Noname manuscript No.
(will be inserted by the editor)

Scalable and Precise Estimation and Debugging of the Worst-Case Execution Time for Analysis-Friendly Processors

A Comeback of Model Checking

Martin Becker[1] · Ravindra Metta[2] · R Venkatesh[2] · Samarjit Chakraborty[1]

This is a pre-print of an article published in the *International Journal on Software Tools for Technology Transfer*. The final authenticated version is available online at: `https://doi.org/TODO`

Abstract Estimating the Worst-Case Execution Time (WCET) of an application is an essential task in the context of developing real-time or safety-critical software, but it is also a complex and error-prone process. Conventional approaches require at least some manual inputs from the user, such as loop bounds and infeasible path information, which are hard to obtain and can lead to unsafe results if they are incorrect. This is aggravated by the lack of a comprehensive explanation of the WCET estimate, i.e., a specific trace showing how WCET was reached. It is therefore hard to spot incorrect inputs and hard to improve the worst-case timing of the application. Meanwhile, modern processors have reached a complexity that refutes analysis and puts more and more burden on the practitioner. In this article we show how all of these issues can be significantly mitigated or even solved, if we use processors that are amenable to WCET analysis. We define and identify such processors, and then we propose an automated tool set which estimates a precise WCET without unsafe manual inputs, and also reconstructs a maximum-detail view of the WCET path that can be examined in a debugger environment. Our approach is based on Model Checking, which however is known to scale badly with growing application size. We address this issue by shifting the analysis to source code level, where source code transformations can be applied that retain the timing behavior, but reduce the complexity. Our experiments show that fast and precise estimates can be achieved with Model Checking, that its scalability can even exceed current approaches, and that new opportunities arise in the context of "timing debugging".

E-mail: martin.becker@tum.de, Tel.+49-89-289-23556

[1]Chair of Real-Time Computer Systems, Technical University of Munich, Munich, Germany · [2]Tata Research Development and Design Centre, Pune, India

Keywords Worst-Case Execution Time · Debugging · Static Analysis · Deterministic Processor

1 Introduction

Many real-time systems need to provide strict guarantees for response times. For instance, airbag deployment in a car, collision avoidance systems in aircraft, and control systems in spacecraft have to meet deadlines for ensuring vehicle and passenger safety. The need to analyze this class of time-critical systems has been the main motivator [50] for research on calculating the Worst-Case Execution Time (WCET), which is the longest time a program takes to terminate, considering all possible inputs and control flows.

This timing depends both on the structure of the program, as well as on the processor it is running on. In the most general case, the WCET problem is undecidable (e.g., because loop bounds have to be known), and hence only an upper bound of the WCET can be determined through automatic analysis. Therefore, in a practical setting, the problem reduces to finding the *tightest* safe upper bound, and in particular using a technique that scales well with program complexity.

The techniques being applied today for analyzing the timing behavior, such as Integer Linear Programming (ILP) and Abstract Interpretation (AI), or a combination thereof [59], work very well for analyzing complex programs, but they exhibit several weaknesses: (1) User annotations, such as loop bounds, have to be provided [59,55], but are hard to obtain, influence the tightness and may even refute the soundness of the WCET estimate. Providing too large bounds leads to a large overestimation, and too small bounds may yield an unsafe estimate. As a result, providing safe and tight bounds has become a research field on its own with a wide range of different approaches, e.g., using Abstract Execution [25],

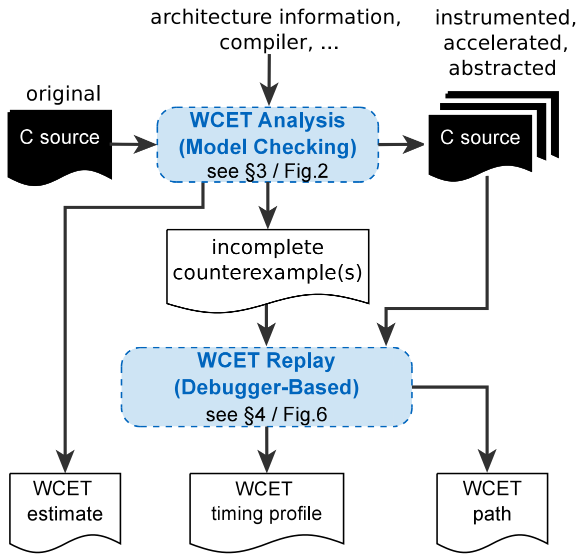


Fig. 1 Overview of our WCET tools and their artifacts

refinement invariants [23] and pattern matching [28]. (2) Existing approaches work at the machine code level, where the high-level information from the original program is hard to extract. Variables are distributed over multiple registers, type information is lost, loops and conditional statements can be implemented in many different ways, and indirect addressing can make it close to impossible to track data flows and function calls. As a consequence, overapproximations have to be used, which usually result in pessimistic estimates. (3) Overapproximations are also used to bound the control flow. As a result, the control flow that would correspond to WCET might not even exist in many cases, contributing further to less tight estimates. (4) Finally, practitioners are facing another challenge with today's analysis tools. Once the WCET of an application has been computed, the output offers little to no explanation of how the WCET has developed, even less of how it can be influenced through changes in the program. Clearly, there is room for improvement in the process of WCET estimation.

However, there exists an even more fundamental problem: Analysis cannot keep up with modern processor architecture. In recent years, the research community arrived at the alarming conclusion that a sound analysis of modern processors is almost impossible, since their complex microarchitecture requires considerably large models, which in turn lead to a state-space explosion [44,48,39,33, Chapter 5.20].

In this article, we see this discouraging situation as a chance to anticipate what improvements could be possible in WCET analysis if processors would become more analyzable, as being proposed by various researchers [39,48]. Assuming that we have a "deterministic processor", we question the techniques being used today. Specifically, we show that by virtue of more analyzable processors, WCET analysis can build on approaches that have already been declared dead for that purpose. The weaknesses in the estimation process, such as the difficulty of providing flow annotations and imprecision of the estimate, can be significantly reduced, and the usability of WCET tools increased. Moreover, we get a chance to realize a true "time debugging", through which practitioners get a handle on the timing behavior, in the same way as functional behavior is being debugged today.

The Idea: Assuming that we have a processor that is amenable to WCET analysis (we specify and identify such processors later on), our approach builds on the following ideas:

- First, we propose WCET estimation by analysis of the *source code*, instead of the machine code. This is enabled by choosing analysis-friendly processors, and makes the control flows easier to analyze, and the additional information, such as variable types and their ranges, helps cutting down the number possibilities to be analyzed, thereby providing more precise estimates in shorter time.
- Second, we propose to use *Model Checking* instead of traditional ILP or AI techniques for path analysis, because it can precisely determine the longest feasible path, as it explores all paths in the program explicitly. Therefore, we expect tighter estimates, and ideally we no longer need flow constraints to be specified by the user, which makes WCET analysis safer and easier to apply.
- Last but not least, we propose to leverage the counterexample that is generated by the model checker, to reconstruct the precise path that is taken in the WCET case, and enable an *interactive replay* that shows not only the path being taken and how time is accumulating, but also the contents of all program variables.

However, there are three main challenges in realizing our idea: First, microarchitecture–specific execution times from the machine-level code need to be represented in the source code. Towards that, a back-mapping from machine-level code to source code has to be developed. Second, Model Checking had been considered for WCET estimation earlier, but it was found to scale poorly with program size [58, 40], and thus never been seen as a competitive technique for WCET estimation. We show that this scalability problem can be mitigated by the mentioned shift of the analysis from machine code to source code level, and that source transformation techniques, such as program slicing and loop summarization, can be used to reduce the complexity and allow Model Checking to scale better. The third challenge is to reconstruct the WCET path, such that it is accessible to the user, and provides specific explanation why a path was taken. That is not directly possible from the counterexample, as this carries incomplete information. We therefore need to devise an efficient path reconstruction technique.

Summary of our approach: The overall approach consists of two steps, as depicted in Fig. 1. First, we estimate the WCET using a model checker, then we reconstruct the WCET path from the counterexample that is provided. The WCET estimation – detailed in Fig. 3 – starts by establishing a mapping between machine code and source code, and evaluate the instruction timing of the program. Given this information, we annotate the source code with increments to a counter variable, which is updated at the end of each source block according to the instruction timing. The resulting code is then sliced with respect to time annotations, to remove all computations not affecting the control flow. In the next step, we accelerate all loops that can be accelerated. Next, we overapproximate the remaining loops with a large or unknown number of iterations. Then, we perform an iterative search procedure with a model checker to determine the WCET value. We terminate the search procedure as soon as we find a WCET estimate within the precision specified by the user. Finally, the path reconstruction takes place in a debugger, while forcing decision variables to the values given in the counterexample, and through that reconstruct the precise path leading to WCET, whilst collecting a timing profile similar to *gprof*.

The contributions are as follows:

- Efficient application of Model Checking at source code level to find the worst-case execution time of a C program (Section 3.6).
- Application of source code transformation techniques to reduce the complexity of the system subject to Model Checking (Section 3.4.1ff).
- A debugger-based technique to reconstruct and replay the precise control flow, all variable contents, and a timing profile of the path leading to the worst-case execution time (Section 4).
- A prototype of a tool set called TIC which implements our proposed approach.
- Experiments with the standard Mälardalen WCET Benchmark Suite, to assess the impact of the source code transformations on scalability and tightness of the WCET estimates, in comparison to an ILP-based analyzer and a cycle-accurate simulator (Section 5).

This article is an extension of our earlier work [42].

2 Technical Background

2.1 WCET Analysis

The goal of WCET analysis is to estimate the longest time a (sub)program *P* takes to terminate, while considering all possible inputs and control flows that might occur, but excluding any waiting times caused by sleep states or interruption by other processes. In real-time systems, this estimate is subsequently used as an input for *schedulability analysis*, which then models the influence of other processes and computes an upper bound of the *reaction time* of *P*. The reaction time, finally, should be shorter than any deadline imposed on *P*. For example, the deadline for *P* could be given by the maximum time that is permissible to detect a car crash and activate the airbags. Consequently, the WCET estimate is a vital metric for real-time systems, and thus needs to be *safe* (i.e., never smaller than what can be observed when executing *P*) and *tight* (i.e., as close as possible to the observed value).

The WCET of an application is influenced by the processor architecture, e.g., caches and pipelines, as well as program structure. Therefore, WCET analysis usually comprises the following steps (not necessarily in that order):

1. **Compilation:** Cross-compile *P* for the processor it is supposed to run on. The source code of *P* is translated to machine instructions *I*, applying various optimizations.
2. **Flow Analysis:** Analyze *I* to discover all possible control flows. This includes finding all potential branches in *I* and storing them in a control flow graph *G*, including their branch conditions.
3. **Value Analysis:** Calculate possible ranges for operand values in *I*, to resolve indirect jumps and classify memory accesses into different memory regions (e.g., slow DRAM vs. fast core-coupled memory).
4. **Loop Analysis:** Bound the control flow of *G*, that is, identify loops and compute their maximum execution counts based on branch conditions, and annotate the nodes and edges in *G* with those execution counts.
5. **Microarchitectural Analysis:** Predict the timing effects of caches, pipelines and other architecture-dependent constructs, based on memory mapping and the paths in *G*. Annotate nodes and edges in *G* with instruction timing considering these features.
6. **Path Analysis:** Formulate a mathematical model based on *G* and solve for WCET. The discovered control flow graph *G* and the computed loop bounds are analyzed together to find those paths along *G* which could produce the longest execution time.

Steps 2 through 5 are often referred to as *low-level analysis*, and step 6 as *high-level analysis*. The employed methods typically involve a combination of Abstract Interpretation and Linear Programming [58]: Flow analysis parses the ISA-specific binary and builds the control flow graph, value and loop analysis typically use Abstract Interpretation to deduce variable values and loop counts that may influence control flow or timing, microarchitectural analysis typically builds on Abstract Interpretation to approximate cache and pipeline effects, and finally path analysis is usually done by translating the annotated control flow graph into a constrained optimization problem [38].

2.1.1 WCET-Amenable Processors

In this paper, we focus on "predictable" hardware, as recently defined by Axner et al. [3]. In particular, the ideal processor for WCET analysis has a *constant* instruction timing. By this we mean, that each instruction takes a constant and bounded amount of time, and this time should neither depend on processor states or operand values, nor be subject to additional waiting states (e.g., pipeline stalls due to pending bus transfers). We allow an exception for branch/jump instructions, where variable instruction timing (e.g., taking a branch may take more time vs. not taking it) can be covered in our flow analysis. With such a WCET-amenable processor, it is neither necessary to perform a value analysis at register level, nor do we require complex microarchitectural models. Although these requirements seem unrealistic even for simple processors, they do not automatically forbid the use of features such as pipelines and caches, as we shall explain in the following.

Pipeline Requirements. Processors may have a pipeline, but the timing of successive instructions must not depend on each other. Again, we allow one exception for conditional jumps (decisions upon the control flow), where we model the variable timing. Furthermore, the processor may make use of bypass/operand forwarding, but potential bubbles are assumed to be included in the instruction timing, or otherwise avoided by the compiler; otherwise, an architectural way around this problem are interleaved pipelines [17]. Furthermore, there must be no out-of-order processing. Through that, structural and data hazards need not be modeled, but only control hazards at the granularity of basic blocks.

Cache Requirements. Processors may have instruction and data caches, but we do not allow for cache misses. That is, cache contents are selected at or prior to compile time and kept static during execution, or deterministically loaded by software at locations known at compile time. For example, caches could be loaded every time a function is entered (as in [54]), or explicitly through statements in the source code (scratchpad memory). Cache locking [45] provides an alternative mechanism to reach the same effect (which, incidentally, can improve the WCET [16]), and potentially qualifies many more processors for our approach. With these cache requirements, timing effects due to caches can be annotated in the source code as part of our analysis, and do not have to be modeled on instruction granularity.

On-Chip Bus Transfers. In many system-on-chip processors, there are peripherals (e.g,. a UART peripheral) which are accessed from the core via on-chip buses. Since usually there will be no model available for the behavior of the peripheral, we cannot support such accesses in a WCET analysis. A WCET-amenable processor therefore should provide time bounds for instructions performing such accesses. Consequently, no model is necessary for bus arbiters and peripheral states.

Multicore Processors. With the presence of hardware threads, there could be interference caused by resource sharing. In principle, there are techniques which provide some temporal isolation for the hardware threads, as described in [3]. We expect that a WCET-amenable processor has to implement such techniques, since otherwise the WCET problem becomes even more intractable than it already is for today's monoprocessors. Consequently, we are only considering monoprocessors here, assuming that an extension to multicore processors can build on such isolation, and otherwise capture high-level interactions such that they are visible in the source code.

Amenable Processors. Due to the complexity and variety of modern processor architectures, only a detailed review on a case-by-case basis allows to identify existing processors which fulfill our requirements. Therefore, we can only give a few select examples, and otherwise refer to our requirements for what we wish future processor architectures would look like, to keep WCET analysis tractable. Slight overapproximations, such as using only the maximum execution time for time-variable instruction, might be applied to use our approach even for processors that are not strictly compliant. The processor family that we target in this paper, the 8-bit Atmel AVR family [36], is a good example for that. While there is a dependency of the instruction timing on the operands in some cases – namely, slight variations in instruction timing for flash memory access – this is negligible and can be overapproximated. Our results shown later in this article justify this approach. Other processors that are a good fit are the SPARC V7 (specifically the ERC32 model [56]), the ARM7TDMI family [22], and the Analog Devices ADSP-21020. Academic examples include the Java-Optimized Processor [54], and the Precision Timed Architecture [39] with minor modifications (namely, port-based I/O to avoid time variances in load/store instructions, and absence of structural hazards).

2.1.2 WCET Analysis at Source Code Level

Our main goal is to shift WCET analysis from the instruction level to the source code level, where control flows are easier to follow and type information is available. We expect two profound effects from this shift. First, a model checker can leverage the clearly visible control flow and type information to perform an automatic path analysis, without requiring user inputs. That should lead to tighter and safer estimates than other approaches, such as ILP. Second, the complexity of the analysis is reduced, since operations are represented in a more high-level view.

A necessary prerequisite for such a source-level analysis is to annotate the source code with the instruction timing, as faithfully as possible. We do so by introducing a counter variable [30] into the source code, which is a global variable representing execution time that is incremented after each statement according to the time taken thereof (see Fig. 4b for an example).

Towards that, timing information needs to be extracted from the executable code, whilst considering the microarchitecture of the target processor, and then mapped back to the source code in the form of assignments to our counter variable at the correct locations. For a WCET-amenable processor, considering the microarchitecture boils down to model variable instruction timing only at branch points. The time annotations in the source code must therefore allow the encoding of such variable timing. In principle this is only possible precisely when the control flow of the instructions is also mapped back to the source code. That is, conditional jumps in the instructions must be lifted in the source code. Later, we show that some overapproximations can be applied, in case a back-mapping is not possible because a source equivalent is lacking for some instructions.

In general, mapping the instruction timing back to the source code means to establish a mapping from the instructions to source-level constructs. This can be difficult, because compiler optimization may produce a control flow in the executable that is very different from that of the source. For example, functions that are present at the source level may be absent in the executable because of inlining. Vice versa, new functions could be introduced in the executable which have no direct match in the source code. For instance, it is common for compilers to introduce loops and functions to compute 64-bit multiplications and shifts, or to implement switch case statements as binary search or lookup tables. These kinds of transformations make it hard to automatically map the basic blocks in the machine instructions to the source code. An ideal compiler would keep track of the mapping during the translation process, however, we are not aware of any compiler doing so. Therefore, some overapproximations must be applied to overcome these difficulties, and they can go a long way in automating the back mapping.

For the work in this paper, we have established a heuristic mapping for the AVR gcc compiler with only little difficulty, as explained later in Section 3.2. Other researchers have accomplished the same goal for different WCET-amenable processors [32, 52] even under some optimization, suggesting that such a mapping usually can be established when only considering the timing.

In the next section, we elaborate how Model Checking can be used on this time-annotated source code to estimate the WCET.

2.2 Model Checking

Model Checking [13] is a formal analysis technique used to verify a property on a model. Given a model and a property, a *model checker* – a tool implementing this technique – determines if the model satisfies the property in every possible execution. If the property does not hold, the model checker produces evidence for the violation, called a counterexample. Model checkers perform reachability analysis over finite-state transition systems, where the number of states is a product of program locations and variable valuations at these locations. Therefore, though Model Checking is sound and complete for finite-state models, scalability is often an issue for complex models.

It has been demonstrated that model checkers are useful at computing the WCET of simple programs [32, 37], but the scalability issue has not been addressed before. The idea is to take a program that is annotated with timing information, translate it into a model and use a model checker to verify the property "at program exit, execution time is always less than X", where X is a proposed WCET value. The model checker acts as an oracle, telling if there exists any execution that takes longer than this proposal. This process is repeated with changing proposals, until we find the lowest upper bound where no counterexample is generated. We will follow the same approach here, but address the scalability issue by minimizing the number of oracle queries, and also the complexity of the individual queries.

For our experiments we have used the model checker CBMC [12], due to its robustness and performance. It is a *bounded model checker* which accepts models and properties in the form of ANSI-C programs. Some important features of CBMC that we use in WCET computation are:

- **Assertions**: CBMC allows expressing properties with assertions. In our case, these assertions are of the form `assert(_time<X)`, where the constant `X` is the proposed WCET, and `_time` denotes the counter variable reflecting the time passing by in the program.
- **Non-determinism**: CBMC allows any of the program variables to be assigned a non-deterministic value. This is done using assignments of the form `y=nondet()`, where `nondet()` is an undefined function having the same return type as `y`. This results in `y` being assigned a non-deterministic value from the range specified by its type.
- **Assumptions**: CBMC allows the use of `assume` statements to block the analysis of undesirable paths in a program. A statement of the form `assume(c)` marks all those paths infeasible for which `c` evaluates to `false` at the execution of this statement. This feature can be used to constrain the value domain of non-deterministic assignments.
- **Checking multiple assertions**: CBMC allows multiple assertions in the input program, which can be checked at once using the `--all-properties` option. This option uses an optimal number of solver calls to verify programs with multiple assertions.

The technique we present here does not depend on CBMC specifically. It could be replaced by any other model checker, possibly through an additional front-end that translates C code into a model to work with (e.g., the CPROVER tools).

2.3 Program Slicing

Program slicing was first introduced by Mark Weiser in 1981 [57]. Given an imperative program and a slicing criterion, a program slicer uses data flow and control flow analysis to eliminate those parts of the program that do not impact the slicing criterion. That is, it removes statements which do not influence the control flow related to the criterion, and do not change its value. The resulting program is called a "program slice", and behaves identically w.r.t. the slicing criterion, but has a reduced complexity.

Slicing works by constructing and evaluating a Program Dependency Graph (PDG), which captures the control and data flow dependencies of a program. For example, Figure 2 shows the PDG corresponding to the code in Figure 4b. This graph has two kinds of edges to denote dependencies: dashed edges denote data dependence, and solid edges denote a control flow dependency. For example, the loop increment "i++" is control-dependent on the loop condition "i $<$ 35", and data-dependent on itself as well as on the loop initialization.

Given the PDG from Fig. 2 and a slicing criterion, we start at the node that corresponds to the location of the criterion, and traverse the PDG from this point until all the root nodes of the graph are reached. Subsequently, we remove from the program all statements and expressions that correspond to nodes that have not been visited during this traversal. For our example program in Fig. 4b, the criterion is the latest possible assignment to variable `_time` in line 17, and the corresponding PDG node is "_time += 5" (in the lower right corner of the graph). When traversing the graph from there, the outlined/red nodes (e.g., "out = acc/scl") are not reachable. Therefore, these parts of the program do not impact the value of variable `_time` at our location, and can be safely removed from the program. The resulting program slice is given in Fig. 4c.

2.4 Loop Acceleration

Loop acceleration describes the action of replacing a loop with a precise closed-form formula capturing the effects of the loop upon its termination. This has been shown to be effective for Model Checking of source code [14,6].

For loop acceleration to be applicable, the loop should have the following characteristics:

- The loop should iterate a fixed number of times, say n, which could either be a constant or a symbolic expression. For example, the loop may execute 10 times or n times, or $x * y$ times and so on.
- The statements in the loop constitute only of assignments and linear arithmetic operators, that is, first order polynomial computations (and no higher order).
- The loop body consist of straight line code, that is, there are no branching statements such as if-else statements inside the loop body.

When a loop satisfies the above constraints, it is possible to replace the loop with simple linear recurrence relations that precisely compute the summarized effect of all the iterations of loop on each of the assignments in the loop body. For example, the for-loop in Figure 4c is replaced with the block spanning lines 7 through 13 in Figure 4d.

2.5 Abstraction

Abstraction is a term used to describe a modification to an input program in such a way that the resulting output program allows more runs than the input program. That is, the set of all possible program states at the end of the output program, is a superset of all possible states at the end of the input program. In other words, abstraction creates a safe overapproximation of a program.

For instance, in Section 2.4, suppose a loop has only the first two characteristics, but violates the last one, viz., the loop does have branching statements in the form of if-else statements. Then, it is still possible to replace the loop with an over-approximate formula. The most simple way to achieve this is to replace it with a non-deterministic assignment that allows for the computation of the result in any arbitrary value. For example, the effect of the loop in Figure 5a on the variable `ans` is abstracted as shown in line 5 of Figure 5b. There we allow `ans` to take on any non-deterministic value in its type range, which is a superset of all the feasible values among all possible executions.

3 Finding the Worst-Case Execution Time

Our tool set TIC estimates the WCET at source level, with the help of timing information that is extracted from the executable code. The overall workflow is illustrated in Fig. 3 and shall now be explained in detail. Given a C program, a corresponding executable that results from cross-compilation for the target, and information about the target architecture, TIC does the following:

1. Estimates the time taken for each basic block in the machine code of the program.
2. Establishes a mapping from these basic blocks to a set of source lines and instruments the C program with the timing obtained in the previous steps.
3. Detects and prepares the handling of persistent variables, i.e., the internal states of the program, since those may have an impact on the control flow and thus on the WCET.

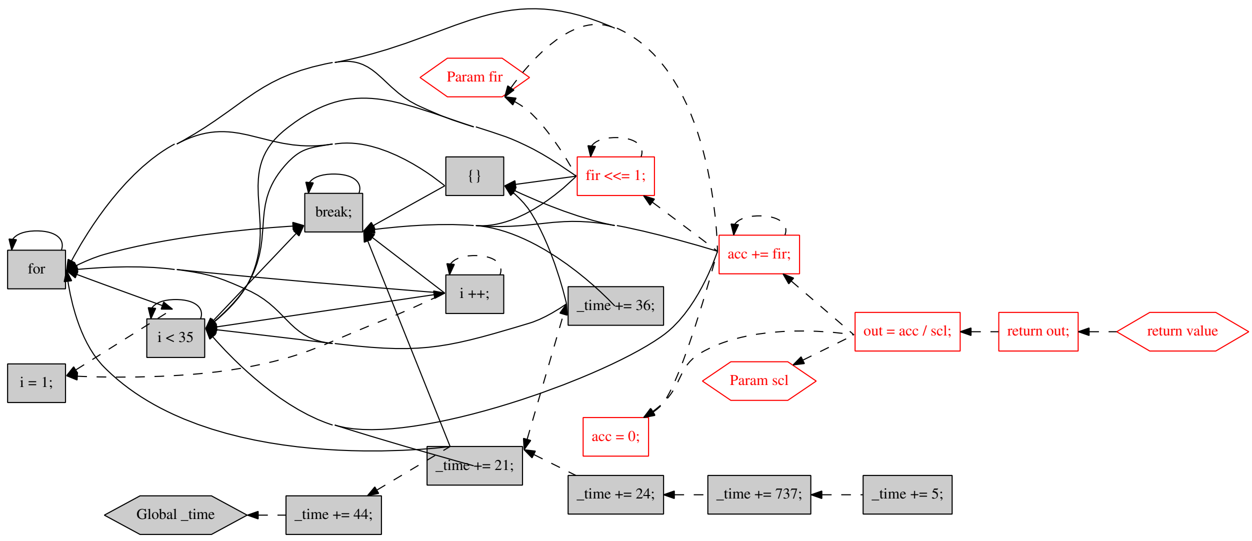


Fig. 2 Program Dependence Graph of the program from Fig. 4c. Sliced statements are outlined/red

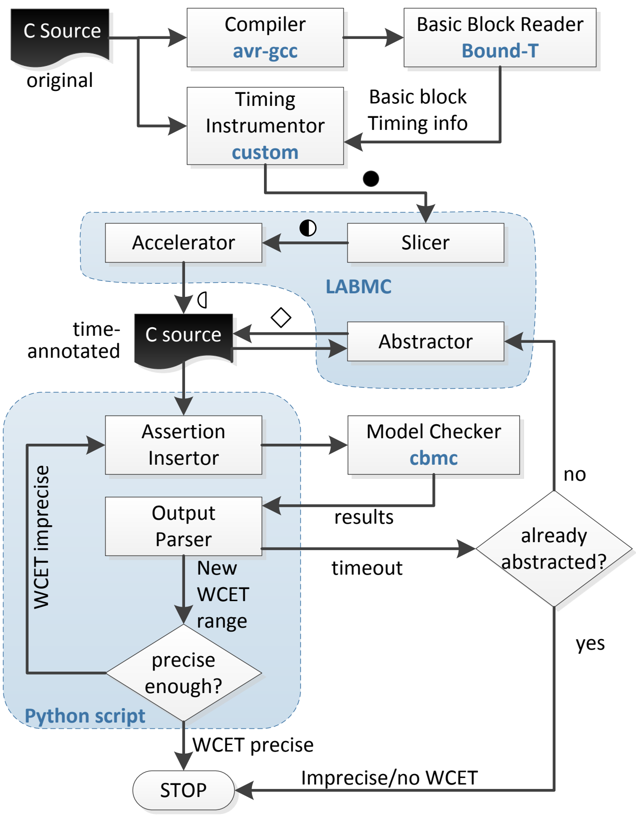


Fig. 3 WCET analysis using Model Checking

4. Applies source code transformations to eliminate all irrelevant data computations and to summarize loops.
5. Adds a driver function which represents the entry point for the model checker and encodes the search for WCET as a property.
6. Uses a model checker and an appropriate search strategy to estimate the WCET with desired precision.
7. If the model checker does not scale, then it abstracts the program and repeats the search.

The details of each of the above steps are given in the following sections. As a running example, we use a simplified version of the *fir* benchmark from the Mälardalen WCET benchmark suite.

3.1 Estimation of Basic Block Time (●)

We first analyze the executable file produced by the target-specific compiler to construct a control flow graph. Towards that, we identify *basic blocks*. These are maximal sequences of instructions that are free of branches, i.e., where the only entry point is the first instruction and the only exit point is the last instruction. Therefore, the basic blocks become the nodes in our control flow graph, and the edges represent branches or function calls, labelled with their branching condition. Finally, we determine the execution time for each basic block by adding up the time taken by the contained instructions, and annotate our nodes with this block timing. Since we allow for branch instructions to have a variable timing (e.g., the instruction may take longer if a conditional branch is taken), we annotate the precise timing of the jump instructions to the edges of the graph. Through that, there is no overestimation due to such timing variations.

Our implementation re-uses parts of the Bound-T WCET analyzer [31] to build the control flow graph and estimate basic block times. We therefore have exactly the same inputs for our approach and the ILP-based analysis in Bound-T. Alternatively, a more generic binary analysis library could be used, such as the *Binary Analysis Platform* [9].

3.2 Back-Mapping and Annotation of Timing

The next task is to match the control flow of the machine instructions with the control flow of the source code, and back-annotate the basic block timing in the source code, in the form of increments to a counter variable. The result shall be a source code that is instrumented with timing, in the following called *instrumented program*.

Specifically, here we annotate the basic blocks of the source with the instruction timing as close as possible to the corresponding source construct. That is, each maximal sequence of statements that is free of branches (just like basic blocks at machine code level) is immediately followed (or preceded, in the case of loop headers) by a timing annotation that reflects the execution time of the instructions corresponding to that block.

First of all, an approximate and incomplete mapping of machine instructions to source line numbers is given by GNU's *addr2line* tool. However, it is incomplete because some instructions do not have a direct match in the source code, and it is only approximate because it does not provide column numbers. Complex expressions falling into multiple basic blocks, such as loop headers or calls in compound expressions, cannot be resolved with this information. We therefore use this information as an initial mapping, and then apply safe heuristics to complete the picture.

For the heuristics that manage the back-annotation of the basic block (BB) timing information according to source code, there are two cases to handle:

1. *One-to-Many mapping.* One basic block in the executable corresponds to one or more continuous expressions in the source code. This case is trivial to handle, all that needs to be done is instrument the basic block before the last expression with the corresponding timing information.
2. *Many-to-one mapping.* In these cases, several basic blocks in the executable map to a single source expression. Typically, this case arises when a source expression splits into different control paths in the machine instructions, for example in case of a division or a bit shift operation being converted into loops. In such cases, it is hard to instrument the source code with this information, since it would require translating back the loop into the source first. To tackle this issue, we summarize the timing information from such multi-path instruction blocks, and instrument the source code with its worst-case timing value.

As an example, the timing of the source code in Figure 4a was mapped as follows:

The table contains timing information of each basic block along with the span of the block in the source code. This information is used to instrument the source code as shown in Figure 4b: We introduce a global counter variable `_time`, which is incremented through macro `TIC` by the corresponding amount of time at the respective source locations.

Table 1 Mapping of basic blocks (BB) to souce code

Start line	End line	BB (#)	Time	Comment
1	3	1	44	including for init
3	3	2	21	for conditional
3	6	3	36	including for iter
7	7	4	24	before div
7	7	mult.	737	div block
7	18	6	5	after div

Basic block 2 is an instance of a one-to-many mapping. This block maps to the conditional part of the for-statement on line 3 of Figure 4a, and therefore the annotation is inserted just before the statement on line 8 in Figure 4b. Similarly, basic blocks 1 and 3 are one-to-many mappings. Basic block 1 implements the start of the function line 1, as well as the declarations and initialization on line 2 and also the for-initialization block on line 3. All these source level lines fall into a single source basic block. Similarly, the for-loop increment on line 3 along with statements on lines 4 and 5 map to basic block 3. The instrumentation in this case is placed in lines 6 and 12 respectively, in Fig. 4b.

The division assignment in line 7 is an instance of a many-to-one mapping, i.e., many basic blocks, and therefore also some conditional jumps, map to this single statement. Here, these basic blocks include a compiler-inserted function call (for the long division). In particular, the compiler-inserted function contains a loop, for which we do not compute the timing precisely, but instead we over-approximate this loop with its worst-case bound, and subsequently we use a single timing annotation for the entire function. Thereafter, the statement at line 7 can be represented by three parts; one before the division (function call), one for division (function WCET) and the final one after the call. The resulting three timing annotations are shown on lines 14, 16 and 17 Figure 4b.

This concludes the shift of WCET analysis from machine code level to source level. At this point, we have an *instrumented program*, i.e., a source code carrying the execution timing, ready to be analyzed for WCET. From now on, we continue the analysis with the instrumented source code only.

3.3 Handling of Persistent Variables

A WCET estimation of a function f (including all its direct and indirect callees) must consider all possible inputs to f, and from those derive the longest feasible paths. Such inputs can be function parameters, but also referenced variables that are persistent between successive calls of f (they can also be seen as the hidden program state). In the C language, such persistent variables are *static* and *global* variables that are referenced by f or its callees.

```
1  int task(int fir, int scl) {
2    int i, out, acc=0;
3    for(i = 1 ; i < 35; i++) {
4        acc += fir;
5        fir <<=1;
6    }
7    out = acc/scl;
8
9
10
11
12
13
14
15
16
17
18   return out;
19 }
```

(a) Original code

```
1  #define TIC(t) (_time += (t))
2  unsigned long _time = 0;
3
4  int task(int fir, int scl) {
5    int i, out, acc=0;
6    TIC(44);    // BB1
7    for(i = 1 ;
8        TIC(21),    // BB2
9        i < 35; i++) {
10       acc += fir;
11       fir <<=1;
12       TIC(36);    // BB3
13   }
14   TIC(24);    // BB4
15   out = acc/scl;
16   TIC(737);  // mult. _divmodsi4
17   TIC(5);     // BB6
18   return out;
19 }
```

(b) Instrumented

```
1  #define TIC(t) (_time += (t))
2  unsigned long _time = 0;
3
4  int task(int fir, int scl) {
5    int i, out, acc=0;
6    TIC(44);
7    for(i = 1 ;
8        TIC(21),
9        i < 35; i++) {
10       acc += fir;
11       fir <<= 1;
12       TIC(36);
13   }
14   TIC(24);
15    out = acc/scl;
16   TIC(737);
17   TIC(5);
18   return out;
19 }
```

(c) Sliced

```
1  #define TIC(t) (_time += (t))
2  unsigned long _time = 0;
3
4  void task(void) {
5    int i, _k0, _k1;
6    TIC(44);
7    {
8      _k0 = 1;
9      i = 35;
10     _k1 = i - _k0;
11     TIC(21*_k1 + 36*_k1);
12     TIC(21);
13   }
14   TIC(24);
15
16   TIC(737);
17   TIC(5);
18
19 }
```

(d) Accelerated

Fig. 4 Example for source code instrumentation and transformations to compute the WCET of a function `task`

In this work we over-approximate the content of such persistent variables by initializing them with a non-deterministic value, as explained in Section 2.2. This guarantees that all the feasible and infeasible values of the persistent variables, as allowed by their data type size, are considered as inputs to f. Thus, the WCET estimate for f is always a safe over-approximation of the maximum observable execution time of f. It is possible to remove (some of) the infeasible values either with manual inputs by users or by analyzing the callees of f. This may lead to a tighter WCET, but we did not explore these approaches as they are orthogonal to our main work.

3.4 Source Transformation Techniques

The source code is now instrumented with its timing behavior, and ready to be analyzed for WCET. However, a direct application of Model Checking to compute the WCET of large and complex programs would not scale due to the size of the generated model. The analysis time could quickly reach several hours for seemingly small programs, and memory requirements may also quickly exceed the available resources. Our next step, therefore, are source code transformations which retain the timing behavior, but reduce the program complexity. This can be done effectively thanks to the additional information available in the source code, such as data types and clearly visible control flows. The transformations are executed sequentially in *stages*, which we explain in the following. All three states require only little computational effort themselves, and therefore speed up the overall process of WCET estimation.

3.4.1 Stage 1: Slicing (◐)

Slicing reduces the size of the program by removing all statements that do not influence a certain criterion, as explained in Section 2.3. For the specific case of WCET analysis using a counter variable, our slicing criterion is the value of this variable upon program termination, through which statements not impacting the counter variable are eliminated, and WCET estimation becomes less complex.

As an example, consider again the instrumented program in Figure 4b. Firstly, observe that line 15 is not needed to compute timing, since the variable `_time` has no data or control dependency on the variable `out`. Similarly, lines 10 and 11 do not impact timing computation and can be sliced away. The sliced source for this example is shown in Figure 4c.

We used a program slicer that builds an inter-procedural program dependency graph [47], capturing both intra-procedural and inter-procedural data and control dependencies. It then runs one pass over this graph to identify statements that impact the property to be verified and outputs the sliced code. The slicer is a conservative one, which means that it discards statements only when sure that the statements do not affect the timing analysis. In all other cases, the statements are preserved.

3.4.2 Stage 2: Loop Acceleration (☾)

A major scalability obstacle in Model Checking are loop-like constructs, since they have to be unrolled before analysis. They can therefore increase the program complexity significantly. This problem can be solved by applying loop acceleration, as described in Section 2.4. The resulting program will have a reduced number of loops, and therefore exhibit a reduced complexity.

As an example, consider the loop in Figure 4c: Here, `_time` is incremented in line 12 within a loop body through `TIC`. The effect of this repeated increment of `_time`, can be summarized by the expression `_time` = `_time` + $n \cdot 36$, where n is the number of loop iterations. Line 11 in Figure 4d shows the accelerated assignment. Note that two new variables `_k0` and `_k1` have been introduced, representing the initial value of the loop counter (`_k0`) and the number of loop iterations (`_k1`). After accelerating all variables, the loop in Figure 4c can be removed and replaced by the statements given in Figure 4d lines 7 through 13. Here, lines 8 to 11 represent the

```
 1 for (i=1 ;
 2       TIC(20), i<=len ;
 3       TIC(8), i++) {
 4    TIC(35);
 5    if(ans & 0x8000) {
 6       TIC(31);
 7       ans ^= 4129;
 8    } else {
 9       TIC(23);
10       ans <<= 1;
11    }
12 }
```

(a) Branch inside loop

```
 1 int _k0, _k1;
 2 _k0 = 1;
 3 i = len + 1;
 4 _k1 = i - _k0;
 5 ans = nondet();
 6 TIC(_k1*(20+8+35)+_k1*31);
 7 TIC(20);
 8
 9
10
11
12
```

(b) Abstracted loop

Fig. 5 Loop abstraction using LABMC

effect of all the 34 iterations of the loop. Line 12 captures the time taken for evaluating the loop condition after the final iteration.

As the accelerated program of Fig. 4d is free of (most) loops, it is less complex than the instrumented program. For this example, the program size (as determined by CBMC) reduces from 325 steps for the instrumented program to only 60 steps for the accelerated program. Last but not least, note that a prior slicing is important for loop acceleration; if we had not sliced the instrumented program w.r.t. `_time`, then the above loop could not have been replaced, as the assignment to `acc` on line 4 of cannot be accelerated due to the assignment to `fir` on line 5.

TIC implements acceleration as proposed in [14], mainly as they are shown to be effective on SVCOMP benchmark C programs and industrial programs [6].

3.4.3 Stage 3: Loop Abstraction (◇)

Finally, further reduction of complexity at the cost of precision can be accomplished by abstraction, as described in Section 2.5. However, we tailor the abstractions by including some specific time-approximations that preserve the WCET estimate and reduce complexity even further.

Abstraction is used to get rid of loops that could not be accelerated, such as the loop shown in Figure 5a: The if-condition on line 5 depends on the variable `ans`, which is updated both in the *then*-branch (line 7) and *else*-branch (line 10). These assignments determine how many times the *then* and *else*-branches would be executed. As these branches depend on values updated within the loop, we cannot determine the number of times this branch would be executed, and hence we cannot accelerate the loop.

The abstracted version of the loop in Figure 5a is shown in Figure 5b. First, we introduce variables `_k0` and `_k1` to capture the initial value of the loop counter (line 2) and the value of it upon loop termination (line 4). Then, the variable `ans` is assigned a non-deterministic value (line 5). This assignment, as explained in Section 2.2, allows `ans` to take any value in the entire range of its data type, `int`. While in the original program `ans` may take only a subset of `int` values, we now allow it to take on any `int` value, and thus have constructed an abstraction.

After this abstraction, the loop can now be accelerated as explained before. Line 6 contains the accelerated assignment to `_time`. In this, `_k1*(20+8+35)` accounts for the time increments in Figure 5a corresponding to the loop condition evaluation (line 2), loop counter increment (line 3) and the first part of the loop body on line 4. Finally, line 7 captures the time taken for evaluating the loop condition after the final iteration.

Time Approximations. The trailing `_k1*31` in Fig. 5b on line 6 summarizes the time taken by the if-statement forming the remainder of the loop body, but it is somewhat special. It contains a WCET-specific modification to the abstraction as explained in Section 2.5: Since now the values of `ans` have been abstracted, we can no longer reason about which branch of the if-statement is taken in the WCET case. Therefore, when abstracting such undecidable conditional statements, we over-approximate the effects on our time variable by only considering the longest branch (here: the then-branch in line 6 with 31 clock cycles). Note that this does not change the WCET estimate, because the model checker would have picked the longer branch anyway, since the possible values of `ans` include the case allowing the then-branch to be taken. It is thus a safe modification to the abstraction in the context of WCET analysis, which however reduces the complexity of the program further.

If a loop contains unstructured code, such as `break` and `return` statements, these are easily handled through a non-deterministic Boolean variable that allows these statements to be executed or not executed.

To implement abstraction, we again used the LABMC tool [14]. However, the standard abstraction has been tailored for the counter variable `_time`, as explained above, to pick the maximum time increment from the branches.

3.4.4 Further transformations

A number of other source transformations could be applied to further reduce the program complexity and yet retain the worst-case timing information. For example, after the back-annotation of timing, the counter variable could be summarized by merging increments belonging to the same source block. Or, live variable analysis and loop folding could be used to reduce the unwinding depth of a program [4]. However, all of this would make it harder to understand how source-level statements contribute to timing, and thus has not been investigated in the context of this work.

Furthermore, CBMC itself (more specifically, goto-instrument, a front-end of CBMC), features transformations that can be applied to the source program, such as k-induction for loops, constant propagation and inlining. While our toolchain supports using those CBMC features, none of them have proven effective in reducing the complexity or analysis time in our WCET framework. In particular, the loop acceleration included there takes longer than our entire WCET analysis and does not improve the analysis time thereafter, and "full slicing" even produces unsound results. Therefore, our source transformations introduced above are justified, because they have a proven impact on WCET analysis.

3.5 Adding a Driver Function

To run a model checker on the instrumented (sliced, accelerated, abstracted) source code, we add a *driver* function to the source that implements the following operations in sequence:

1. Initialize counter variable `_time` to zero.
2. Initialize all input variables of the program to nondeterministic values according to their data type. This includes handling of persistent variables as described in Section 3.3.
3. Call function f for which WCET shall be estimated.
4. Encode assertion properties to query for WCET (details in Section 3.6).

The driver function is handed over as entry point to the model checker.

3.6 Determining WCET

At this point, a model checker such as CBMC can verify whether the counter variable `_time` always carries a value less than X after the program terminates. In this section we explain how to choose candidates for X, such that we eventually approach the WCET.

A WCET candidate X is encoded as `assert(_time <= X)`, and subsequently passed to the model checker. Unless the model checker runs into a timeout or out of memory, only two outcomes are possible:

1. Successfully verified, i.e., `_time` can never exceed X. Therefore, X is a valid upper bound for the WCET.
2. Verification failed, i.e., `_time` may exceed X in some executions. Therefore, X is a lower bound for WCET. If a counterexample was generated, then it may contain a value $Y > X$, which then is a tighter lower bound for the WCET.

Our strategy is to use both outcomes to narrow down on the WCET value from both sides: Initially, we start with lower bound e_{lower} as zero, and upper bound e_{upper} as a very large value (*intmax*). We now place a number of assertions[1] in the program, where each is querying one WCET candidate X. In particular, the candidates are equidistantly spaced between e_{lower} and e_{upper}; except for the first step, where we use a logarithmic spacing to initially find the correct order of magnitude. Subsequently, we invoke the model checker to verify the assertions – in the case of CBMC, all at once. For each assertion we obtain a result. We set e_{lower} as the largest X where the assertion was failing (or, when a counterexample with $Y > X$ was generated, to Y), and e_{upper} as the smallest X where the assertion was successfully verified. The search is now repeated with the new bounds, and stopped when these upper and lower bounds are close enough to each other, which can be interpreted as a *precision goal* for the WCET estimate. The full algorithm is given in Algorithm 1.

Algorithm 1: Iterative search for WCET bound

```
Input: instrumented C source code C, required precision P
Output: WCET estimate e_upper, s.t. e_upper − e_lower < P.
begin
    N_assert ← 10                 // number of assert per call
    e_lower ← 0
    e_upper ← intmax
    p = e_upper − e_lower
    while p > P and not timeout do
        if e_upper = intmax then
            candidates ← logspace(e_lower..e_upper, N_assert)
        else
            candidates ← linspace(e_lower..e_upper, N_assert)
        C′ ← insert asserts for candidates into C
        results ← model checker (C′)
        for i = 1 to N_assert do
            if verified(results[i]) then
                e_upper ← min(e_upper, candidates[i])
            else
                B ← getCounterexample(results[i])
                e_lower ← max(e_lower, candidates[i], B)
        p = e_upper − e_lower
```

At any point in time the model checker could terminate due to either a user-defined timeout or when it runs out of memory. In such cases the algorithm returns the WCET estimate as at-this-point tightest bound e_{upper} that could be verified. In combination with the precision goal P, this gives the user a fine control over how much effort shall be spent on computing the WCET. For example, an imprecise and fast estimate may be sufficient during early development, whereas a precise analysis may be only of interest when the WCET estimate approaches a certain time budget.

The maximum number of search iterations can be determined in advance; in the worst case the number of search iterations n is

$$n = \left\lceil \log_{N_{\text{assert}}}\left(\frac{e_{\text{upper}} - e_{\text{lower}}}{P}\right) \right\rceil, \tag{1}$$

[1] We have empirically chosen N_{assert}=10; placing either more or less assertions usually take longer, because either the computational effort is growing, or more iterations are required.

where $e_{\text{upper}} = intmax$ and $e_{\text{lower}} = 0$, if no a-priori knowledge about the bounds of WCET is available. Usually the number of iterations is lower, since the values found in the counterexamples speed up the convergence (point 4 in Alg. 1).

Leveraging A-Priori Knowledge. In this article we assume that no information about the timing behavior of the program is available. If, however, the user has some knowledge on the WCET bounds already, then these bounds can be tightened by the algorithm, reducing the number of iterations. If an upper bound is known, then e_{upper} can be initialized with that bound, and the algorithm tightens it up to the required precision. Similarly, if a lower bound is known from measuring the worst-case execution time on the target (e.g., from a high watermark), then e_{lower} can be set to that measured value.

Implementation. The WCET search procedure was implemented as a Python script. It further implements monitoring of memory and CPU time as given in Table 4. As model checker we used CBMC with various solver backends. However, other model checkers, such as *cpachecker* could be used as alternatives with only little changes.

3.6.1 Target-Specific Analysis

Since the analysis takes place at source level, it is essential to include target-specific information, such as word with, endianness, interrupts, I/O facilities etc. If neglected, the behavior between the model in the analysis and the real target may differ, leading to unsafe results. Most importantly, we provide target-specific preprocessor definitions and the word widths with the corresponding flags that CBMC offers. For more details on how to include target-specific information for the model checker, we refer the reader to [4], where the specific pitfalls for CBMC have been explained. We further employ checks during the WCET path reconstruction, which can identify missing or incorrect target information.

3.7 Determining BCET

Our tool set can also be used to compute the Best-Case Execution Time (BCET), with minor modifications. Such a lower bound of execution time may be of interest for running a schedulability analysis of event-driven tasks, for example, interrupts. Schedulability analysis then requires a *minimum inter-arrival time* (MINT), i.e., the shortest possible time between two consecutive releases, to bound the processing load that is generated by the task. If a software has no inherent mechanism to limit the MINT of an event-driven task, then the BCET can be computed and used in place of MINT.

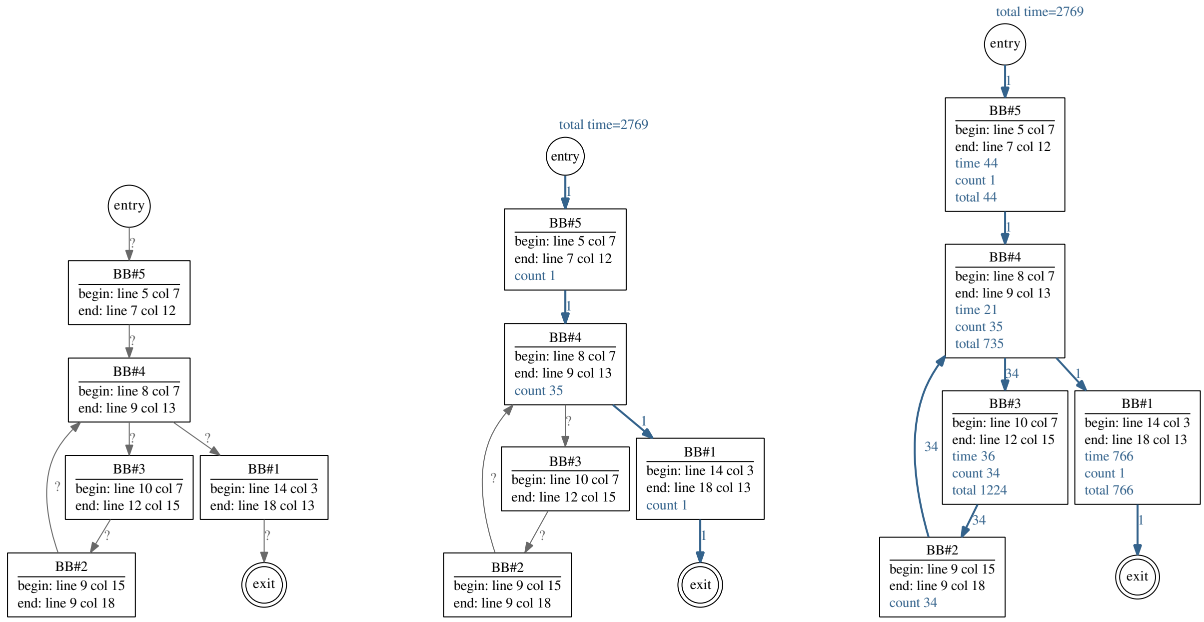
4 Reconstructing the WCET Trace

From a software developer's point of view, the WCET value in itself (and its inputs) are of limited use. Without understanding the exact path and the decision variables leading to the WCET, counteractive measures are limited to a time-consuming trial-and-error approach. Instead, the following information would be of high value for comprehension and proactive mitigation of the WCET case:

1. The exact inputs leading to the WCET path,
2. a concrete walk-through of the worst-case path where all variables can be inspected, to identify the timing drivers, and
3. a timing profile of the worst-case path (how much time was spent where?).

Especially a detailed walk-through of the worst-case path is very important to understand why a certain path incurs high timing cost. From our experience, it is not sufficient to know only the WCET path. Oftentimes the specific values of decision variables are important to understand why this path was taken, but such information is not provided by any tool that we know of. The user is therefore left with the mental puzzle of finding an explanation how the presented path came to be, and at times there can be very subtle and unintuitive reasons. For example, in our *prime* benchmark, we found that an integer overflow in a loop condition caused the loop to terminate only after a multiple of its explicit bound. In turn, this overflow depends on the word width of the processor being used (more details about this case are given in Section 5.3). In such a case a developer would most likely struggle to explain why the loop did not obey to its bounds, and thus not understand the WCET estimate. In summary, the WCET trace that we want to reconstruct shall be detailed enough to inspect not only the control flow and its timing, but also all data that is relevant for the specific path being taken. Towards that, we want to leverage the final counterexample from Alg. 1 to provide a maximum-detail explanation of how the WCET can be reached.

The challenge in reconstructing the WCET path from a counterexample is illustrated in Figure 6, for the same program from Fig. 4b introduced earlier. The goal is to annotate the corresponding control flow graph in Fig. 6a with execution counts at both its nodes (basic blocks in source code) and its edges (branches being taken). However, the counterexample produced by the model checker only contains sparse information as shown in Fig. 6b. Typically, it only provides a subset of the visited code locations, variable assignments that are relevant for branches being taken, and assignments to `_time` oftentimes occur only at the end of the counterexample, depending on which solver backend was chosen. In fact, the counterexample is only guaranteed to provide exactly one value for the variable `_time`, which is at the location where the WCET assertion is failing.



(a) Control flow graph (b) Typical output of model checker (c) Fully reconstructed path

Fig. 6 Reconstruction problem of WCET path for example code from Figure 4a

It is clear that the counterexample provided by the model checker is insufficient as an explanation for the WCET path, much less for values of arbitrary variables. It could carry even less information than what is provided by an ILP-based approach, where execution counts for all basic blocks on the WCET path are available. Without such data, neither can we compute a timing profile in the presence of function calls, nor is it possible for the developer to understand or even walk through the WCET path.

Therefore, towards reconstruction of the WCET path, we have to interpolate the control flow in between the locations given in the counterexample, and deduce variable valuations that are not available (most importantly, variable `_time`). There are two fundamental approaches for reconstructing the path:

1. **By Analysis:** Use SMT, AI, or a similar technique to fill the "location gaps" of the counterexample, and to conclude about assignments of all (possibly sliced) variables. This is expected to be computationally complex and not precise.
2. **By Execution:** Execute the code with the worst-case inputs. An "injection" of critical values beyond inputs is required, for example to all persistent variables. This could be compiled into the application through look-up tables that contain the known assignments from the counterexample, or done dynamically during execution.

It should be clear that an execution is preferable in terms of precision and speed, however, there are some challenges in such an approach:

1. **Using an Interpreter:** Whereas the most logical choice, only few good interpreters available for the C language. Most of them have limited "stepping" support (e.g., *cling* uses JIT; which means stepping would inline/flattening functions), and in general they are slow. There would be no support for target-specific code, such as direct pointer dereferencing. Often only a subset of the language is implemented.
2. **Instrumented Execution:** Compile and run the application, preferably on the target for maximum fidelity, while capturing an execution trace which subsequently could be replayed. Capturing a complete trace including variable valuations could produce a huge amount of data and incur memory and timing overhead. If the program under analysis is supposed to run on a different target (cross-compilation), then it might not be feasible to capture the trace for reasons of memory limitations or missing interfaces.

We decided for an execution-based approach, since this is computationally less complex than analysis, and thus expected to be faster. The problem of insufficiently granular interpretation and trace capturing has been addressed as follows: Our path reconstruction is accomplished by means of executing the application in a debugger, whilst injecting variable valuations from the counterexample when necessary. By choosing a debugger as replay framework, replaying the WCET has the potential to become intuitive to most developers, and thus could be seamlessly integrated into the software development process. Furthermore, debuggers are read-

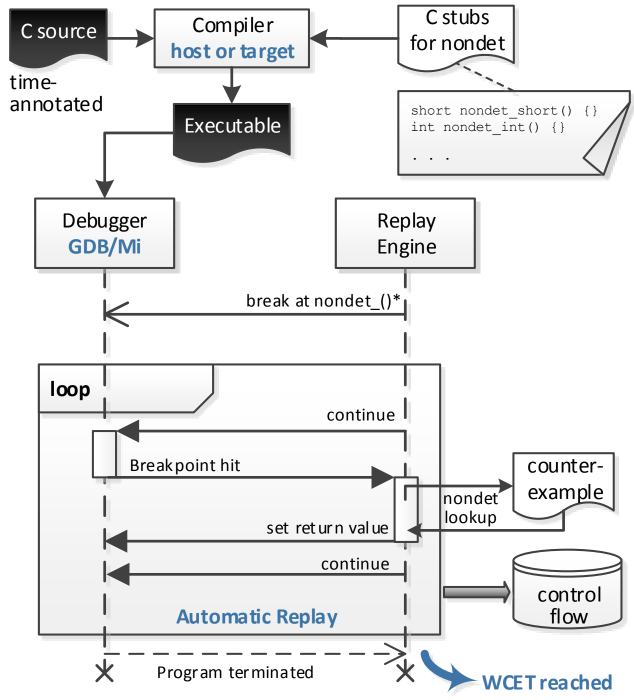


Fig. 7 Automatic WCET replay using a debugger

ily available for most targets, and some processor simulators even can be debugged like a real target (in our case, *simulavr* allows this). And, finally, word widths, endianness etc, are bound to be correct, in contrast to any approach based on an interpreter. However, to use these advantages to full capacity, the replay process must be of low overhead, and able to be fully automated to not require any additional inputs compared to traditional debugging.

4.1 Preparing Replay

Our proposed replay process is illustrated in Figure 7. We start by compiling the instrumented (or sliced, accelerated, abstracted) program with additional stubs (empty functions) for each "nondet" function (see section 2.2). Then, we load the program into a debugger. Depending on which compiler was chosen, this could either be the host's debugger, or the one for the target (connecting to either an actual target or to a simulator). Since in all cases the replay process is similar, for the rest of this paper we do not distinguish anymore between a host-based replay, a simulator or a real target.

Finally, it should be mentioned that the replay path is based on the time instrumentation in the sources, which is why the path always exists unless the model checker is handed wrong target-specific settings (e.g., word widths), or unless the analysis is unsound.

4.2 Injection of Nondeterminism

To inject the critical variables as found by the model checker, we set breakpoints on the inserted nondet stubs and start the execution. As soon as the debugger reaches one of our breakpoints (additional user breakpoints for stepping may exist and are not interfering), we automatically inject the correct value as follows: First, we query the current call stack to identify the caller, i.e., the source location of the nondet call. Knowing the location of the designated nondet assignment, the value that leads to WCET is extracted from the counterexample, and subsequently forced as a return value in the debugger. As an effect, the call to the empty nondet stub returns the critical value suggested by the model checker. After that, the execution is resumed automatically.

However, there exists a second source of non-determinism, besides the explicit "nondet" function calls. In the C language, every uninitialized variable that is not static nor at file scope, initially carries an unspecified value. Therefore, every such uninitialized local variable is considered by the model checker as non-deterministic input, as well. Since no explicit "nondet" function calls exist, the breakpoints inserted earlier do not allow us to inject values into those variables. As a solution, we first identify all uninitialized local variables from the parse tree of the C source (declarations without right-hand sides, mallocs, etc.), and then insert additional breakpoints for every such variable that is mentioned in the counterexample. Through this, injecting values into uninitialized variables is handled in the same way as the nondet function calls (not shown in the Fig. 7 for clarity).

With this technique, the injection of the critical variables from the counterexample is accomplished without any user interaction, and without embedding the assignments in the program itself (no memory overhead). Furthermore, this live injection allows for additional checks during the execution, such as matching the assignment with the actual call location, and ensuring that the execution path does not deviate from the analysis.

4.3 Identification of WCET-Irrelevant Variables

The valuations of some variables do not have an effect on the control flow, and thus do not influence the timing[2]. As explained before, such variables are identified and sliced away during the analysis phase. In particular, both our preprocessing (slicing, acceleration, abstraction), as well as the model checker itself remove such variables. Consequently, the counterexample does not include valuations for variables that have been found irrelevant for the WCET.

As an effect, any location having a non-deterministic assignment that is visited during replay *and* and does not have

[2] Note that this only holds true for cache-less processors.

Table 2 Timing profile obtained from WCET path of *adpcm decode* benchmark

%total	cycles	%self	cycles	calls	self/call	total/call	name
100.0	69,673	35.3	24,593	1	24,593	69,673	decode
13.7	9,522	13.7	9,522	2	4,761	4,761	upzero
13.2	9,168	13.2	9,168	2	4,584	4,584	uppol2
12.9	8,984	12.9	8,984	2	4,492	4,492	filtez
9.3	6,472	9.3	6,472	2	3,236	3,236	uppol1
5.5	3,854	5.5	3,854	2	1,927	1,927	filtep
5.1	3,576	5.1	3,576	2	1,788	1,788	scalel
2.5	1,758	2.5	1,758	1	1,758	1,758	logscl
2.5	1,746	2.5	1,746	1	1,746	1,746	logsch

an equivalent assignment in the counterexample, indicates that the respective assignment is irrelevant for WCET. We highlight such irrelevant statements to the developer, to help focus on the drivers for the worst-case timing, and not get distracted by surrounding code.

4.4 Collecting a Timing Profile

When larger programs are being analyzed, it may quickly become impractical to step through the whole path to find the main drivers of the worst-case execution time. To help the developer identify interesting segments that need to be stepped through in the debugger, we also generate a timing profile of the WCET path, showing which location was visited how often, and how much time was spent there.

Towards that, we capture the complete control flow on the fly during the replay. Since the timing profile is especially useful for larger programs, capturing the control flow during the debugger execution must scale well with growing path length and thus cannot be realized by a slow step-by-step execution in the debugger. Instead, we set a hardware *watchpoint* on our counter variable. That is, every time this variable is modified, the debugger pauses execution and notifies the replay engine of the new variable content. Since hardware watchpoints are realized through exceptions, the debugger can run without polling and interruptions between the watchpoints, and therefore the control flow is captured with very little additional delay. Considering that the counter variable is embedded at least once per source block, the sequence of all reached watchpoints (their valuation and location), represents the actual control flow in the source code. As a result, a timing profile similar to the outputs of the well-known tools *gprof* or *callgrind* can be reconstructed and intuitively used by the developer. Table 2 shows the resulting flat WCET timing profile for the *adpcm decode* benchmark. Note that additionally to the shown per-function metrics, the execution times are also available at the even finer granularity of source blocks, which helps pinpointing the timing bottlenecks to specific source blocks within the functions.

5 Experiments

We applied TIC to the Mälardalen WCET Benchmark Suite [24] to evaluate the performance and the tightness of WCET estimates computed with TIC. As a target, we used the Atmel ATmega 128 [36], for which WCET analyzers (Bound-T [31]) and simulators (simulavr) are freely available and can be used as a baseline for evaluating TIC. This target satisfies our requirements for WCET-amenable processors, since there is practically no timing dependence on the operands.

The complete set of experimental data (instrumented, sliced, accelerated and abstracted sources, as well as complete traces of the WCET search) is available at `https://github.com/TRDDC-TUM/wcet-benchmarks`.

5.1 Setup and Reference Data

Selected Benchmarks. We selected a representative subset consisting of 17 Mälardalen WCET benchmarks, such that all program properties, e.g., multi-path flows, nested loops, arrays and bit operations were covered, except for recursion and unstructured code (they cannot be handled by the basic block extractor, yet), and floating point variables (cannot be handled by the timing instrumentor, yet). However, these missing properties in principle can be addressed; this is not a limitation of our approach.

Host Platform. We conducted our experiments on a 64bit machine with a 2.7GHz Intel Xeon E5-2680 processor and 16GB RAM, using CBMC 5.6 as model checker. As CBMC's backend we have used a portfolio of solvers, consisting of minisat (built-in), mathsat (v5.3.10/GMP), cvc4 (v.1.5pre/-GLPK-cut), z3 (v.4.5.1) and yices (v.2.5.1/GMP). We stopped the analysis as soon as the first solver provided a WCET estimate. Solvers finishing within the same second were considered equally fast. All programs have been analyzed sequentially, to minimize interference between the analyses and with third-party background processes. The computational effort (CPU time, peak memory usage) are derived from the Linux system call `wait3`, and thus expected to be accurate.

Bounding the Control Flow. In most cases our approach could bound the control flow automatically, and no manual input was required. However, in two benchmarks, namely *bs* and *insertsort*, CBMC could not find the bounds automatically and we were not able to accelerate or abstract either, and thus bounds had to provided. This occasional need for manual bounds is discussed in detail in Section 6.3.

In contrast, we frequently had to provide loop bounds for the ILP-based WCET analyzer. Since in an ILP approach the bounds cannot be verified, they have to be specified correctly and tightly in the first place. Towards this, whenever the ILP-based estimation required manual loop annotations, we have taken the deduced bounds from our approach and handed them to the ILP-based analyzer. Consequently, both techniques had similar preconditions for their WCET estimation.

Simulation Baseline. All benchmarks were also simulated with the cycle-accurate ISS *simulavr*, with the goal of having sanity checks for the WCET estimates, and also to get an impression on their tightness. Whenever possible, the simulation was performed with those inputs triggering the WCET. In other cases, we used random simulations in an attempt to trigger the WCET, but naturally we cannot quantify how close to the actual WCET we have come (e.g., in nsichneu). Hence, the simulation results can only serve as a lower bound for the actual WCET (i.e., no estimate must be less) and as an upper bound for the tightness (i.e., the overestimation of the actual WCET is at most the difference to the simulation value).

5.2 Results

Tables 3 and 4 summarize our experiments. We evaluated our technique of WCET estimation for each source processing stage (Section 3.1) of the selected benchmarks, that is:

1. instrumented with execution times (●),
2. sliced w.r.t. timing (◐),
3. loops accelerated (☾) and
4. abstracted (◇).

In Table 3 we compare the tightness of our WCET estimate with that of Bound-T, an ILP-based WCET analyzer. We computed the tightest possible WCET, i.e., precision $P = 1$, while allowing for a (relatively long) timeout of one hour, to show how our source transformations influence the tightness. Cases denoted with *timeout* are those where no solution was found within that time budget. Finally, the columns denoted as Δ represent the difference between the respective WCET estimates and simulation value, i.e., they give an upper bound of the tightness for both techniques.

Table 4 summarizes the computational effort of the estimation process for a practical time budget of one minute and a precision of 10,000 clock cycles. The table also quantifies the speedup we have achieved with our source transformation techniques. Again we denote *timeout* in cases where the WCET could either not be bounded within the time budget, or not up to the required precision. The column *prog.size* shows the number of program steps found by CBMC. Cases where the program size is not given (viz., in *fir* and *prime*) indicate a state-space explosion. That is, the model checker never finished constructing the state space before the timeout. The column *iter* denotes the number of iterations of the search procedure (Algorithm 1) and the column *time* denotes the total time taken by the search procedure in seconds. Cases with a valid program size *and* a timeout (e.g., *bsort100* ●) indicate that the solver backend could not verify the given properties within 10 minutes.

Table 3 Tightest WCET estimates per method and benchmark (timeout 1 hour)

	Simulation	ILP-based		Model Checking		
benchmark	observed	WCET	Δ%	stage	WCET	Δ%
adpcm-decode	48,168	71,575	+48.6	●	69,673	+44.6
				◐	69,673	+44.6
				☾	69,673	+44.6
				◇	69,673	+44.6
adpcm-encode	72,638	113,154	+55.8	●	110,901	+52.7
				◐	110,901	+52.7
				☾	110,901	+52.7
				◇	110,901	+52.7
bs	401	496	+23.7	●	410	≈0
bsort100	788,766	1,553,661	+97.0	●	timeout	–
				◇	797,598	+1.1
cnt	8,502	8,564	≈ 0	●	8,564	≈0
				◐	8,564	≈0
				◇	8,564	≈0
crc	129,470	143,137	+10.6	●	130,114	≈0
				◐	130,114	≈0
				☾	130,114	≈0
				◇	143,426	+10.8
fdct	17,500	17,504	≈ 0	●	17,504	≈0
				◐	17,504	≈0
				☾	17,504	≈0
fibcall	1,777	1,781	≈ 0	●	1,780	≈0
				◐	1,780	≈0
				☾	1,780	≈0
fir	5,204,167	5,690,524	+9.3	●	timeout	–
				◐	timeout	–
				☾	5,476,023	+5.2
insertsort	5,472	5,476	≈ 0	●	5,476	≈0
jfdctint	14,050	14,054	≈ 0	●	14,054	≈0
				◐	14,054	≈0
				☾	14,054	≈0
matmult	1,010,390	1,010,394	≈ 0	●	1,010,394	≈0
				◐	1,010,394	≈0
				☾	1,010,394	≈0
ndes	459,967	470,499	≈ 0	●	timeout	–
				◐	timeout	–
				◇	465,459	≈0
ns	56,409	56,450	≈ 0	●	56,413	≈0
				◇	56,450	≈0
nsichneu	33,199	timeout	–	●	timeout	–
				◇	75,369	+127.0
prime	27,702,943	30,343,092	+9.5	●	timeout	–
				◐	timeout	–
				◇	30,146,785	+8.8
ud	35,753	93,487	+161.5	●	38,992	+9.1
				◐	38,992	+9.1
				☾	38,992	+9.1

● instrumented, ◐ sliced, ☾ accelerated, ◇ abstracted, Δ upper bound for tightness

Finally, some benchmarks in Tables 3 and 4 are lacking a sliced, accelerated or abstracted version. This occurs when the respective source transformation technique did not result in any changes to the source code. For example, *bsort100* remained unmodified post slicing and acceleration and thus does not have a dedicated sliced or accelerated version.

5.3 WCET Path Reconstruction

We were able to reconstruct and replay the WCET path for all benchmarks. The time taken for the debugger-based replay is in the range of a few seconds in all cases. For a better

Table 4 Complexity and computational effort of WCET search for precision of 10,000 clock cycles and timeout of 1min

benchmark	stage	fastest solver(s)	iter.	time [s]	mem [MB]	prog.size
adpcm-decode	●	C	3	7.4	356	3,537
	◐	C	3	2.8	120	2,416
	☾	C	3	2.9	121	2,134
	◇	C	3	2.5	123	1,676
adpcm-encode	●	C	–	timeout	15,853	4,911
	◐	C	3	10.4	168	4,131
	☾	C	3	8.5	173	3,986
	◇	C	3	2.6	136	1,846
bs	●	A, B, C, D, E	1	0.2	117	303
bsort100	●	timeout	–	timeout	13,259	$1.57 \cdot 10^5$
	◇	A, C	5	2.3	114	5,833
cnt	●	C	3	22.1	102	3,238
	◐	C	3	23.8	100	2,212
	◇	A, B, C, D, E	3	0.3	104	373
crc	●	A	3	4.6	407	41,398
	◐	C, D	3	4.7	367	40,648
	☾	A, C, D	3	4.1	268	39,812
	◇	A, B, C, D, E	3	1.2	103	11,142
fdct	●	A, B, C, D, E	3	0.6	101	990
	◐	A, B, C, D, E	3	0.3	96	183
	☾	A, B, C, D, E	3	0.3	97	99
fibcall	●	A, B, C, D, E	1	0.1	96	592
	◐	A, B, C, D, E	1	0.1	96	496
	☾	A, B, C, D, E	1	0.1	96	79
fir	●	timeout	–	timeout	6,718	–
	◐	timeout	–	timeout	16,322	–
	☾	A	5	18.5	1,574	50,488
insertsort	●	C	1	2.3	127	1,663
jfdctint	●	A, B, C, D, E	3	0.5	101	787
	◐	A, B, C, D, E	3	0.3	96	168
	☾	A, B, C, D, E	3	0.7	98	100
matmult	●	B	5	22.7	773	70,769
	◐	A	5	6.0	661	62,367
	☾	A, B, C, D, E	5	1.6	101	9,638
ndes	●	timeout	–	timeout	40,867	75,932
	◐	timeout	–	timeout	40,877	75,797
	◇	A, C	2	0.9	167	5,727
ns	●	C	3	28.3	827	22,399
	◇	C	3	6.5	200	8,272
nsichneu	●	timeout	–	timeout	25,491	23,244
	◇	A, B, C, D, E	3	0.5	120	85
prime	●	timeout	–	timeout	22,052	–
	◐	timeout	–	timeout	23,532	–
	◇	A, B, D, E	5	0.7	99	158
ud	●	D	3	1.2	101	2,382
	◐	A, B, C, D, E	3	0.4	98	1,881
	☾	A, B, C, D, E	3	0.4	99	1,380

Step: ● instrumented, ◐ sliced, ☾ accelerated, ◇ abstracted
Solvers: A=minisat, B=mathsat, C=yices, D=z3, E=cvc4

usability, we enabled our replay engine to output the trace as a CBMC-like counterexample, but additionally augmented with all assignments to `_time`. Recall that the complete information about variable `_time` implicitly carries the control flow, because each block in the source code increments this variable. As an effect, the graphical user interface can load this augmented counterexample, and map back the counterexample to the control flow graph, as well as compute a timing profile at a granularity of block- or function-level.

Identifying a Timing Hotspot. As an example, we show the resulting annotated control flow graph for the *ud* benchmark in Figure 8. There, we have applied a heatmap over timing, where red marks the timing hotspots. At first glance, it is apparent that there exists one source block forming a timing bottleneck, which consumes about one third of the overall execution time.

The *ud* program performs a simple form of LU decomposition of a matrix A, and subsequently solves an equation system $Ax = b$. The timing bottleneck in this program occurs in the LU decomposition, but interestingly not at the innermost or most frequently executed loop, but at a location where a division occurs. With this, the reconstruction of the WCET path made it easy to spot that for the chosen target, a long division incurs a high execution cost (see also the inline annotations `TIC`), and that this is the single driver for the WCET of this program.

Discovering a Bug. The replay of the WCET path can also reveal defects in the program. Such a defect has been found in the *prime* benchmark on a (simulated) 32-bit target, where initially the WCET estimate just seemed suspiciously high. After reconstructing the WCET path, an unexpected implausibility showed up. The path was indicating that the following loop had been executed $349,865$ times:

```
for (uint i=3; i*i <= n; i+=2) {
  if (divides (i, n)) return 0;
  ...
}
```

where `n` is a formal parameter of the surrounding function that shall be checked for being a prime number. At first glance it seems like the loop can only execute $65,536$ times: the maximum 32-bit unsigned integer `n` that the user can provide is $4,294,967,295$, therefore the loop must execute at most $\lceil \sqrt{4,294,967,295} \rceil = 65,536$ times to evaluate whether that number is prime. Thus, we replayed the WCET path in our debugger, setting a watchpoint on `i`. It turned out that the expression `i*i` will overflow for some specific values of `n` $> 4,294,836,225$, since this would require at least `i`$\geq 65,536$, which, when squared, lies beyond the maximum representable unsigned integer and thus overflows. Specifically, this happens for only those numbers which are also not divisible by anything less than $65,536$, hard to replicate if only the path and no values would be available. As an effect, the loop keeps running until finally the 32-bit modulo of `i*i` is larger than `n` before the loop terminates (which luckily is always the case). Clearly, this is a defect in the context of this program, resulting in a WCET which was orders of magnitude higher than it should have been[3]. Thanks to the ability of interactively replaying the WCET path and inspecting variable values, such defects can be easily spotted.

Top Ten Longest Paths. In principle our approach could also provide the "top 10" longest paths in the program. This could be done by repeating the overall WCET estimation, while excluding already known paths from the solution, and decreasing the WCET proposal to the Model Checker if no coun-

[3] Note that this code works correctly for 16-bit targets, as $\forall n \in$ `uint16`, $\exists i \leq 255$, s.t. `divides(i,n)`.

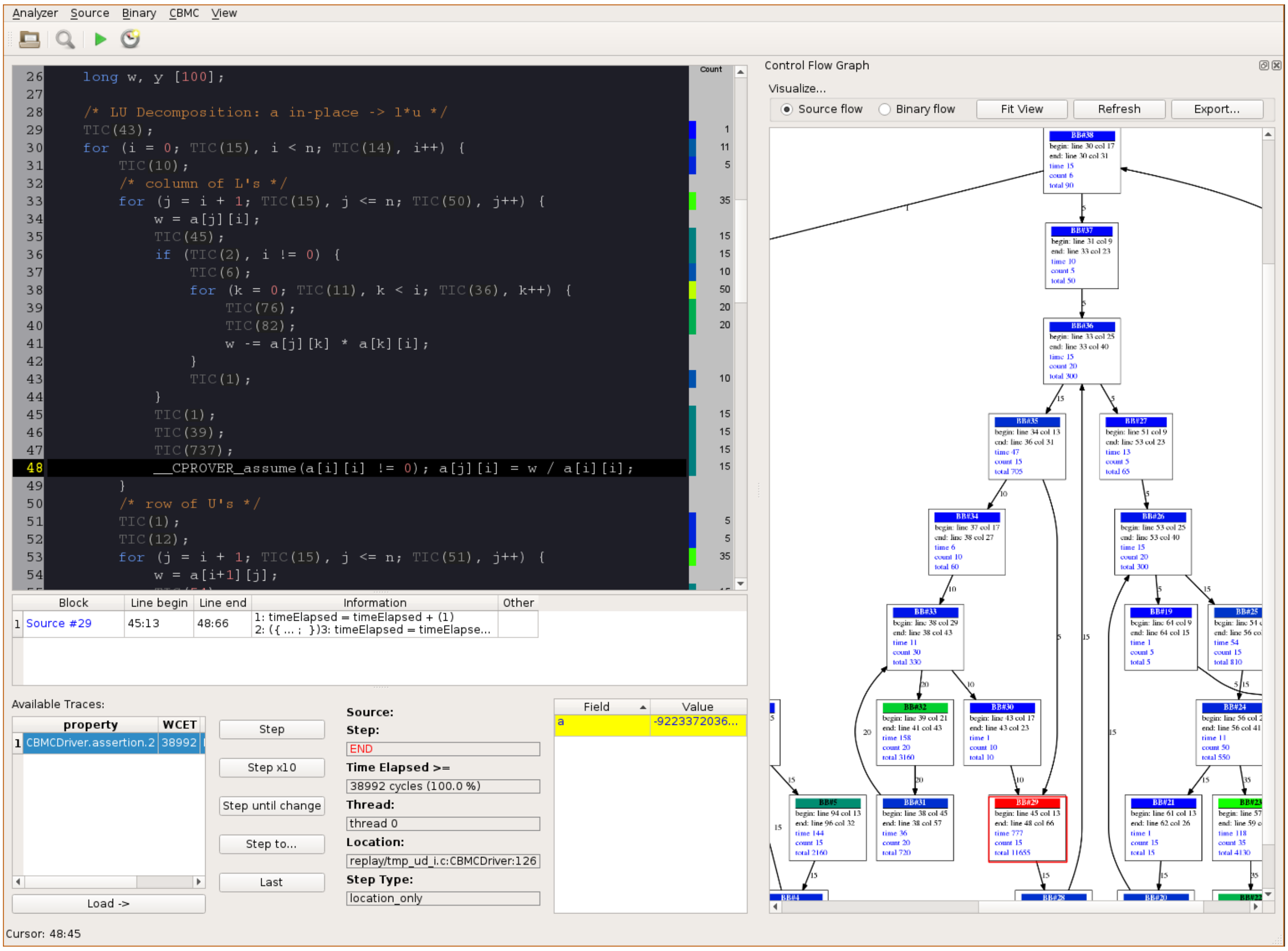


Fig. 8 WCET path reconstruction for benchmark *ud* with identified timing hotspot

terexample can be found. However, then this would become an explicit enumeration of paths, typically exponentially in the size of the program [38], and would still leave us with the question of how the program's overall probability distribution looks like (unless we repeat our estimation until every possible path length has been found). We did not investigate this further, as this clearly would not scale well with program size.

6 Discussion

Using our approach, the WCET could be estimated for all benchmarks, within a few seconds, including a timing profile and path replay of the WCET case. Through that, not only did we provide an estimate without minimal user inputs, but also we generate useful insights into the WCET path, enabling a true "time debugging". As we elaborate on the following pages, source transformation techniques had a noticeable impact on scalability of Model Checking, and thus paved the way to use Model Checking for WCET analysis. As a result, the estimates are comparable to those of an ILP-based technique, but with less effort and more feedback for the user.

6.1 Tightness

The tightness of TIC's WCET estimate can be expected to be almost exact when Model Checking is allowed to explore all paths (i.e., no abstraction applied), and becoming less tight when abstractions are applied, comparable to an ILP-based solver where only loops are bounded with no further flow analysis. In fact, Table 3 shows that our WCET estimates often are even tighter than an ILP-based approach.

For the *adpcm* benchmarks, there is still a lot of room for improvement. The computed WCET, even on the original version, is likely a large overestimation, as suggested by the simulation. The reasons for this are discussed in the following.

6.1.1 Intrinsic Sources of Overapproximation

During the back translation of timing information from machine instructions to source code, overapproximations have been used as described earlier. We argue that these over-approximations are common even when using the existing ILP-based techniques. During back translation, wherever there is a difference between the source blocks and basic blocks in the machine instructions, we over-approximate that part machine instructions into one block. However, these over-approximations are usually small, since these differences are often formed by few and small basic blocks, amounting to only a few clock cycles per iteration.

Without any abstractions, TIC will exhaustively explore all feasible paths in the (instrumented) source code, and therefore not make additional overapproximations. Thus, when identifying the longest path, TIC will never be worse than an ILP-based path search. In fact, the ILP solver could only perform better, if the control flow could be bounded *exactly* and then encoded in the ILP formulation.

A typical case is that of *crc*. When no abstractions were applied, our estimate is around 10% tighter than that of Bound-T (130,114 vs. 143,137), which we tracked down to infeasible paths being considered by Bound-T: In *crc*, there exists a loop containing an if-else statement, whereas the if-part takes more time than the else-part. Bound-T, not analyzing the data flow and dependency between these two parts, always assumes the longer if-branch is taken. However, this is only feasible 70% of the time, whereas the other 30% the shorter else-branch takes place. Without additional user annotations or a flow analysis, ILP cannot discover this dependency and thus must overestimate the WCET.

Consequently, TIC performs better than an ILP-based approach when abstractions are left aside, if the program complexity had been reduced enough by slicing and acceleration. For the remaining cases, where abstraction was necessary to make Model Checking scale, we now must discuss the overestimation caused thereby.

6.1.2 Overestimation due to Abstraction

Abstraction overapproximates the control and data flows, trading analysis time for tightness. When applying loop abstraction, the abstracted code forces the model checker to pick times along the branch with the longest times, cutting out shorter branches. Thus, we compute the WCET assuming that every branch inside the loop will always take the worst local choice, which may be infeasible. Naturally, this leads to an overestimation of WCET. However, ILP-based analyzers usually over-approximate with similar strategies when bounding the control flow. The result is a WCET estimate comparable to that of an ILP-based analyzer.

Again, consider *crc* as a typical case. When no abstractions were applied, our estimate was around 10% tighter than that of Bound-T. When applying abstractions, the estimate became very close to the estimate of Bound-T, whereas the complexity was cut down to approximately 25%.

Surprisingly, in some benchmarks, namely *adpcm-decode* and *encode*, *cnt*, *fdct*, *jfdctint* and *ud*, the loop abstraction did not lead to a higher WCET estimate. This is because in all the loops with branches in these programs (a) either there is an input to the program that forces the branch with the highest time to be taken in all the iterations of the loop, or (b) there is a break or return statement that gets taken in the last iteration of the loop. These cases match the exact pessimistic loop abstraction. In short, these loops do exhibit the pessimistic worst case timing behavior.

An extreme overestimation due to abstraction seems likely (reminder: the simulation is not guaranteed to contain the worst-case) for *nsichneu*, where the estimate is 127% higher than the observed WCET. This benchmark has only one loop with two iterations, but its simulated WCET is far away from the observed WCET. Upon inspection, we found that this loop has 256 if-statements in a single sequence, many of which do not get executed in every iteration. However, our loop abstraction pessimistically assumes that all the 256 if-statements do get executed in each iteration, which explains the overestimation in this case. Note that this benchmark could not be solved with Bound-T, at all.

Consequently, TIC performs close to and often slightly better than an ILP-based approach when abstractions are used. However, this result depends on the way control flows are bounded before the ILP solver is called, and thus it might not hold true when AI and ILP are combined.

6.2 Reduction of Computational Effort

Our claim was, that the scalability issues of Model Checking could be mitigated with appropriate source preprocessing techniques, which we expected to be particularly effective at source code level. The experimental data clearly confirms that claim. In all cases, the analysis time – usually in the range of minutes up to unsolvable size for the original version – could be reduced to an acceptable time of at most a few seconds, which makes TIC an interesting alternative to existing WCET estimation techniques.

However, taking the analysis time (computational effort) as measure of the computational complexity can be misleading. The time to solve the WCET problem in our approach consists of two fundamental parts: 1. building the SAT/SMT model and 2. solving the model. Whereas the time to build the model is often proportional to the size of the program (number of steps as found by CBMC), the time for solving the model cannot be predicted trivially by looking at program metrics. In particular, we have found no correlation between any of program size, cyclomatic complexity, number of statements, number of loops and the analysis time.

The reason for this is that the analysis time of a SAT or SMT problem can vary significantly between two problems of the same size or between two solvers, due to the nature of modern solver algorithms [15]. For instance, compare *adpcm-encode*● (program size 4,911 steps, timeout after 1 minute) with *ndes*◇ (program size 5,727 steps, solved in less than one second).

Therefore, we used a portfolio of solvers to reduce the effect of solver-specific strengths and weaknesses, and we consider the program size (last column in Table 4) as a prime indicator for the complexity of the program. With this, we show in the following that the complexity of a program can be significantly reduced with our techniques. Nevertheless, note that the solving time also points towards our interpretation. In all benchmarks we were able to greatly reduce the computational effort with each source processing stage. In particular, several benchmarks could not be solved in a reasonable amount of time without our proposed processing, viz., *adpcm-encode*, *bsort100*, *fir*, *ndes*, *nsichneu* and *prime*. The memory usage suggests, that a state-space explosion prevents building the model. Their program size could be reduced to a tractable size with acceleration and abstraction. Moreover, the benchmark *nsichneu* could not even be processed with Bound-T, since bounding of the control flow had failed because of too many variables and flows (out of memory after running for several hours). After applying our abstraction, the WCET of this benchmark could be successfully estimated with Model Checking, within one second of analysis time.

The overall impact of source transformations is quantified in Figure 9, where the program size after the respective processing stage is compared to the instrumented program and over all benchmarks. It can be seen, that on average each additional processing stage reduces the program complexity; in average we reach 78% of the original complexity after slicing, 63% after acceleration, and 22% after abstraction. Furthermore, it can be seen that in the worst case slicing and acceleration have no effect, whereas abstraction has a much more consistent impact on the complexity.

Note that the numbers in Figure 9 exclude those benchmarks where a timeout had occurred for the instrumented version, i.e., the original program without our source processing. Here, the program size could not be determined. For each of those benchmarks, we have additionally spent 24 hours of processing to determine the program size, but without success.

As explained in Section 3.6, further reduction of the analysis time can be reached if a lower bound for the WCET is already known. This is often the case for safety-critical systems, where run-time measurements (like high watermarks) are common practice. The user would initialize the search algorithm with an observed WCET, thereby reducing the analysis time. However, this does not change the structure of the model under analysis, and therefore is not considered a complexity reduction.

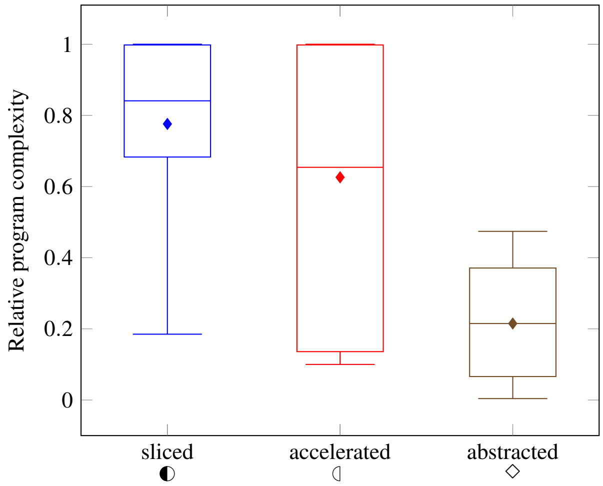


Fig. 9 Box plot showing reduction of program complexity of after different source transformation stages, relative to original program. Whiskers are denoting the mix/max values, diamonds the average value

6.3 Safety and Usability

Most programs can be analyzed automatically, without any user input. However, two programs, namely *bs* and *insertsort*, could not be bounded automatically. The loop bounds could not be deduced by CBMC, which shows up as an infinite loop unwinding phase. In such cases, the user has to specify an *unwinding depth*, at which the loop unwinding stops. CBMC then places an assertion at this point, which proves that the given depth is sufficient. In case of an insufficient bound, the model checker identifies this as a failed property. In case of a too generous bound, the unwinding may take longer, but WCET is not overestimated. Therefore, unsound or too imprecise flow constraints do not refute (or even worsen) the WCET estimate, which makes our approach safer than others.

Another contribution to the safety of our toolchain are the run-time checks introduced during WCET replay. As discussed earlier, a wide range of problems can be discovered by this, including confounding of input data and failure to specify the details of the target architecture.

Regarding the usability of our toolchain, we argue that the results of a source-level WCET analysis, especially the possibility to replay the worst-case path, should increase the level of acceptance for WCET analysis in a practical setting. As opposed to machine code or assembly, developers can be expected to be proficient in understanding a source code, for its higher level of abstraction and higher readability. By offering the possibility to replay the WCET path through an ordinarily looking debugging session – which can be controlled in the usual ways of stepping, inspecting and resuming – it becomes intuitive to walk through the path an identify the drivers of WCET. This is also supported by generating a profiler-like timing profile of the worst-case path, which is another, well-known view at a program, and a quick entry point for WCET inspection. A developer can identify those parts of the WCET path that need an in-depth inspection, and subsequently dive into debugging.

In summary, our approach is inherently safer than existing approaches for its protection against unsound user input, and it ensures that developers need no additional tools or training to analyze and understand the WCET path of a program.

6.4 Correctness

The WCET estimate computed by TIC will always be an upper bound of the actual WCET. This is easy to see as in each step in TIC, we either over-approximate or preserve the timing information of machine instructions. In step 1, while the execution time computed for each basic block in TIC (Section 3.1) is precise, the back-annotation done in Section 3.2 over-approximates the time in the case where multiple basic blocks map to one source line. Therefore, at the end of this step, the WCET of the machine instructions is either preserved or over-approximated in the source.

The slicing removes only those statements that do not impact the values of the counter variable, thus preserving the WCET as per the instrumented program.

In the next stage, acceleration preserves the computation of the counter variable and abstraction potentially over-approximates it. So, at the end of this step, too, WCET is either preserved or over-approximated. Finally, the iterative search procedure of Algorithm 1 terminates only upon finding an upper bound of the WCET as per the accelerated or abstracted program. Thus, if TIC scales for the input program, it will provide a safe upper bound for the WCET.

6.5 Threats to Validity

Microarchitectural Features. We presented our results for a simple microcontroller. Modern processors, such as ARM SoCs, have architectural features like caches and pipelines, and memory-mapped I/O. As shown in [11], some of these features can be modeled in C in the form of appropriate constraints. However, adding such constraints would increase the complexity of the C code. Thus, while TIC can be extended to handle for these features, the additional constraints may hinder its scalability. In such a situation, we can eliminate infeasible paths using Model Checking (as in [11]) and reduce the search space of the model checker (as in [29]) while applying our technique. Furthermore, when the instruction timing depends on the value of operands, a register value analysis becomes necessary. For example, to determine the timing of a load instruction, the specific address decides whether this is cached, or it becomes a slow bus access, possibly with waiting states, or a fast access to a core-coupled memory. At the moment, we have not addressed how to carry over such a value analysis to the source code level.

Back-Annotation of Timing. The mapping of temporal information from machine instructions to source-level is a rather technical problem. A compiler could be build which enables complete traceability in at least one direction. Plans for such a compiler have been made in [51], unfortunately, we are not aware of any implementation, which leaves us with the task of matching instructions to source code. Without any mapping information, this poses an optimal inexact graph matching problem. Specifically, when compiler optimization is off, then we expect this to boil down to a graph minor test (where the edit sequence represents our wanted mapping and the source graph is fixed), known to have a polynomial complexity in the number of nodes [53]. With optimization on, however, it is unclear how the matching could be established in general. We therefore have to rely on some mapping information from the compilation process (as given by *addr2line* in our toolchain), and apply techniques to complete the mapping.

Our mapping strategy assumes that the program is compiled using gcc 4.8.1 for the 8bit AVR microprocessor family [36], with standard compilation options. If we change the target-compiler pair or the compilation options, then our backmapping strategy may not work. Since it was established on a case-by-case basis, our strategy might not be complete, but it was extensive enough to cover all cases in the Mälardalen benchmarks. In general, compiler transformations are a common problem for all WCET techniques. And, to the best of our knowledge, there has been no generic solution to this problem, except to provide support for each architecture, compiler and transformation individually, often in the form of pattern matching [59].

Compiler-Inserted Code. The compiler may insert low-level procedures, such as loops for arithmetic shifts or code for soft-floating point operations. These are not covered in our current tool. We believe that this is only a matter of tooling. If the source code for such computations is not available, then TIC requires a pre-computed library of the WCETs of low-level functions, to facilitate the backmapping for the low-level loops.

7 Related Work

WCET Analysis. An excellent survey about techniques and tools for WCET analysis was published by Wilhelm et al. [59]. We refer the reader to this article for a profound overview on the topic. A commonly used set of benchmark programs for WCET analysis are the Mälardalen WCET benchmarks [24].

Model Checking for WCET Analysis. There have been studies highlighting the inadequacy [58,40] of Model Checking for the task of WCET computation, concluding that is does not scale for WCET estimation. Experiments in [40] confirmed that model checkers often face run-time- or memory-infeasibility for complex programs, whereas the ILP technique can compute the WCET almost instantly. However, because they used the same benchmarks that we are using here (on a very similar processor) *and* we come to the opposite conclusion, this confirms that our shift of the analysis to the source code indeed mitigates the scalability issue.

Further, there have been instances where a model checker was used as a subanalysis or optimization step in WCET estimation. Chattopadhyay et al. [11] propose the use of AI for cache analysis and Model Checking for pruning infeasible paths considered by AI. Marref et al. [41] show automatic derivation of exact program flow constraints for WCET computation using Model Checking for hypothesis validation. Another proponent for Model Checking in WCET analysis is found in Metzner [43], who has shown that it can improve cache analysis, while not suffering from numerical instabilities, unlike ILP. However, none of these approaches addressed the scalability issue of Model Checking.

WCET Analysis at Higher Levels of Abstraction. One of the first works proposing WCET analysis at source level was published by Puschner [50]. He introduced the notion of *timing schemata*, where each source-level construct is assigned a constant execution time. Many constraints have been imposed on the programming language, as well as annotation constructs to aid analysis. A similar approach was described shortly after that in [46]. In fact, the processors they used were very comparable to the one used in our experiments. However, in their case, all loops must be statically bounded, and overestimation is a direct result of assigning a constant execution time to source level constructs (since the compiler may translate the same construct very differently, depending on the specific variable types and values).

Holsti [30] proposed modeling execution time as a global variable, and to use a dependency and value analyses (Presburger Arithmetic) to determine the value of the variable at the end of a function and meanwhile exclude infeasible paths. Similar to our approach, slicing and loop acceleration were suggested. However, he showed by example that only some infeasible paths can be excluded by his method, whereas we can precisely detect all infeasible paths. Furthermore, his approach seems to have scalability issues, which unfortunately are not detailed further due to a lacking set of experiments. Kim et al. [32] experimented with computing WCET using Model Checking at source level on small and simple programs, but without addressing the scalability issues, nor providing experimental data.

Puschner [49] later computed WCET on an AST-like representation of a program, where flow constraints are expected to be given by the user, and assuming that the compiler performs the back-mapping of actual execution times to the source code. He used ILP to compute the longest path. It should be noted that any ILP-based approach cannot handle variable execution times of basic blocks without large overapproximation, whereas Model Checking can encode such properties with non-determinism and range constraints. Furthermore, complete path reconstruction is not possible either with that approach.

WCET analysis has also been proposed at an intermediate level in between source code and machine code, similarly because of easier analysis of data and control flow. Altenbernd [2] et al. developed an approximation of WCET which works on an ALF representation of the program, without analyzing the executable. They automatically identified a timing model of the intermediate instruction set through execution and measurement of training programs. As an effect, the analysis is very efficient, but the result is a possibly unsafe WCET estimate.

Program Slicing, Acceleration and Abstraction. Hatcliff [27] was the first to suggest the use of program slicing to help scale up Model Checking. In this work, we have build on the acceleration and abstraction capabilities of LABMC [14]. Different abstractions for improving the precision of WCET computation or determining loop bounds have been explored by other researchers. Ermedahl et al. [19] show precision improvement in WCET computations by clustering basic blocks in a program. Knoop et al. [34] use recurrence relations to determine loop bounds in programs. Blazy et al. [7] use program slicing and loop bound calculation techniques to formally verify computed loop bounds. Černý et al. [10] apply a segment abstraction technique to improve the precision of WCET computations. While in these abstractions could be used in some situations, in general they are either too restrictive because they do not work for a large class of programs, or they fail to address the scalability issue arising in the context of using a model checker to compute the WCET. Al-Bataineh et al. [1] use a precise acceleration of timed-automata models (with cyclic behavior) for WCET computation, to scale up their IPET technique. However, these ideas are not readily applicable to loop acceleration in C programs in the absence of suitable abstractions.

Timing Debugging. Understanding how the worst-case timing is produced is very important for practitioners, but there is only a small body of work on this topic. Reconstructing the inputs leading to WCET has been done before by Ermedahl [18], and is perhaps the closest work in respect to the degree of detail that is provided on the WCET path. Our approach, however, uses entirely different methods. While Ermedahl applies a mixture of static analysis and measurements to perform a systematic search over the value space with WCET estimation, and only provides the inputs to the WCET path, our approach is leveraging the output of the model checker that witnesses the WCET estimate, performs only a single run of the application, and reconstructs the WCET inputs, as well as the precise path being taken together with a timing profile. It is thus less expensive and better integrated with the actual WCET estimation.

Making the worst-case (and best-case) timing visible in the source code is a rather old idea, first appeared around 1995 [35]. The authors highlighted WCET/BCET paths in the source code, allowing the user to visually see the WCET path and how time passes on that path. This work was later extended in [60] to apply genetic algorithms for minimizing WCET, and still later to introduce compiler transformations to reduce the WCET (this is, however, beyond the scope of this article). A similar, but interactive tool was presented in [26]. This was also a tree-based approach for it was focusing on analysis speed instead of precision. All these approaches did not provide tight results, as they were based on timing trees which did not consider data dependencies; they can only reconstruct the WCET path w.r.t. locations and time spent there, but lack a specific trace including variable values. Furthermore, trivial loop bounds had to be given manually by the user.

The commercial WCET tool RapiTime [5] estimates the timing of software, and enables the user to identify timing bottlenecks at source-code level. For that, the tool provides a simple path highlighting, but also allows predicting what would happen if a specific function would be optimized. However, the tool is measurement-based and thus cannot give any guarantees. The commercial tool AbsInt [20] provides time debugging capabilities at the assembly level, e.g., a call graph and a timing profile. It is also capable of mapping back this timing information to a high-level model, where the building blocks of the model are annotated with their contribution to the WCET. Further, the tool allows to make changes to the model, and compare the results with the previous ones, essentially enabling a trial-and-error approach to improve the WCET. Finally, a similar toolset that realizes a generic formal interface between a modeling tool and a timing analysis tool has been presented in [21], but relies on external tools for the actual analysis. All of these tools enable timing debugging to some extent, but they cannot provide a specific trace with values, or even allow the user to interactively step through the same. In contrast, our WCET path has the maximum level of detail, and can be inspected in a well-known debugger environment, thus offers deeper and more intuitive explanation of how the WCET path came to be.

Possible Enhancements. One approach that could be combined with ours to further speed up Model Checking, is that of Henry et al. [29]. They also employ an SMT solver to compute the WCET (just like our back-end), but they propose additional constraints to have the solver ignore infeasible paths. This helps to further increase the scalability, but under the assumption that the programs are loop-free, or that loops have been unrolled. This is therefore an enhancement that fits well our approach. Brandner [8] computed time criticality information on all parts of a program, to help focus on optimizing paths that are close to WCET, and not only those on it. However, this work only provides a relative ranking of all code blocks (not absolute numbers on time consumption), and requires an external WCET analyzer that annotates code blocks with individual WCETs. It can therefore be viewed as another possible extension for our work.

8 Concluding Remarks

We have shown that Model Checking can be a competitive approach to Worst-Case Execution Time Analysis, in particular when an analysis-friendly processor is used. The estimates are comparable to and sometimes even more precise than estimates of the widely-used ILP technique, but with several practical advantages. Additional to a precise estimate, we also reconstruct and replay the WCET path, while providing profiling data and a well-known debugger interface, allowing the user to inspect arbitrary details on the WCET path at both source code and machine code level. Although our approach does not entirely remove the need for manual inputs from the user, WCET analysis is no longer prone to human error coming from there, because the model checker also verifies whether such inputs are sound. If too small bounds are given, an error is flagged. Too large bounds, on the other hand, only influence the analysis time, but not the outcome. In summary, we therefore arrive at a safer WCET analysis and a more intuitive understanding of the outcome.

An essential part of our approach is the shift of the analysis from machine instructions to the source code. Through this, data and control flows can be tracked more precisely, and source code transformation techniques can be applied to summarize loops, to remove statements not related to timing, and to over-approximate larger programs. As a result, the analysis time can be reduced significantly, making Model Checking a viable approach to the Worst-Case Execution Time problem.

What we have shown in this article is merely the first step of reintroducing Model Checking to Worst-Case Execution Time Analysis. However, there is substantially more work to be done to catch up with the proven approaches of combining AI and ILP: Here, we assumed processors with almost constant instruction timing. While certainly processors for real-time applications should be simplified to address the self-made and acknowledged problem of processors becoming more and more unpredictable, the approach presented here must be extended to meet current processors half way. Specifically, we plan to include a register-level value analysis to be able to handle variable timing due to memory-mapped I/O, and to propose a specific source-level model for scratchpad memories. This will result in support for a much wider range of processors. The challenge in these extensions will mainly be how to lift models for these architectural behaviors to the source code, without impairing the scalability of Model Checking too much. Nevertheless, pursuing this route should be worth the efforts that are on the way, by reason of the practical benefits over existing approaches, such as higher automation and usability.

–

References

1. Al-Bataineh, O., Reynolds, M., French, T.: Accelerating worst case execution time analysis of timed automata models with cyclic behaviour. Formal Aspects of Computing **27**(5), 917–949 (2015)
2. Altenbernd, P., Gustafsson, J., Lisper, B., Stappert, F.: Early execution time-estimation through automatically generated timing models. Real-Time Systems **52**(6), 731–760 (2016)
3. Axer, P., Ernst, R., Falk, H., Girault, A., Grund, D., Guan, N., Jonsson, B., Marwedel, P., Reineke, J., Rochange, C., Sebastian, M., von Hanxleden, R., Wilhelm, R., Yi, W.: Building timing predictable embedded systems. ACM Trans. Embedded Comput. Syst. **13**(4), 82:1–82:37 (2014)
4. Becker, M., Neumair, M., Söhn, A., Chakraborty, S.: Approaches for Software Verification of an Emergency Recovery System for Micro Air Vehicles. In: F. Koornneef, C. van Gulijk (eds.) Proc. Computer Safety, Reliability, and Security - 34th International Conference (SAFECOMP), *Lecture Notes in Computer Science*, vol. 9337. Springer (2015)
5. Bernat, G., Davis, R., Merriam, N., Tuffen, J., Gardner, A., Bennett, M., Armstrong, D.: Identifying opportunities for worst-case execution time reduction in an avionics system. Ada User Journal **28**(3), 189–195 (2007)
6. Beyer, D.: Status report on software verification - (competition summary SV-COMP 2014). In: E. Ábrahám, K. Havelund (eds.) Proc. 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), *Lecture Notes in Computer Science*, vol. 8413, pp. 373–388. Springer (2014)
7. Blazy, S., Maroneze, A.O., Pichardie, D.: Formal verification of loop bound estimation for WCET analysis. In: E. Cohen, A. Rybalchenko (eds.) Proc. 5th International Conference on Verified Software: Theories, Tools, Experiments (VSTTE), *Lecture Notes in Computer Science*, vol. 8164, pp. 281–303. Springer (2014)
8. Brandner, F., Hepp, S., Jordan, A.: Static profiling of the worst-case in real-time programs. In: L. Cucu-Grosjean, N. Navet, C. Rochange, J.H. Anderson (eds.) Proc. 20th International Conference on Real-Time and Network Systems (RTNS), pp. 101–110. ACM (2012)
9. Brumley, D., Jager, I., Avgerinos, T., Schwartz, E.J.: BAP: A binary analysis platform. In: G. Gopalakrishnan, S. Qadeer (eds.) Proc. 23rd International Conference on Computer Aided Verification (CAV), *Lecture Notes in Computer Science*, vol. 6806, pp. 463–469. Springer (2011)
10. Cerný, P., Henzinger, T.A., Kovács, L., Radhakrishna, A., Zwirchmayr, J.: Segment abstraction for worst-case execution time analysis. In: J. Vitek (ed.) Proc. 24th European Symposium on Programming Languages and Systems (ESOP), *Lecture Notes in Computer Science*, vol. 9032, pp. 105–131. Springer (2015)
11. Chattopadhyay, S., Roychoudhury, A.: Scalable and precise refinement of cache timing analysis via path-sensitive verification. Real-Time Systems **49**(4), 517–562 (2013)
12. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: K. Jensen, A. Podelski (eds.) Proc. 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), *Lecture Notes in Computer Science*, vol. 2988, pp. 168–176. Springer (2004)
13. Clarke Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge, MA, USA (1999)
14. Darke, P., Chimdyalwar, B., Venkatesh, R., Shrotri, U., Metta, R.: Over-approximating loops to prove properties using bounded model checking. In: W. Nebel, D. Atienza (eds.) Proc. Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1407–1412. ACM (2015)
15. Demyanova, Y., Pani, T., Veith, H., Zuleger, F.: Empirical software metrics for benchmarking of verification tools. In: D. Kroening, C.S. Pasareanu (eds.) Proc. 27th International Conference on Computer Aided Verification (CAV), *Lecture Notes in Computer Science*, vol. 9206, pp. 561–579. Springer (2015)
16. Ding, H., Liang, Y., Mitra, T.: Wcet-centric partial instruction cache locking. In: P. Groeneveld, D. Sciuto, S. Hassoun (eds.) Proc. 49th Annual Design Automation Conference (DAC), pp. 412–420. ACM (2012)
17. Edwards, S.A., Kim, S., Lee, E.A., Liu, I., Patel, H.D., Schoeberl, M.: A disruptive computer design idea: Architectures with repeatable timing. In: Proc. 27th International Conference on Computer Design (ICCD), pp. 54–59. IEEE Computer Society (2009)
18. Ermedahl, A., Fredriksson, J., Gustafsson, J., Altenbernd, P.: Deriving the worst-case execution time input values. In: Proc. 21st Euromicro Conference on Real-Time Systems (ECRTS), pp. 45–54. IEEE Computer Society (2009)
19. Ermedahl, A., Stappert, F., Engblom, J.: Clustered worst-case execution-time calculation. IEEE Trans. Computers **54**(9), 1104–1122 (2005)
20. Ferdinand, C., Heckmann, R., Le Sergent, T., Lopes, D., Martin, B., Fornari, X., Martin, F.: Combining a high-level design tool for safety-critical systems with a tool for wcet analysis of executables. In: Proc. 4th European Congress on Embedded Real Time Software (ERTS). SIA/AAAF/SEE (2008)
21. Fuhrmann, I., Broman, D., von Hanxleden, R., Schulz-Rosengarten, A.: Time for reactive system modeling: Interactive timing analysis with hotspot highlighting. In: A. Plantec, F. Singhoff, S. Faucou, L.M. Pinho (eds.) Proc. 24th International Conference on Real-Time Networks and Systems (RTNS), pp. 289–298. ACM (2016)
22. Furber, S.: ARM System-on-Chip Architecture. Addison Wesley (2000)
23. Gulwani, S., Jain, S., Koskinen, E.: Control-flow refinement and progress invariants for bound analysis. In: M. Hind, A. Diwan (eds.) Proc. Conference on Programming Language Design and Implementation (PLDI), pp. 375–385. ACM (2009)
24. Gustafsson, J., Betts, A., Ermedahl, A., Lisper, B.: The mälardalen WCET benchmarks: Past, present and future. In: B. Lisper (ed.) Proc. 10th International Workshop on Worst-Case Execution Time Analysis (WCET), *OASICS*, vol. 15, pp. 136–146. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany (2010)
25. Gustafsson, J., Ermedahl, A., Sandberg, C., Lisper, B.: Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution. In: Proc. 27th International Real-Time Systems Symposium (RTSS), pp. 57–66 (2006)
26. Harmon, T., Klefstad, R.: Interactive back-annotation of worst-case execution time analysis for java microprocessors. In: Proc. 13th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pp. 209–216. IEEE Computer Society (2007)
27. Hatcliff, J., Dwyer, M.B., Zheng, H.: Slicing software for model construction. Higher-Order and Symbolic Computation **13**(4), 315–353 (2000)

28. Healy, C.A., Sjödin, M., Rustagi, V., Whalley, D.B., van Engelen, R.: Supporting timing analysis by automatic bounding of loop iterations. Real-Time Systems **18**(2/3), 129–156 (2000)
29. Henry, J., Asavoae, M., Monniaux, D., Maiza, C.: How to compute worst-case execution time by optimization modulo theory and a clever encoding of program semantics. In: Y. Zhang, P. Kulkarni (eds.) Proc. 15th Conference on Languages, Compilers and Tools for Embedded Systems (LCTES), pp. 43–52. ACM (2014)
30. Holsti, N.: Computing time as a program variable: a way around infeasible paths. In: R. Kirner (ed.) Proc. 8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, *OASICS*, vol. 8. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany (2008)
31. Holsti, N., Saarinen, S.: Status of the Bound-T WCET tool. Space Systems Finland Ltd (2002)
32. Kim, S., Patel, H.D., Edwards, S.A.: Using a Model Checker to Determine Worst-Case Execution Time. Tech. rep., Columbia University (2009). CUCS-038-09
33. Kirner, R., Puschner, P.P.: Obstacles in worst-case execution time analysis. In: Proc. 11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC), pp. 333–339. IEEE Computer Society (2008)
34. Knoop, J., Kovács, L., Zwirchmayr, J.: Symbolic loop bound computation for WCET analysis. In: E.M. Clarke, I. Virbitskaite, A. Voronkov (eds.) Proc. 8th International Conference Perspectives of Systems Informatics (PSI), Revised Selected Papers, *Lecture Notes in Computer Science*, vol. 7162, pp. 227–242. Springer (2012)
35. Ko, L., Healy, C.A., Ratliff, E., Arnold, R.D., Whalley, D.B., Harmon, M.G.: Supporting the specification and analysis of timing constraints. In: Proc. 2nd Real-Time Technology and Applications Symposium (RTAS), pp. 170–178. IEEE Computer Society (1996)
36. Kuhnel, C.: AVR RISC Microcontroller Handbook, first edn. Newnes (1998)
37. Kuo, M.M.Y., Yoong, L.H., Andalam, S., Roop, P.S.: Determining the worst-case reaction time of IEC 61499 function blocks. In: Proc. 8th IEEE International Conference on Industrial Informatics, pp. 1104–1109. IEEE (2010)
38. Li, Y.T., Malik, S.: Performance analysis of embedded software using implicit path enumeration. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **16**(12) (1997)
39. Lickly, B., Liu, I., Kim, S., Patel, H.D., Edwards, S.A., Lee, E.A.: Predictable programming on a precision timed architecture. In: E.R. Altman (ed.) Proc. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, (CASES), pp. 137–146. ACM (2008)
40. Lv, M., Gu, Z., Guan, N., Deng, Q., Yu, G.: Performance comparison of techniques on static path analysis of WCET. In: C. Xu, M. Guo (eds.) Proc. International Conference on Embedded and Ubiquitous Computing (EUC), pp. 104–111. IEEE Computer Society (2008)
41. Marref, A.: Fully-automatic derivation of exact program-flow constraints for a tighter worst-case execution-time analysis. In: L. Carro, A.D. Pimentel (eds.) Proc. International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), pp. 200–208. IEEE (2011)
42. Metta, R., Becker, M., Bokil, P., Chakraborty, S., Venkatesh, R.: TIC: a scalable model checking based approach to WCET estimation. In: T. Kuo, D.B. Whalley (eds.) Proc. 17th Conference on Languages, Compilers, Tools, and Theory for Embedded Systems (LCTES), pp. 72–81. ACM (2016)
43. Metzner, A.: Why model checking can improve WCET analysis. In: R. Alur, D.A. Peled (eds.) Proc. 16th International Conference on Computer Aided Verification (CAV), *Lecture Notes in Computer Science*, vol. 3114, pp. 334–347. Springer (2004)
44. Mitra, T., Teich, J., Thiele, L.: Adaptive Isolation for Predictability and Security (Dagstuhl Seminar 16441). Dagstuhl Reports **6**(10), 120–153 (2017)
45. Mittal, S.: A survey of techniques for cache locking. ACM Trans. Design Autom. Electr. Syst. **21**(3), 49:1–49:24 (2016)
46. Park, C.Y., Shaw, A.C.: Experiments with a program timing tool based on source-level timing schema. IEEE Computer **24**(5), 48–57 (1991)
47. Pingali, K., Bilardi, G.: APT: A data structure for optimal control dependence computation. In: D.W. Wall (ed.) Proc. Conference on Programming Language Design and Implementation (PLDI), pp. 32–46. ACM (1995)
48. Puschner, P.: Is WCET Analysis a Non-Problem? – Towards New Software and Hardware Architectures. In: G. Bernat (ed.) Proc. 2nd International Workshop on Worst-Case Execution Time Analysis (WCET), pp. 89–92. Technical University of Vienna, Austria (2002)
49. Puschner, P.P.: A tool for high-level language analysis of worst-case execution times. In: Proc. 10th Euromicro Conference on Real-Time Systems (ECRTS), pp. 130–137. IEEE Computer Society (1998)
50. Puschner, P.P., Koza, C.: Calculating the maximum execution time of real-time programs. Real-Time Systems **1**(2), 159–176 (1989)
51. Puschner, P.P., Prokesch, D., Huber, B., Knoop, J., Hepp, S., Gebhard, G.: The T-CREST approach of compiler and wcet-analysis integration. In: Proc. 16th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, (ISORC), pp. 1–8. IEEE Computer Society (2013)
52. Raymond, P., Maiza, C., Parent-Vigouroux, C., Carrier, F.: Timing analysis enhancement for synchronous program. In: M. Auguin, R. de Simone, R.I. Davis, E. Grolleau (eds.) Proc. 21st International Conference on Real-Time Networks and Systems (RTNS), pp. 141–150. ACM (2013)
53. Robertson, N., Seymour, P.: Graph minors .xiii. the disjoint paths problem. Journal of Combinatorial Theory, Series B **63**(1), 65 – 110 (1995)
54. Schoeberl, M.: JOP: A Java Optimized Processor. In: R. Meersman, Z. Tari (eds.) Proc. International Workshop On The Move to Meaningful Internet Systems (OTM), pp. 346–359. Springer Berlin Heidelberg (2003)
55. Souyris, J., Pavec, E.L., Himbert, G., Jégu, V., Borios, G., Heckmann, R.: Computing the Worst Case Execution Time of an Avionics Program By Abstract Interpretation. In: Proc. 5th Intl Workshop on Worst-Case Execution Time (WCET) Analysis (2005)
56. Sun Microsystems Inc.: The SPARC Architecture Manual, Version 7. Sun Microsystems Inc. (1987)
57. Weiser, M.: Program slicing. In: S. Jeffrey, L.G. Stucki (eds.) Proc. 5th International Conference on Software Engineering (ICSE), pp. 439–449. IEEE Computer Society (1981)
58. Wilhelm, R.: Why AI + ILP is good for WCET, but MC is not, nor ILP alone. In: B. Steffen, G. Levi (eds.) Proc. 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI), *Lecture Notes in Computer Science*, vol. 2937, pp. 309–322. Springer (2004)
59. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The Worst-Case Execution Time Problem – Overview of Methods and Survey of Tools. ACM Trans. Embed. Comput. Syst. **7**(3), 36:1–36:53 (2008)
60. Zhao, W., Kulkarni, P.A., Whalley, D.B., Healy, C.A., Mueller, F., Uh, G.: Tuning the WCET of embedded applications. In: Proc. 10th Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 472–481. IEEE Computer Society (2004)