LLVMTA: An LLVM-Based WCET Analysis Tool

Sebastian Hahn
Saarland University, Saarland Informatics Campus[1], Saarbrücken, Germany

Michael Jacobs
Saarland University, Saarland Informatics Campus[2], Saarbrücken, Germany

Nils Hölscher
TU Dortmund University, Germany

Kuan-Hsun Chen
University of Twente, The Netherlands

Jian-Jia Chen
TU Dortmund University, Germany

Jan Reineke
Saarland University, Saarland Informatics Campus, Saarbrücken, Germany

Abstract

We present LLVMTA, an academic WCET analysis tool based on the LLVM compiler infrastructure. It aims to enable the evaluation of novel WCET analysis approaches in a state-of-the-art analysis framework without dealing with the complexity of modeling real-world hardware architectures. We discuss the main design decisions and interfaces that allow to implement new analysis approaches. Finally, we highlight various existing research projects whose evaluation has been enabled by LLVMTA.

2012 ACM Subject Classification Software and its engineering → Software performance; Software and its engineering → Automated static analysis; Software and its engineering → Compilers; Computer systems organization → Real-time system architecture

Keywords and phrases WCET analysis, low-level analysis, LLVM

Digital Object Identifier 10.4230/OASIcs.WCET.2022.2

Supplementary Material LLVMTA and a patched version of LLVM are available as open source for academic research purposes:
Software (Source Code): `https://gitlab.cs.uni-saarland.de/reineke/llvmta`
Software (Source Code): `https://gitlab.cs.uni-saarland.de/reineke/llvm`

Funding This work has been supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the Transregional Collaborative Research Centre SFB/TR 14 (AVACS); by the Saarbrücken Graduate School of Computer Science, which received funding from the DFG as part of the Excellence Initiative of the German Federal and State Governments; and by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 101020415).

Acknowledgements We would like to thank all the students at Saarland University who contributed during the development of LLVMTA: in particular Claus Faymonville for his extensive contributions in a very early development stage, but also Tina Jung, Darshit Shah, and Tobias Stark.

[1] Work underlying this paper was performed between 2014 and 2019 while the author was still working at Saarland University. Since 2019, the author is affiliated with AbsInt Angewandte Informatik GmbH which is not related to work described in this paper.
[2] Work underlying this paper was performed between 2014 and 2017 while the author was still working at Saarland University. Since 2018, the author is affiliated with Artengis GmbH which is not related to work described in this paper.

1 Introduction

In this paper, we introduce LLVMTA, an open-source worst-case execution time (WCET) analysis tool developed at Saarland University from 2014 onwards. Our aims with this paper are twofold: First, we want to convey the main design goals of LLVMTA and how these design goals manifest in its current design. Second, to facilitate future work on LLVMTA, we describe the most important interfaces, which need to be implemented to adapt and extend LLVMTA.

At the onset of the work on LLVMTA, four WCET analysis tools were at the disposal of the authors: the commercial WCET analyzer aiT by AbsInt GmbH, and the three academic WCET analysis tools OTAWA [7], developed in Toulouse, Chronos [49] developed in Singapore, and Heptane [38], developed in Rennes. Our goals at the time were to study WCET analysis for microarchitectures that feature timing anomalies [53, 64] and to explore the potential of compositional timing analysis [37]. In order to evaluate the full potential of the planned approaches, we decided to implement them in a state-of-the-art analysis framework. To the best of our knowledge, the required state-of-the-art analysis features (in particular abstract execution graphs, as discussed in Section 2) were only available in the commercial aiT tool created by Absint. Commercial analysis tools as aiT, however, exhibit a high degree of complexity as they have to support a wide range of real-world hardware platforms and need to scale to large real-world applications under analysis. Thus, we decided to create our own analysis framework from scratch, to enable the rapid prototyping of novel analysis approaches. Due to the use of state-of-the-art analysis techniques, it is still reasonable to judge whether the observed gain in precision and/or efficiency of novel analysis approaches would translate to commercial tools.

Also worth mentioning is the WCET compiler WCC [20] developed in Dortmund. WCC is a compiler focusing on WCET optimizations, implementing its own high- and machine-level intermediate representations, namely ICD-C and ICD-LLIR. WCC uses aiT for its timing analysis and does not provide timing analysis on its own. While LLVMTA is using the LLVM compiler infrastructure it does not perform code transformations on its own.

As part of T-CREST [60], a tool called Platin has been developed in Vienna. Platin also uses aiT to provide low-level timing analysis or, alternatively, an internal analyzer, which, however, does not feature a detailed microarchitectural analysis. Platin can compute loop bounds by both static analysis, using LLVM infrastructure, and by simulating short traces. These loop bounds can then be fed as flow facts to aiT.

We set out to create a new academic WCET analysis tool with these minimal requirements:

- Support of precise and accurate analysis of microarchitectures with timing anomalies using state-of-the-art techniques, in particular *abstract execution graphs* [72].
- Support of compositional analysis [37] approaches in which the analysis of different timing contributors is performed separately.
- Flexibility to easily replace pipeline or cache models or to add new path constraints.

WCET analysis is challenging for several reasons, including some that we were not particularly interested in, which entails some explicit "non-goals":

- Our goals has been to study fundamental challenges in WCET analysis, not to support particular (commercial) microarchitectures. We note, however, that given sufficient knowledge of the underlying microarchitectures, it would be possible to support particular microarchitectures within LLVMTA.
- WCET analysis typically applies to binary executables. This entails the challenge of reconstructing the control-flow graph of the code, which is not explicit in the binary. LLVMTA bypasses this challenge by integrating into the LLVM compiler.

The remainder of this paper is structured as follows: We begin by giving a brief overview of the architecture of static WCET analyzers. Next, we present an overview of the architecture of LLVMTA. Subsequently, we present the usage of the command-line tool LLVMTA for a small example. Finally, we conclude the paper with a brief discussion of existing applications of LLVMTA by pointing out future steps in the development of the tool.

2 Standard Architecture of Static WCET Analysis Tools

In this section, we describe the *de facto* standard architecture underlying static WCET analysis tools today, which in particular underlies LLVMTA. There are fundamentally different WCET analysis approaches, such as measurement and hybrid WCET analysis, which are out of scope in this discussion.

LLVMTA and other WCET analyzers operate on a control-flow graph (CFG) representation of the program under analysis. A CFG is a directed graph whose nodes correspond to basic blocks, i.e. straight-line code sequences, and whose edges correspond to possible control flow between these nodes. The CFG of a program is not explicit in the machine code executed on the hardware. Thus the first step of most WCET analysis tools is to reconstruct a CFG from the program binary [75, 76, 45, 23, 67, 8]. In LLVMTA, the CFG is directly obtained from the compiler generating the binary rather than by reconstructing it.

Given the program's CFG, the WCET analysis problem can then be decomposed into three subproblems:

1. Deriving constraints that approximate the subset of paths through the control-flow graph that are semantically feasible.
2. Determining the possible execution times of program parts, such as basic blocks, accounting for the timing effects of microarchitectural features such as pipelining and caching.
3. Combining the information from 1. and 2. to derive a bound on the program's WCET.

The first subproblem depends only on the program's semantics and is thus independent of the underlying microarchitecture. If the program under analysis contains loops, it is necessary to bound each loop's maximum number of iterations; otherwise, no WCET bound can possibly be derived. Different *loop bound analyses* have been described in the literature [31, 52, 30, 74, 15, 19, 58, 6, 10]. Generalizing loop bound analysis, *control-flow analysis* derives constraints on the possible execution paths through the CFG [46, 16, 5, 42, 68, 59, 61] including loop bounds.

By definition, the second subproblem critically depends on the underlying microarchitecture. Thus the underlying analysis is often called *microarchitectural analysis*. Traditionally, the output of microarchitectural analysis have been bounds on the timing contributions of the basic blocks [7, 38, 49] of the program. As modern processors employ pipelining, the execution of successive basic blocks may overlap substantially. Thus, it is important not to "pay" for this overlap multiple times, and to analyze basic blocks within the context of their surrounding basic blocks. Different approaches to this end have been proposed [51, 77, 39, 53, 12, 22, 17, 78, 48, 65, 82]. We are unable to give a complete account of these approaches here due to space limitations; instead we focus on a brief description of the approach taken in LLVMTA.

Microarchitectural analysis in LLVMTA can be seen as a static cycle-by-cycle simulation of the execution of the program on *abstract* microarchitectural states, accounting for any microarchitectural component that influences the execution's timing, such as pipelining (including e.g. forwarding effects), branch prediction, load and store buffers, and caches.

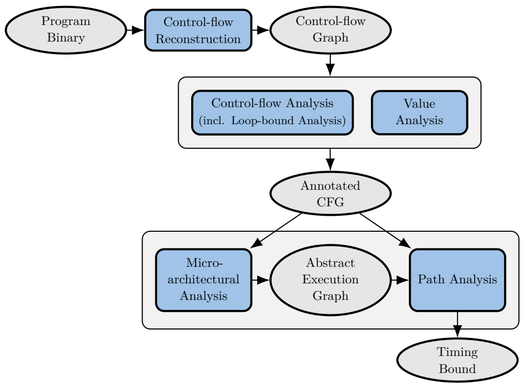

Figure 1 Overview of the general steps in WCET analysis.

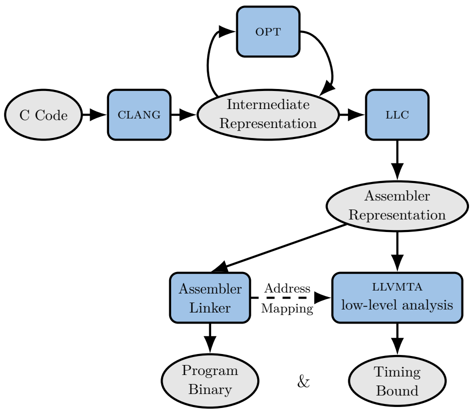

Figure 2 Overview of the common LLVM compilation flow (CLANG, OPT, LLC) including the integration of our low-level analysis tool LLVMTA.

Due to abstraction this static simulation may lack information, e.g. whether an access results in a cache hit or a cache miss, or whether two memory accesses alias, the simulation may have to “split” following multiple successor states, introducing nondeterminism. The output of microarchitectural analysis is an *abstract execution graph* (AEG) [72] whose nodes correspond to abstract microarchitectural states and whose edges correspond to the passage of processor cycles. An important distinguishing feature of this approach is that the AEG may capture correlations between the timing contributions of different basic blocks, rather than computing a single bound for each basic block. The key to make this approach successful in practice is to find abstractions that strike a good balance between analysis complexity and precision. For caches various compact abstractions have been developed [1, 25, 26, 27, 71, 33, 29, 11, 14, 28, 24, 54, 79, 9, 80] that offer varying degrees of precision depending on the underlying replacement policy [40, 63].

The third and final step of WCET analysis is to combine the information gathered in the first two steps to compute a bound on the program’s WCET. The most popular approach to this *path analysis problem* is the *implicit path enumeration technique* (IPET) [50]. The basic idea behind IPET is to solve the path analysis problem via an integer linear program (ILP). Integer variables are introduced for each edge in the AEG, encoding the frequency of taking those edges during an execution. The structure of the AEG imposes linear constraints relating these frequencies, implicitly encoding all possible paths through the AEG. Additional constraints are obtained from control-flow analysis; otherwise unbounded solutions would be possible in the presence of loops. Finally the objective function captures the cost of a given path through the AEG. Maximizing the objective function yields a safe WCET bound provided that the previous analyses capture all possible executions of the program on the microarchitecture. The overall WCET analysis flow is depicted in Figure 1.

For a more detailed discussion of static WCET analysis and related techniques we refer to the survey paper by Wilhelm et al. [81]. The same techniques can be used to safely approximate the number of occurrences of other microarchitectural events [43]. E.g. one can similarly determine a bound on the number of cache misses in any possible execution of the program.

3 LLVMTA Tool Architecture

3.1 High-level Structure

We implemented a low-level analysis tool called LLVMTA, following the scheme sketched in Figure 1. In this section, we provide details on this tool. LLVMTA is based on the LLVM compiler infrastructure [47], and it is hooked into the common LLVM compilation flow as depicted in Figure 2.

Overall Tool Architecture. Given a C program, the compiler frontend CLANG (`https://clang.llvm.org`) translates the program into the LLVM intermediate representation. After an optional optimization phase (OPT), the program is further translated to the assembler code (LLC) which results in the final binary after the linking step. Our analyses are implemented on the final assembler representation in the LLVM backend which is the representation closest to the machine level. The timing bound determined by LLVMTA is valid for the resulting binary, i.e., it will change accordingly if the binary changes, e.g. due to different compiler optimizations.

The integration of low-level timing analysis and compilation offers several advantages. First, no control-flow reconstruction of the binary is required because control-flow elements such as functions, basic blocks, and loops are provided by the prior compilation step. Second, the low-level analysis in the backend can make use of (high-level) information obtained at earlier stages and maintained during compilation. On the downside, the analysis requires as input the program to be analyzed in LLVM intermediate representation, and provides timing estimates only for the binary produced by the specific compiler. Commonly, this representation can be obtained using the compiler frontend from the high-level source program, for example given in C. It is conceivable, but it has not been experimentally validated, to apply binary lifters [4, 2, 18] to obtain an intermediate representation directly from binaries. The analysis results would then be valid for binaries obtained by recompiling the intermediate representation. Furthermore, the addresses of the instructions and the static data are only known after the linking step and would have to be fed back to the low-level analysis for sound analysis results. This is currently not implemented, but there are no major technical obstacles to doing so.

LLVMTA low-level analysis. To obtain precise results, we have implemented *context-sensitive* analysis [70], i.e. the analysis distinguishes different contexts that influence the execution behaviour. As an example, the execution behaviour of the first iteration of a loop usually differs from the behaviour of later iterations because the caches are being filled during the first iteration [55]. To establish a context-sensitive analysis framework, we implemented trace partitioning [56] on the final assembler representation in the LLVM backend. Context sensitivity is achieved by partitioning the set of execution traces according to some predicate on traces. We implemented predicates to discriminate different iterations of a loop, as well as different call sites of a function. The degree of context sensitivity, i.e. the number and size of these predicates, is an analysis parameter.

Based on our context-sensitive analysis framework, we have implemented a value analysis that tracks constant values of registers and memory cells. This value information is used to derive address information for data accesses. Despite the simplicity of the analysis domain, it is sufficient to precisely analyse stack-relative accesses. For accesses to globally defined objects such as global arrays, our tool uses information provided by the compiler to determine the range of possible addresses.

In order to derive loop bounds, we use the LLVM-internal scalar evolution analysis that provides an upper bound on the iteration count of loops in their intermediate representation. Our tool matches loops in the assembler representation to loops in intermediate representation in order to automatically obtain upper loop bounds on the assembler level. Manual loop annotations can be provided by the user for loops with complex iteration patterns. The scalar evolution analysis, originally based on [6] and extended in [10], computes a closed-form expression to describe how the values of variables evolve within a single loop iteration. These expressions are used to derive upper loop bounds, either in the form of numeric values or symbolic expressions w.r.t. the function parameters.

Our tool supports the analysis of different generic hardware platforms rather than proprietary industrial platforms for the ARM and the RISC-V instruction sets. This is sufficient to evaluate the general concepts used in timing analysis and takes significantly less effort to implement. We model textbook pipelines (see [41]) with in-order, strictly in-order [35, 32, 36], and out-of-order execution. The microarchitectural analysis supports scratchpad memories, as well as caches with least-recently-used replacement policy and both write-through and write-back policy. We have implemented must, may, and persistence cache analysis [1, 54, 62]. As background memory, the tool supports fixed-latency memory as well as dynamic random-access memory with a closed-page controller and distributed refreshes.

LLVMTA implements the *fast-forwarding* technique presented in [44] to increase the performance of the microarchitectural analysis. This optimization exploits the fact that pipelines tend to *converge* while waiting for memory, i.e. the pipeline cannot advance further until the current memory request is finished. Once converged, the (abstract) state of the pipeline stays the same as long as the memory is busy.

The abstract execution graph produced by the microarchitectural analysis is compressed afterwards. LLVMTA supports two different levels of compression. Either all start and end nodes within a basic block are kept separate to allow for a precise path analysis [72], or the graph is compressed into a single edge per basic block to allow for an efficient path analysis. Our tool supports multiple solvers to solve the ILP formulation resulting from the path analysis, including the commercial tools IBM ILOG CPLEX Optimization Studio (`https://www.ibm.com/us-en/marketplace/ibm-ilog-cplex`) and Gurobi Optimizer (`https://www.gurobi.com`) that exhibit the best performance [57].

3.2 Limitations

While offering many advantages such as code reuse and flexibility, the nature of an academic prototype and the tight coupling with a compiler infrastructure also comes with limitations.

LLVMTA operates on the machine-level IR rather than on the binary, which may yield results that are not entirely faithful to the generated machine code for the following reasons. The assembler may break down pseudo-assembly instructions used in the machine-level IR – the level we perform the analysis on – into several machine instructions in the actual binary (especially on RISC-V). The address mapping is only determined after linking, and LLVMTA currently operates on a made-up address mapping. We note that it would be possible to obtain a faithful address mapping from the linker. For the ARM instruction set, predication is supported for branch instructions, but not for arbitrary machine instructions as specified in the instruction set architecture. As mentioned earlier, there are currently no microarchitectural models that correspond to existing commercial hardware designs. Finally, the tool is reasonable fast on the standard WCET benchmarks, but will likely not scale to real-world applications.

3.3 Main Design Interfaces

To reach our goal of flexibility, we use shared interfaces. New low-level analyses can be obtained by implementing these interfaces with new classes (possibly inheriting existing ones) and directing the analysis framework to employ these implementations. The most important interfaces of LLVMTA allow for:

- static program analysis on machine-level LLVM intermediate representation,
- microarchitectural analysis, in particular pipeline modeling,
- cache analysis, and
- additional path analysis constraints.

3.3.1 Program Analysis at Machine-level Intermediate Representation

The interface class `ContextAwareAnalysisDomain` enables context-sensitive analysis on a control-flow graph with machine-level instructions.

Part of the interface specifies the basic operation on abstract domain values in the spirit of abstract interpretation [13]:

- `isBottom`: does the current abstract value represent the bottom element of the analysis lattice?
- `lessequal` compares the current abstract value with another given one w.r.t. the partial order $\sqsubseteq$ of the analysis domain,
- `join` joins a given abstract value into the current abstract value w.r.t. the partial order of the analysis domain. The behaviour should be consistent with `lessequal`.

The second part of interface specifies the transfer behaviour of abstract values while abstractly interpreting the control-flow graph of the program under analysis.

- `transfer` takes the next instruction to analyze, the current analysis context, and, optionally, analysis information of preceding static analyses at this program point. It modified the current abstract value by the effect of the instruction in the specified context.
- `guard` can be used to sharpen the current abstract value by the knowledge of the outcome of a branch instruction (either taken or not taken).
- `enterBasicBlock` models the effect of entering a basic block on the analysis information.

3.3.2 Microarchitectural Analysis

The interface class `MicroArchitecturalState` models the abstract state of the microarchitecture under analysis. Microarchitectural analysis, unlike most program analysis techniques operates at the granularity of processor cycles rather than program instructions. `MicroArchitecturalState` has the following interface:

- The constructor of the class creates the initial microarchitectural state from which the state space exploration starts. This usually represents a state with an empty pipeline and unknown cache contents.
- `cycle` models the behavior of executing the machine for a single cycle. The abstract microarchitectural state is modified in-place. It takes a configurable set of precomputed analysis information, e.g. address information for memory-accessing instructions.
- `isFinal` specifies whether a given instruction has just finished execution in the current microarchitectural state. An instruction is hereby identified by its instruction address and a context. This predicate allows use to map microarchitectural states to instructions in the control-flow graph of the program under analysis. This is mostly a technicality, but influences e.g. at which points during the analysis microarchitectural states can be *joined*.
- `isJoinable` and `join` are used to test whether microarchitectural states can be joined - and do so if it is possible.

The interface class also features helper functions to model common functionality found in any microarchitecture such as the program counter and its evolution during program execution.

The `cycle` behaviour implicitly induces an microarchitectural state graph from the initial state up to states in which the last instruction of the program under analysis finished execution. At cycle granularity, i.e. having one edge per single execution cycle, such graphs are too large to be used in path analysis. In Section 3.3.4 below, we will describe how to obtain a more compact graph where edges are collapsed to describe multiple execution cycles at once.

3.3.3 Cache Analysis

An important part of the microarchitecture that needs attention during timing analysis are caches. The interface class `AbstractCache` models the abstract cache state of the cache under analysis and specifies the behaviour of the cache replacement policy.

- `update` models the effect of loading from or storing to a given abstract address, usually an address interval.
- `lessequal` and `join` specify the abstract analysis domain by providing basic lattice operations $\sqsubseteq$ and $\sqcup$.

For cache analysis with local classifications [54], the following functions are also relevant:

- `classify` tells for a given abstract address whether an access is guaranteed to hit or miss the cache.

For persistence cache analysis [62], the following functions are also relevant:

- `enterScope` and `leaveScope` model the behaviour on entering and leaving a persistence scope.
- `getPersistentScopes` determines for a given abstract address the scopes in which the address is persistent.

3.3.4 Path Analysis

To perform path analysis on the results of microarchitectural analysis, there are mostly two tasks to perform. The first is to determine a compact micorarchitectural state graph with what we call *weights* for each edge – collapsing multiple execution cycles at once. Weights can be as simple as numeric values such as the number of cycles, the number of cache misses, etc., but also more complex things such as the set of persistent cache misses. The second task is, to use these weights to build up the constraints of an integer linear program that encodes the worst-case path through the microarchitectural state graph. While solutions to the second task are rather specific to the constraints to be generated, solutions to the first task share sufficient commonalities to be captured by an interface class.

During construction of the microarchitectural state graph, we traverse the implicit graph at cycle granularity as described in Section 3.3.2. To keep the graph compact, LLVMTA joins nodes, i.e. microarchitectural states, and edges where possible (see `isJoinable`) without losing too much precision. The general interface class to construct the weights on these collapsed edges is `StateGraphEdgeWeightProvider`. There is a simpler but more limited interface class for numeric weights only, called `StateGraphNumericEdgeWeightProvider`.

- `extractWeight` takes a part of a microarchitectural state called *LocalMetrics* and extracts a weight (of any type) from it. The LocalMetrics can accumulate any weight within a state during microarchitectural analysis, e.g. the number of cycles executed or the number of cache misses since entering a basic block.
- `joinWeight` combines weights of to-be-collapsed edges with same source and same target node, e.g. by taking the maximum.

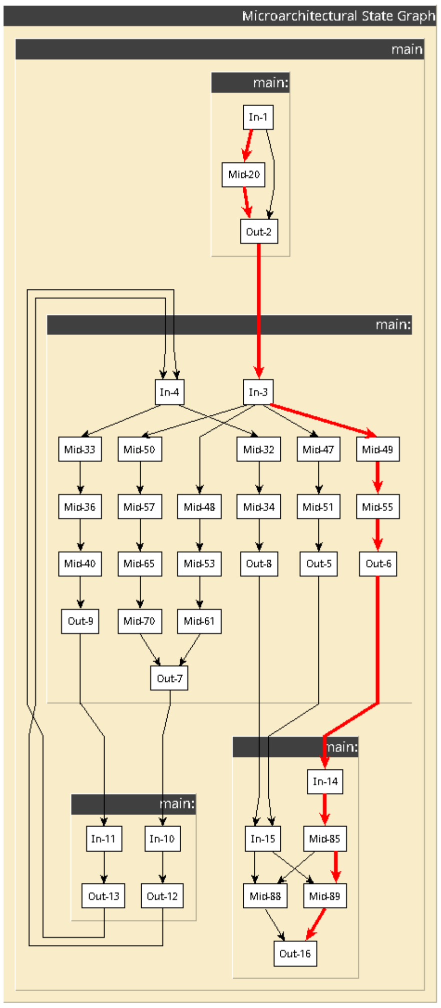

Figure 3 Simple AEG visualized with yComp.

- `concatWeight` concatenates two weights of two consecutive to-be-collapsed edges, e.g. by adding two numeric weights.
- `getNeutralWeight` returns the neutral element w.r.t. operation `concatWeight`.

4 Using LLVMTA

In this section, we shortly discuss the usage and the expected output of LLVMTA.

As input, a program written in C is provided to LLVMTA. CLANG translates the C program to its machine intermediate representation that is specific to the selected target architecture (ARM or RISC-V). The `runtestcase` script takes care of all necessary compilation steps and is the main script to perform timing analysis using LLVMTA. LLVMTA can automatically derive loop bounds using LLVM's scalar evolution analysis in many cases. If LLVMTA fails to obtain a loop bound, the user has to provide loop bounds manually in a CSV format `LoopAnnotations.csv`. The file can be auto generated with placeholders to fill the bounds using the `-ta-output-unknown-loops` option.

Table 1 Output files generated by LLVMTA.

Compiler Frontend	Assembler (`.txt`)	Program in LLVM's machine intermediate representation
	Assembler (`.S`)	Unlinked program assembly
Preanalysis	AnnotatedHeuristics	Contains inserted partitioning directives guiding context-sensitive analysis
	PersistenceScopes	Start and end of all persistence scopes
	CallGraph	List of all statically known callers and callees
	ConstantValueAnalysis	Constant values determined for machine registers and memory cells for each instruction
	LoopBounds	Contains loop bounds determined
	AddressInformation	Addresses of instructions and addresses of data accessed by memory operations
Microarch. Analysis	MicroArchAnalysis	Invariant set of microarchitectural state at the beginning and end of each basic block
	StateGraph_Time (`.vcg`)	Microarchitectural state graph
Path Analysis	LongestPath	ILP path analysis formulation
	PathAnalysis_<Weight> ..._<Max|Min>	Detailed results of the path analysis including worst-case path
Results	TotalBound (`.xml`)	Machine readable output of calculated bounds
	Statistics	Resource consumption of analysis

During the analysis execution, intermediate results are printed as the different analysis stages are performed. The output files are listed in Table 1. These files help the user understand what happens during the analysis and to inspect intermediate and final results. One important output is the file `StateGraph_Time`, which contains the abstract execution graph with microarchitectural states as nodes and edges with weights such as timing or number of cache misses. The graph can be viewed with the `yComp` (`https://pp.ipd.kit.edu/firm/yComp.html`) graph viewer [66], as shown in Figure 3 for the "simplewhile" example, which is provided along with the test suite of LLVMTA. The example consists of a single while loop, incrementing a single variable 50 times before terminating. The WCEP is highlighted by the red edges. It is worth noting that multiple abstract microarchitectural states are created for the while loop in the main function allowing for higher analysis precision than a simple "single execution time per basic block" path analysis scheme.

5 Existing Research Applications of LLVMTA

In this section, we briefly summarize research carried out with the help of LLVMTA.

Cache Analysis. Classically, *write-back caches* have not been used in hard real-time systems as it was not known how to model them in a sufficiently precise way during WCET analysis. This gap has been closed by combining two perspectives: a store-focussed one, which answers whether a store may dirtify a clean cache line, and an eviction-focussed one, which answers whether a cache miss may evict a dirty cache line and thus cause a write back [9].

We also employed LLVMTA in the development and evaluation of exact cache analysis, in particular of exact cache persistence analyses [73].

Compositional Analysis. LLVMTA supports compositional analysis approaches, i.e. several weights can be chosen for maximization such as useful cache blocks or accesses to the shared bus. The impact of such independent maximizations of different metrics on the efficiency and precision of WCET analysis has been explored in a master's thesis [21]. In addition, LLVMTA can sample *interference response curves* and calculate *compositional base bounds* based on the results of a microarchitectural analysis that safely covers all possible cases of temporal interference [34]. In this way, it bridges the gap between schedulability analyses, which typically rely on timing compositionality, and modern microarchitectures, which typically exhibit timing anomalies.

Calculation of Interference on Shared Resources. In the context of WCET analysis for multi-core processors, an important quantity is the amount of shared-resource interference that a concurrent processor core can generate while the program under WCET analysis is executed. To safely overapproximate worst-case interference generation scenarios, we generalized the well-known *implicit path enumeration technique* (IPET) [50] in a way that takes into account arbitrary subpaths of the abstract execution graph for the concurrent processor core [44]. As we assume that this generalized IPET does not scale to multiple real-world programs executed on a concurrent processor core, we sketched a *program-modular* calculation scheme [43] that calculates compositional base bounds per program on top of the generalized IPET.

Cache-Related Preemption Delay. In the presence of preemptive scheduling, preempting tasks evict cached memory blocks of preempted tasks, which have to be reloaded when the preempted tasks resume their execution [69]. This is commonly referred to as *cache-related preemption delay* (CRPD) [3]. We have experimentally evaluated the state-of-the-art techniques used to account for CRPD during timing analysis. Our experiments used task sets obtained by running LLVMTA on actual benchmarks. It turned out that the difference in precision of different CRPD analysis techniques and the overall impact of CRPD on schedulability are not as significant as observed for purely synthetically-generated task sets [69].

Strictly In-order Pipeline. To enable efficient and precise microarchitectural analysis, we introduced the strictly in-order pipeline in [35, 36]. This pipeline design enables an efficient progress-based abstraction and allows quantification of the effect of (amplifying) timing anomalies. The effect on WCET estimates and analysis performance have been evaluated using LLVMTA.

6 Conclusions

Since the kickoff of LLVMTA in 2014, it has been employed and extended in various research projects at Saarland University. It was exposed to TU Dortmund University in 2020. With the open source release of the code base in a version control system, and steady maintenance and integration, we foresee a great potential for LLVMTA to be used in research and education on WCET analysis more globally. Our short-term plan is to release practical exercises and tutorials for use of LLVMTA in education at the graduate level.

LLVMTA and a patched version of LLVM are available as open source for academic research purposes at:

`https://gitlab.cs.uni-saarland.de/reineke/llvmta`
`https://gitlab.cs.uni-saarland.de/reineke/llvm`

References

1 Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache behavior prediction by abstract interpretation. In *Proceedings of the Third International Static Analysis Symposium, SAS 1996, Aachen, Germany*, pages 52–66, 1996. doi:10.1007/3-540-61739-6_33.

2 Anil Altinay, Joseph Nash, Taddeus Kroes, Prabhu Rajasekaran, Dixin Zhou, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, Cristiano Giuffrida, Herbert Bos, and Michael Franz. BinRec: dynamic binary lifting and recompilation. In Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer, editors, *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 36:1–36:16. ACM, 2020. doi:10.1145/3342195.3387550.

3 Sebastian Altmeyer. *Analysis of preemptively scheduled hard real-time systems*. PhD thesis, Saarland University, 2013. URL: http://scidok.sulb.uni-saarland.de/volltexte/2013/5279/.

4 Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. A compiler-level intermediate representation based binary analysis and rewriting system. In Zdenek Hanzálek, Hermann Härtig, Miguel Castro, and M. Frans Kaashoek, editors, *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*, pages 295–308. ACM, 2013. doi:10.1145/2465351.2465380.

5 Mihail Asavoae, Claire Maiza, and Pascal Raymond. Program semantics in model-based WCET analysis: A state of the art perspective. In Claire Maiza, editor, *13th International Workshop on Worst-Case Execution Time Analysis, WCET 2013, July 9, 2013, Paris, France*, volume 30 of *OASIcs*, pages 32–41. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013. doi:10.4230/OASIcs.WCET.2013.32.

6 Olaf Bachmann, Paul S. Wang, and Eugene V. Zima. Chains of recurrences - a method to expedite the evaluation of closed-form functions. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation, ISSAC 1994, Oxford, UK*, pages 242–249, 1994. doi:10.1145/190347.190423.

7 Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: an open toolbox for adaptive WCET analysis. In Sang Lyul Min, Robert G. Pettit IV, Peter P. Puschner, and Theo Ungerer, editors, *Software Technologies for Embedded and Ubiquitous Systems - 8th IFIP WG 10.2 International Workshop, SEUS 2010, Waidhofen/Ybbs, Austria, October 13-15, 2010. Proceedings*, volume 6399 of *Lecture Notes in Computer Science*, pages 35–46. Springer, 2010. doi:10.1007/978-3-642-16256-5_6.

8 Sébastien Bardin, Philippe Herrmann, and Franck Védrine. Refinement-based CFG reconstruction from unstructured programs. In Ranjit Jhala and David A. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, volume 6538 of *Lecture Notes in Computer Science*, pages 54–69. Springer, 2011. doi:10.1007/978-3-642-18275-4_6.

9 Tobias Blaß, Sebastian Hahn, and Jan Reineke. Write-back caches in WCET analysis. In *Proceedings of the 29th Euromicro Conference on Real-Time Systems, ECRTS 2017*, June 2017.

10 Silvian Calman and Jianwen Zhu. Interprocedural induction variable analysis based on interprocedural SSA form IR. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE 2010, Toronto, Ontario, Canada*, pages 37–44, 2010. doi:10.1145/1806672.1806680.

11 Sudipta Chattopadhyay and Abhik Roychoudhury. Scalable and precise refinement of cache timing analysis via path-sensitive verification. *Real Time Syst.*, 49(4):517–562, 2013. doi:10.1007/s11241-013-9178-0.

12 Antoine Colin and Isabelle Puaut. A modular & retargetable framework for tree-based WCET analysis. In *13th Euromicro Conference on Real-Time Systems (ECRTS 2001), 13-15 June 2001, Delft, The Netherlands, Proceedings*, pages 37–44. IEEE Computer Society, 2001. doi:10.1109/EMRTS.2001.933995.

13 Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA*, pages 238–252, 1977. doi:10.1145/512950.512973.

14 Christoph Cullmann. Cache persistence analysis: Theory and practice. *ACM Transactions on Embedded Computing Systems*, 12(1s):40:1–40:25, 2013. doi:10.1145/2435227.2435236.

15 Christoph Cullmann and Florian Martin. Data-flow based detection of loop bounds. In *Proceedings of the 7th International Workshop on Worst-Case Execution Time Analysis, WCET 2007, Pisa, Italy*, 2007. doi:10.4230/OASIcs.WCET.2007.1193.

16 Sun Ding, Hee Beng Kuan Tan, and Kaiping Liu. A survey of infeasible path detection. In Joaquim Filipe and Leszek A. Maciaszek, editors, *ENASE 2012 - Proceedings of the 7th International Conference on Evaluation of Novel Approaches to Software Engineering, Wroclaw, Poland, 29-30 June, 2012*, pages 43–52. SciTePress, 2012.

17 Jakob Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Sweden, 2002. URL: http://nbn-resolving.de/urn:nbn:se:uu:diva-1832.

18 Alexis Engelke, Dominik Okwieka, and Martin Schulz. Efficient LLVM-based dynamic binary translation. In Ben L. Titzer, Harry Xu, and Irene Zhang, editors, *VEE '21: 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, Virtual USA, April 16, 2021*, pages 165–171. ACM, 2021. doi:10.1145/3453933.3454022.

19 Andreas Ermedahl, Christer Sandberg, Jan Gustafsson, Stefan Bygde, and Björn Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In *Proceedings of the 7th International Workshop on Worst-Case Execution Time Analysis, WCET 2007, Pisa, Italy*, 2007. doi:10.4230/OASIcs.WCET.2007.1194.

20 Heiko Falk and Paul Lokuciejewski. A compiler framework for the reduction of worst-case execution times. *Real Time Syst.*, 46(2):251–300, 2010. doi:10.1007/s11241-010-9101-x.

21 Claus Faymonville. Evaluating compositional timing analyses. Master's thesis, Saarland University, 2015.

22 Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and precise WCET determination for a real-life processor. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *Embedded Software, First International Workshop, EMSOFT 2001, Tahoe City, CA, USA, October, 8-10, 2001, Proceedings*, volume 2211 of *Lecture Notes in Computer Science*, pages 469–485. Springer, 2001. doi:10.1007/3-540-45449-7_32.

23 Andrea Flexeder, Bogdan Mihaila, Michael Petter, and Helmut Seidl. Interprocedural control flow reconstruction. In *Proceedings of the 8th Asian Symposium on Programming Languages and Systems, APLAS 2010, Shanghai, China*, pages 188–203, 2010. doi:10.1007/978-3-642-17164-2_14.

24 David Griffin, Benjamin Lesage, Alan Burns, and Robert I. Davis. Lossy compression for worst-case execution time analysis of PLRU caches. In Mathieu Jan, Belgacem Ben Hedia, Joël Goossens, and Claire Maiza, editors, *22nd International Conference on Real-Time Networks and Systems, RTNS '14, Versailles, France, October 8-10, 2014*, page 203. ACM, 2014. doi:10.1145/2659787.2659807.

25 Daniel Grund and Jan Reineke. Abstract interpretation of FIFO replacement. In Jens Palsberg and Zhendong Su, editors, *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings*, volume 5673, pages 120–136. Springer, 2009. doi:10.1007/978-3-642-03237-0_10.

26 Daniel Grund and Jan Reineke. Precise and efficient FIFO-replacement analysis based on static phase detection. In *22nd Euromicro Conference on Real-Time Systems, ECRTS 2010, Brussels, Belgium, July 6-9, 2010*, pages 155–164. IEEE Computer Society, 2010. doi:10.1109/ECRTS.2010.8.

27 Daniel Grund and Jan Reineke. Toward precise PLRU cache analysis. In Björn Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis, WCET 2010, July 6, 2010, Brussels, Belgium*, volume 15 of *OASICS*, pages 23–35. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2010. doi:10.4230/OASIcs.WCET.2010.23.

28 Nan Guan, Mingsong Lv, Wang Yi, and Ge Yu. WCET analysis with MRU cache: Challenging lru for predictability. *ACM Trans. Embed. Comput. Syst.*, 13(4s):123:1–123:26, April 2014. doi:10.1145/2584655.

29 Nan Guan, Xinping Yang, Mingsong Lv, and Wang Yi. FIFO cache analysis for WCET estimation: a quantitative approach. In Enrico Macii, editor, *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013*, pages 296–301. EDA Consortium San Jose, CA, USA / ACM DL, 2013. doi:10.7873/DATE.2013.073.

30 Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *Proceedings of the 27th IEEE Real-Time Systems Symposium, RTSS 2006, Rio de Janeiro, Brazil*, pages 57–66, 2006. doi:10.1109/RTSS.2006.12.

31 Jan Gustafsson, Björn Lisper, Christer Sandberg, and Nerina Bermudo. A tool for automatic flow analysis of C-programs for WCET calculation. In *Proceedings of the 8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems, WORDS 2003, Guadalajara, Mexico*, pages 106–112, 2003. doi:10.1109/WORDS.2003.1218072.

32 Sebastian Hahn. *On static execution-time analysis.* PhD thesis, Saarland University, Saarbrücken, Germany, 2019. URL: https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/27440.

33 Sebastian Hahn and Daniel Grund. Relational cache analysis for static timing analysis. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems, ECRTS 2012, Pisa, Italy*, pages 102–111, 2012. doi:10.1109/ECRTS.2012.14.

34 Sebastian Hahn, Michael Jacobs, and Jan Reineke. Enabling compositionality for multicore timing analysis. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems, RTNS 2016, Brest, France*, pages 299–308, 2016. doi:10.1145/2997465.2997471.

35 Sebastian Hahn and Jan Reineke. Design and analysis of SIC: A provably timing-predictable pipelined processor core. In *2018 IEEE Real-Time Systems Symposium, RTSS 2018, Nashville, TN, USA, December 11-14, 2018*, pages 469–481. IEEE Computer Society, 2018. doi:10.1109/RTSS.2018.00060.

36 Sebastian Hahn and Jan Reineke. Design and analysis of SIC: a provably timing-predictable pipelined processor core. *Real Time Syst.*, 56(2):207–245, 2020. doi:10.1007/s11241-019-09341-z.

37 Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. Towards compositionality in execution time analysis: definition and challenges. *SIGBED Review*, 12(1):28–36, 2015. doi:10.1145/2752801.2752805.

38 Damien Hardy, Benjamin Rouxel, and Isabelle Puaut. The Heptane static worst-case execution time estimation tool. In Jan Reineke, editor, *17th International Workshop on Worst-Case Execution Time Analysis, WCET 2017, June 27, 2017, Dubrovnik, Croatia*, volume 57 of *OASIcs*, pages 8:1–8:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/OASIcs.WCET.2017.8.

39 Christopher A. Healy, Robert D. Arnold, Frank Mueller, David B. Whalley, and Marion G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Trans. Computers*, 48(1):53–70, 1999. doi:10.1109/12.743411.

40 Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, 2003. doi:10.1109/JPROC.2003.814618.

41 John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach (5. ed.).* Morgan Kaufmann, 2012.

42 Julien Henry, Mihail Asavoae, David Monniaux, and Claire Maiza. How to compute worst-case execution time by optimization modulo theory and a clever encoding of program semantics. In Youtao Zhang and Prasad Kulkarni, editors, *SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems 2014, LCTES '14, Edinburgh, United Kingdom - June 12 - 13, 2014*, pages 43–52. ACM, 2014. doi:10.1145/2597809.2597817.

43 Michael Jacobs. *Design and Implementation of WCET Analyses: Including a Case Study on Multi-Core Processors with Shared Buses.* PhD thesis, Saarland University, 2021. doi:10.22028/D291-34893.

44 Michael Jacobs, Sebastian Hahn, and Sebastian Hack. WCET analysis for multi-core processors with shared buses and event-driven bus arbitration. In *Proceedings of the 23nd International Conference on Real-Time Networks and Systems, RTNS 2015, Lille, France*, 2015.

45 Johannes Kinder, Florian Zuleger, and Helmut Veith. An abstract interpretation-based framework for control flow reconstruction from binaries. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2009, Savannah, GA, USA*, pages 214–228, 2009. doi:10.1007/978-3-540-93900-9_19.

46 Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. Symbolic loop bound computation for WCET analysis. In Edmund M. Clarke, Irina B. Virbitskaite, and Andrei Voronkov, editors, *Perspectives of Systems Informatics - 8th International Andrei Ershov Memorial Conference, PSI 2011, Novosibirsk, Russia, June 27-July 1, 2011, Revised Selected Papers*, volume 7162 of *Lecture Notes in Computer Science*, pages 227–242. Springer, 2011. doi:10.1007/978-3-642-29709-0_20.

47 Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the Second IEEE / ACM International Symposium on Code Generation and Optimization, CGO 2004, San Jose, CA, USA*, pages 75–88, 2004. doi:10.1109/CGO.2004.1281665.

48 Xianfeng Li, Abhik Roychoudhury, and Tulika Mitra. Modeling out-of-order processors for WCET analysis. *Real-Time Systems*, 34(3):195–227, 2006. doi:10.1007/s11241-006-9205-5.

49 Xianfeng Li, Liang Yun, Tulika Mitra, and Abhik Roychoudhury. Chronos: A timing analyzer for embedded software. *Sci. Comput. Program.*, 69(1-3):56–67, 2007. doi:10.1016/j.scico.2007.01.014.

50 Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, & Tools for Real-Time Systems, LCT-RTS 1995, La Jolla, California*, pages 88–98, 1995. doi:10.1145/216636.216666.

51 Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon, and Chong-Sang Kim. An accurate worst case timing analysis for RISC processors. *IEEE Trans. Software Eng.*, 21(7):593–604, 1995. doi:10.1109/32.392980.

52 Björn Lisper. SWEET - A tool for WCET flow analysis (extended abstract). In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part II*, volume 8803 of *Lecture Notes in Computer Science*, pages 482–485. Springer, 2014. doi:10.1007/978-3-662-45231-8_38.

53 Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium, RTSS 1999, Phoenix, AZ, USA*, pages 12–21, 1999. doi:10.1109/REAL.1999.818824.

54 Mingsong Lv, Nan Guan, Jan Reineke, Reinhard Wilhelm, and Wang Yi. A survey on static cache analysis for real-time systems. *Leibniz Transactions on Embedded Systems LITES*, 3(1):05:1–05:48, 2016. doi:10.4230/LITES-v003-i001-a005.

55 Florian Martin, Martin Helmut Alt, Reinhard Wilhelm, and Christian Ferdinand. Analysis of loops. In Kai Koskimies, editor, *Compiler Construction, 7th International Conference, CC'98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1383 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 1998. `doi:10.1007/BFb0026424`.

56 Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In *Proceedings of the 14th European Symposium on Programming, ESOP 2005, Edinburgh, UK*, pages 5–20, 2005. `doi:10.1007/978-3-540-31987-0_2`.

57 Bernhard Meindl and Matthias Templ. Analysis of commercial and free and open source solvers for linear optimization problems. Technical report, Institut für Statistik und Wahrscheinlichkeitstheorie, Vienna University of Technology, 2012.

58 Marianne De Michiel, Armelle Bonenfant, Hugues Cassé, and Pascal Sainrat. Static loop bound analysis of C programs based on flow analysis and abstract interpretation. In *The Fourteenth IEEE Internationl Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2008, Kaohisung, Taiwan, 25-27 August 2008, Proceedings*, pages 161–166. IEEE Computer Society, 2008. `doi:10.1109/RTCSA.2008.53`.

59 Vincent Mussot, Jordy Ruiz, Pascal Sotin, Marianne De Michiel, and Hugues Cassé. Expressing and exploiting conflicts over paths in WCET analysis. In Martin Schoeberl, editor, *16th International Workshop on Worst-Case Execution Time Analysis, WCET 2016, July 5, 2016, Toulouse, France*, volume 55 of *OASIcs*, pages 3:1–3:11. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. `doi:10.4230/OASIcs.WCET.2016.3`.

60 Peter P. Puschner, Daniel Prokesch, Benedikt Huber, Jens Knoop, Stefan Hepp, and Gernot Gebhard. The T-CREST approach of compiler and WCET-analysis integration. In *16th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, ISORC 2013, Paderborn, Germany, June 19-21, 2013*, pages 1–8. IEEE Computer Society, 2013. `doi:10.1109/ISORC.2013.6913220`.

61 Pascal Raymond, Claire Maiza, Catherine Parent-Vigouroux, Erwan Jahier, Nicolas Halbwachs, Fabienne Carrier, Mihail Asavoae, and Rémy Boutonnet. Improving WCET evaluation using linear relation analysis. *Leibniz Trans. Embed. Syst.*, 6(1):02:1–02:28, 2019. `doi:10.4230/LITES-v006-i001-a002`.

62 Jan Reineke. The semantic foundations and a landscape of cache-persistence analyses. *Leibniz Trans. Embed. Syst.*, 5(1):03:1–03:52, 2018. `doi:10.4230/LITES-v005-i001-a003`.

63 Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, November 2007. `doi:10.1007/s11241-007-9032-3`.

64 Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In *Proceedings of the 6th International Workshop on Worst-Case Execution Time Analysis, WCET 2006, Dresden, Germany*, 2006. `doi:10.4230/OASIcs.WCET.2006.671`.

65 Christine Rochange and Pascal Sainrat. A context-parameterized model for static analysis of execution times. *Trans. High Perform. Embed. Archit. Compil.*, 2:222–241, 2009. `doi:10.1007/978-3-642-00904-4_12`.

66 Georg Sander. Graph layout through the VCG tool. In Roberto Tamassia and Ioannis G. Tollis, editors, *Graph Drawing*, pages 194–205, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

67 Alexander Sepp, Bogdan Mihaila, and Axel Simon. Precise static analysis of binaries by extracting relational information. In Martin Pinzger, Denys Poshyvanyk, and Jim Buckley, editors, *18th Working Conference on Reverse Engineering, WCRE 2011, Limerick, Ireland*, pages 357–366. IEEE Computer Society, 2011. `doi:10.1109/WCRE.2011.50`.

68 Thomas Sewell, Felix Kam, and Gernot Heiser. Complete, high-assurance determination of loop bounds and infeasible paths for WCET analysis. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Vienna, Austria, April 11-14, 2016*, pages 185–195. IEEE Computer Society, 2016. `doi:10.1109/RTAS.2016.7461326`.

69 Darshit Shah, Sebastian Hahn, and Jan Reineke. Experimental Evaluation of Cache-Related Preemption Delay Aware Timing Analysis. In Florian Brandner, editor, *18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018)*, volume 63 of *OpenAccess Series in Informatics (OASIcs)*, pages 7:1–7:11, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/OASIcs.WCET.2018.7`.

70 Micha Sharir and Amir Pnueli. Two approaches of interprocedural data flow analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program flow analysis: theory and applications*, chapter 7, pages 189–233. Prentice Hall, Englewood Cliffs, New Jersey, 1981.

71 Tyler Sondag and Hridesh Rajan. A more precise abstract domain for multi-level caches for tighter WCET analysis. In *Proceedings of the 31st IEEE Real-Time Systems Symposium, RTSS 2010, San Diego, California, USA*, pages 395–404, 2010. `doi:10.1109/RTSS.2010.8`.

72 Ingmar Jendrik Stein. *ILP-based Path Analysis on Abstract Pipeline State Graphs.* PhD thesis, Saarland University, 2010.

73 Gregory Stock, Sebastian Hahn, and Jan Reineke. Cache persistence analysis: Finally exact. In *IEEE Real-Time Systems Symposium, RTSS 2019, Hong Kong, SAR, China, December 3-6, 2019*, pages 481–494. IEEE, 2019. `doi:10.1109/RTSS46320.2019.00049`.

74 Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. Efficient detection and exploitation of infeasible paths for software timing analysis. In Ellen Sentovich, editor, *Proceedings of the 43rd Design Automation Conference, DAC 2006, San Francisco, CA, USA, July 24-28, 2006*, pages 358–363. ACM, 2006. `doi:10.1145/1146909.1147002`.

75 Henrik Theiling. Extracting safe and precise control flow from binaries. In *7th International Workshop on Real-Time Computing and Applications Symposium (RTCSA 2000), 12-14 December 2000, Cheju Island, South Korea*, pages 23–30. IEEE Computer Society, 2000. `doi:10.1109/RTCSA.2000.896367`.

76 Henrik Theiling. *Control Flow Graphs for Real-Time System Analysis - Reconstruction from Binary Executables and Usage in ILP-based Path Analysis.* PhD thesis, Saarland University, 2002.

77 Henrik Theiling and Christian Ferdinand. Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium, Madrid, Spain, December 2-4, 1998*, pages 144–153. IEEE Computer Society, 1998. `doi:10.1109/REAL.1998.739739`.

78 Stephan Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models.* PhD thesis, Saarland University, 2004.

79 Valentin Touzeau, Claire Maïza, David Monniaux, and Jan Reineke. Ascertaining uncertainty for efficient exact cache analysis. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 22–40. Springer, 2017. `doi:10.1007/978-3-319-63390-9_2`.

80 Valentin Touzeau, Claire Maïza, David Monniaux, and Jan Reineke. Fast and exact analysis for LRU caches. *Proc. ACM Program. Lang.*, 3(POPL):54:1–54:29, January 2019. `doi:10.1145/3290367`.

81 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):36:1–36:53, 2008. `doi:10.1145/1347375.1347389`.

82 Stephan Wilhelm. *Symbolic representations in WCET analysis.* PhD thesis, Saarland University, 2012. URL: `http://scidok.sulb.uni-saarland.de/volltexte/2012/4914/`.