or more spurious program execution traces exhibiting the WCET are identified. These execution traces are then excluded from the set of possible program executions, by adding a new ILP constraint to the ILP problem. The resulting new ILP problem is then used in the next iteration of the approach, by again applying IPET in combination with symbolic execution. At each iteration of [20], either a new and lower WCET estimate is derived or a program trace exhibiting the old WCET is obtained. In the latter case, the computed WCET is the actual WCET of the program, and the method of [20] terminates. The computed WCET is precise wrt the underlying hardware-model.

Note that in [20] only paths extracted from the ILP solution are symbolically executed, yielding thus a partial remedy to the path explosion problem of symbolic execution. The WCET estimates used in [20] serve as a measure for the *relevance* of paths: it allows to select relevant paths that need to be symbolically executed in order to tighten the WCET. We call these selected paths *relevant*.

We implemented the approach of [20] in r-TuBound and describe it here. The program is parsed and the IPET approach is applied. An ILP problem is next obtained and solved, by using the ILP solver lp_solve of [2]. The obtained ILP solution encodes a WCET estimate of the program. Further, the execution trace specified by the ILP solution is (re)-constructed. To this end, we encode execution traces as sequences of branching decisions, where branching decisions are obtained as follows. For each conditional block, i.e. a program block with jump-instructions to other blocks, the execution frequency of the jump-targets specifies which block is assumed to be executed on the program path exhibiting the WCET estimate. If the condition evaluates to `false`, the `else`-block of the conditional is executed. Thus, the ILP solution specifies an execution frequency of zero for the `then`-block and an execution frequency of one for the jump-target, the `else`-block. The inferred branching decision is `false`.

If both edges of a conditional have an execution frequency greater than zero in the ILP solution, both program blocks of the conditional are executed. In this case, the WCET candidate encodes multiple actual program executions. Hence, from the ILP solution, one or more program paths exhibiting the WCET estimate can be constructed. We therefore refer to program paths exhibiting the

WCET as *WCET trace candidates*. If one of the WCET trace candidates is feasible, the computed WCET bound is proven precise and yields the actual WCET of the program in the underlying hardware model. If all WCET trace candidates are infeasible, the ILP problem of IPET can be refined and a tighter WCET estimate can be computed. WCET trace candidates in r-TuBound are expressed as SMT formulas in the combined theory of bit-vectors and arrays, and the SMT solver Boolector [6] is used to check their feasibility. In our current r-TuBound implementation we rely on a manually constructed mapping between the assembly analyzed with CalcWcet167 and the source of the application. In other words, we manually verify that the source and assembly exhibit a compatible branching behaviour. Construction of this mapping can be omitted when symbolic execution is applied on the binary level.

Example. The initial ILP solution derived from the ILP problem of IPET (using the loop bound 9) specifies the execution of the conditional block in each iteration of the loop in Figure 2. The WCET trace candidate extracted from the ILP solution encodes exactly one program path; this path executes the conditional block 9 times. This WCET trace candidate is specified by the following sequence of branching decisions `ttttttttt` (9 times `t`), where `t` denotes the true-edge of the conditional statement. By symbolically executing this WCET trace candidate, we derive the infeasibility of `ttttttttt`. Thus, an additional ILP constraint is constructed to exclude this WCET trace candidate from the ILP problem. The new ILP problem is solved again, yielding a tighter WCET estimate and new WCET trace candidates. This process is iterated until a feasible WCET trace candidate is found. In Figure 2, a feasible WCET trace candidate is derived after 8 iterations of refining the ILP problem and WCET estimate. As a result, the exact execution frequency of the `true`-block of the conditional is inferred and constrained to 1. In a simplified scenario where execution of each program instruction takes 1 time unit, the actual WCET of the program is then derived to be 40 time units. The WCET is derived by summing up the execution times (a)-(c) detailed below. (a) The initialization in the loop header (`i=0`) takes 1 time unit; (b) Among the execution frequencies of loop iterations, based on the derived execution frequency of the conditional statement, the following case distinction is made: for eight loop iterations, an iteration takes 4 time units, 1 time unit is the evaluation of the loop condition `i<5`, 2 time units are needed to execute the condition `i==4 && flag` of the conditional (two instructions) and 1 time unit is taken for the loop counter increment `i++`. All together, these eight loop iterations take 32 time units. One loop iteration, namely the one in which the conditional statement is executed, requires 6 time units to be executed. When compared to the previous cases, the additional 2 time units result from the execution of the true-block of the conditional. (c) The last evaluation of loop condition $i < 5$ after nine loop iterations takes 1 time unit.

Summarizing, the applications (i)-(iii) discussed above share a common approach: instead of symbolically executing the entire program, selective symbolic execution is performed only on fragments or single paths of the program in order to prevent symbolic execution from running into the path explosion problem.

5 Further Applications of Symbolic Execution for WCET

In this section, we discuss three additional applications of symbolic execution in WCET analysis, by using the symbolic execution framework presented in Sections 3. Similarly to Section 4, these applications apply only partial symbolic coverage of the program, as follows. The material presented in this section is work-in-progress and requires further experimentation.

Precise execution frequencies for loops. When the loop bound computation techniques of [18] fails in r-TuBound, exhaustive symbolic execution over the loop is applied as described in Section 4(ii). The application scenario of Section 4(ii) can be further extended to compute execution frequencies for conditional blocks inside the loop, as follows. By applying exhaustive symbolic execution, a

loop bound and feasible WCET traces are derived. The execution frequencies of program blocks in the feasible traces is also obtained. In case of multiple feasible traces, a set of execution frequencies is inferred for each block. We then set the execution frequency of a block to be less or equal to the highest and greater or equal to the lowest value among its set of possible execution frequencies, and use this value in the ILP encoding of the program. To ensure that the such chosen execution frequency is precise, we use the approach of Section 4(iii) and iteratively refine the execution frequency of each edge inside the loop.

Example. Consider Figure 2 again. An initial IPET-based WCET analysis (without additional flow facts) sets the execution frequency of the `true`-block of the conditional to 9. By applying our approach, we use exhaustive symbolic execution on the loop of Figure 2. As a result, we derive the maximum execution frequency of 1 for the `true`-block of the conditional. Using this additional flow fact in the initial IPET-based WCET analysis, the precise WCET of the program is also derived.

(ii) WCET path test-cases. Our implementation of Section 4(iii) in r-TuBound can also be used to generate WCET path tests-cases, as follows. The approach of Section 4(iii) already extracts and symbolically executes WCET trace candidates. A symbolic execution engine can be used to generate concrete program inputs from the trace that is symbolically executed, forcing actual executions of the program to follow the same trace. The program inputs for feasible WCET trace candidates thus represent program inputs which lead to the execution of the actual WCET path, that is, a concrete program path exhibiting the WCET. The program inputs generated by symbolic execution can be used by hardware-aware dynamic WCET analyzers, see e.g. [23], to take additional, hardware-dependent, time measurements. We refer to such analyzers as measurement-based WCET analyzers. The generated test-cases can help measurement-based WCET analyzers to derive relevant timing behavior of the application on the WCET path. At the same time, the measurements can be used as feedback about the precision of static analyzers: little variation between the statically computed WCET estimate and the measurements on the WCET path are a sign of precision of the static analyzer. Even more, the statically calculated WCET estimate must never be below the WCET value reported by the measurement-based analyzer.

Example. Consider Figure 2. The WCET analysis of Section 4(iii) infers the WCET path to execute the `true`-block of the conditional once, when `i` is 4 and when the value of the symbolic variable `flag` is assumed to be `true`. Based on the symbolic execution of this WCET trace, the symbolic execution engine in r-TuBound generates a test case which initially assumes `flag` to evaluate to `true`. Supplying this input to a measurement-based WCET analyzer that runs the actual program allows to take measurements on the WCET path.

(iii) Mode-sensitive analysis. The symbolic execution engine of r-TuBound used in Section 4(iii) can further be used to support automated *mode-sensitive* WCET analysis. Modes characterize a certain state that the program is executed in. For example, the program could be in a "normal operation", an "initializing" or an "error" mode. Given a program, its precise WCET is derived using the method of Section 4(iii). Assume now that the program is modified, for example, by setting a program flag resulting in a different mode of execution. The approach of Section 4(iii) will then automatically recompute the WCET for the changed program. This mode-sensitive behaviour can be observed in functions from the Mälardalen benchmark suite of [13], where the flags (`init`, `found`) control the execution mode and thus the WCET path. These control variables are not yet found automatically in r-TuBound. However, simple methods could be used to identify these flags, for example by checking whether two conditions are mutually exclusive. We leave the integration of r-TuBound with such techniques for further work.

Example. Consider again Figure 2 and assume that `flag == true` indicates an *error-mode*. Changing the initial value of the variable `flag` from uninitialized to `false` changes the feasible WCET path of the program, and hence the WCET computed in Section 4. In the such changed

program the approach of Section 4(iii) infers that the loop is executed 5 times instead of 9 times as computed in Section 4. Applying the technique of Section 4(iii) to the changed program incrementally changes the ILP to reflect the change in the program behavior, resulting in a WCET for the new WCET path, that is a WCET when the program is not executed in the *error* mode.

Summarizing, similarly to Section 4 the three applications discussed in this section selectively apply symbolic execution to WCET relevant parts of the program. In (i), we argue that bookkeeping the exact frequencies of program blocks can be done in a cheap way and exhaustive symbolic execution can be applied to derive loop bounds. In (ii) and (iii) we rely on the implemented symbolic execution infrastructure and use them in conjunction with the approach of Section 4(iii). This way, we only apply symbolic execution on a (reduced) number of WCET trace candidates, and avoid the burden of exploring all program paths.

6 Related Work

Symbolic execution originally was used for test-case generation and has recently found more and more applications in program verification. For example, it has been successfully applied for test-case generation and bug-hunting [7, 8]. Applications of symbolic execution in program verification use various heuristics to speed up symbolic execution, identify and track relevant program information and use constraint solvers to prove caching queries. Our symbolic execution engine in r-TuBound offers only few heuristics and derives as much program information as needed for the WCET analysis. In general, inferring precise program information comes with high computational costs, a problem which we avoid by using selective symbolic execution. As presented, r-TuBound applies symbolic execution when information about the program is too coarse or when other analysis methods fail while keeping the computational costs for symbolic execution low. A similar idea is presented in [5] where symbolic execution is used to refine refine spurious results of a path feasibility analysis. To this end, in [5] 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 r-TuBound always executes single paths.

Symbolic execution for WCET analysis is also used in [16] by avoiding some typical pitfalls of symbolic execution. For example, loops are not unfolded and hence multiple executions of the same block is omitted. We note that [16] analyses each program block whereas our selective symbolic execution approach in r-TuBound only analyses relevant program blocks and paths.

A related approach to our symbolic execution method is described in the abstract execution framework of [14], 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 applied in [14], where an abstract value can, for example, be represented as an interval. The use of an abstract values prevents the evaluation of multiple conditional branches and a single abstract execution is computed to represent the execution of multiple (concrete) program paths. This is not the case in the traditional use of symbolic execution. However, when compared to r-TuBound, we note that abstract execution in [14] analyses the entire program, whereas in r-TuBound we apply symbolic execution only on relevant parts of the program. An itegration of abstract execution in r-TuBound is an interesting research direction for further work.

In the traditional use of static WCET analyzers, high-level tools gather flow fact information about the program under analysis. This information is consequently used in further (low-level) WCET analysis and hardware features of the underlying system are also modeled. Static WCET analyzers, see e.g. [19, 1, 12], often use the IPET technique [22] to calculate WCET estimates. This leads to over-estimation of the WCET since the IPET modeling of the program usually encodes spurious execution traces that are infeasible in the concrete program. The approach of [20] addresses the problem of refining imprecise WCET estimates, by using symbolic execution in conjunction

with IPET. We implemented this combination in r-TuBound. The results and applications of our implementation, described in this paper, offer an automated technique to reduce, and possibly avoid over-estimation in WCET computation. A similar method is also described in [15], 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 is checked using model checking, by encoding block execution frequencies as program assertions. Unlike [15], we apply path-wise symbolic excution to avoid model checking the entire program and use SMT solving for checking feasibility of program paths.

Measurement-based timing analysis techniques, such as [23], 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 a tight WCET bound with a high confidence. The method of [23] systematically generates test-cases for arbitrary program executions, based on model checking and various heuristics. In the proposed application of symbolic execution in r-TuBound we generate test-cases only for program executions along the WCET trace candidate paths(s).

A different approach to WCET analysis is given in [10] and relies on segment- and state-based abstract interpretation [11]. This state-based approach has similarities with the ILP problem refinement of r-TuBound. Integrating this approach in r-TuBound is an interesting task to be investigated.

7 Conclusion

We described applications of symbolic execution in WCET analysis. These include reasoning about single statements in loops, computing loop bounds, and refining the results of an a-priori used WCET analyzer. These applications are implemented in the r-TuBound tool and described in this paper. The presented approaches have been successfully tested on a number of WCET benchmarks. Additional applications of symbolic execution can be implemented in r-TuBound by only using minor changes on the underlying symbolic execution engine. With such changes at hand, we argue that symbolic execution can also be used in hardware-aware dynamic WCET analyzers.We believe that, an efficient use of symbolic execution, called selective symbolic execution in this article, gives a valuable extension to the program analysis toolbox applied in WCET analysis.

References

1 C. Ballabriga, H. C. Hugues, C. Rochange, and P. Sainrat. OTAWA: an Open Toolbox for Adaptive WCET Analysis. In Proc. of IFIP Workshop (SEUS), pages 35–46, 2010.

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

3 D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The Software Model Checker Blast. STTT, 9(5-6):505–525, 2007.

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

5 Bodík, Rastislav and Gupta, Rajiv and Soffa, Mary Lou. Refining Data Flow Information Using Infeasible Paths. SIGSOFT Softw. Eng. Notes, 22(6):361–377, Nov. 1997.

6 R. Brummayer and A. Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In Proc. of TACAS, pages 174–177, 2009.

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

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

9 C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. Commun. ACM, 56(2):82–90, 2013.

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

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

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

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

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

15 Ho Jung Bang and Tai Hyo Kim and Sung Deok Cha. An Iterative Refinement Framework for Tighter Worst-Case Execution Time Calculation. In Proc.of ISORC, pages 365–372, 2007.

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

17 R. Kirner. The WCET Analysis Tool CalcWcet167. In Proc. of ISoLA (2), pages 158–172, 2012.

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

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

20 J. Knoop, L. Kovács, and J. Zwirchmayr. WCET Squeezing: On-demand Feasibility Refinement. 2013. under submission.

21 A. Prantl, M. Schordan, and J. Knoop. TuBound - A Conceptually New Tool for Worst-Case Execution Time Analysis. In Proc. of WCET, 2008.

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

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

24 J. Zwirchmayr. A Satisfiability Modulo Theories Memory-Model and Assertion Checker for C. Master's thesis, JKU Linz, Austria, 2009.