Symbolic State Traversal for WCET Analysis

Stephan Wilhelm
AbsInt GmbH and Saarland University
Saarbrücken, Germany
step@absint.com

Björn Wachter
Saarland University
Saarbrücken, Germany
bwachter@cs.uni-sb.de

ABSTRACT

Static worst-case execution time analysis of real-time tasks is based on abstract models that capture the timing behavior of the processor on which the tasks run. For complex processors, task-level execution time bounds are obtained by a state exploration which involves the abstract model and the program. Partial state space exploration is not sound. A full exploration can become too expensive. We present a novel symbolic method for WCET analysis based on abstract pipeline models which produces sound results and is scalable in terms of the considered hardware states.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems

General Terms

Reliability, Verification

Keywords

abstract interpretation, binary decision diagram, hard real time, processor models, worst-case execution time

1. INTRODUCTION

Designers of safety-critical real-time systems require safe and precise worst-case execution times (WCET) for each task. The execution time of a task depends on the execution speed of the processor on which the task runs, as well as on the executed program code and on input values. Further, complex processors implement various features to reduce the average execution time, e.g. pipelines and caches. Execution times on such processors also depend on the execution history and on the start state of the hardware [17, 28]. As a consequence, tools for WCET prediction have to cover all feasible program paths, inputs, and hardware states.

EMSOFT'09, October 12–16, 2009, Grenoble, France.

Measurement-based methods cannot guarantee full coverage [4] and, notoriously, miss worst-case situations, in particular, hardware-related timing accidents like pipeline stalls and cache misses which cause tremendous variations in execution time. In contrast, static WCET analysis tools guarantee safe upper bounds on the WCET. To this end, a timing model of the hardware is used, a finite state machine whose transitions correspond to processor clock cycles [15]. An upper bound on the execution time of the program can be obtained by counting cycles of the timing model. To predict timing accidents, an overapproximation of the set of reachable hardware states is computed for each program point.

Static WCET analysis only becomes computationally feasible in practice by using abstraction, which is applied to both the modeling of processor and program behavior. However, abstraction loses information which leads to uncertainty, e.g. it may not be possible to statically determine the exact address of a memory access. Furthermore, program inputs are not precisely known in advance. At the level of the hardware model, this lack of information is accounted for by non-deterministic choices. To be safe, the analysis has to exhaustively explore all possibilities. This can lead to state explosion making an explicit enumeration of states infeasible due to memory and computation time constraints [35].

We address the state explosion problem in static WCET analysis by storing and manipulating hardware states in a more efficient data structure based on Ordered Binary Decision Diagrams [7] (BDDs). Our work is inspired by BDD-based symbolic model checking [8]. Symbolic model checking has been successfully applied to components of processors [10, 19]. Its success sparked a general interest in BDDs and other symbolic representations. Today, BDDs are also used extensively to analyze software, e.g. in software model checking [5] and points-to analysis [3].

While the aforementioned analyses exclusively focus on *either* hardware or software, WCET analysis considers *both* the software and the processor simultaneously, which brings about quite unique challenges. The arising complexity has been alleviated by careful modeling and abstraction, and by a modular analysis architecture. However, the interplay of hardware and software and different analyses is complex, conceptually and in terms of implementation:

- the modular analysis architecture requires exchange of information between different analyzers. Program invariants must be imported and used in the symbolic state space exploration.
- to reap the benefits of BDDs, efficient encodings are needed which take into account program structure. In

particular, efficient handling of program addresses is crucial to scale to realistic programs.

This interplay between program analysis and symbolic engine, and the use of complex models of industrial processors including pipelines and buffers, sets our work apart from the aforementioned applications of symbolic techniques.
Based on previous work [31], we present a novel framework for static WCET analysis using symbolic representations of abstract pipeline models, and assess its effectiveness on a set of industrial benchmarks. We have developed a prototype implementation which is integrated into the commercial WCET analysis tool `aiT` [1]. `aiT` has successfully been applied to avionics [31, 36] and to automotive software [20] and compares very favorably with other WCET analysis tools [32]. In our prototype implementation, we employ the model of a real-life processor, the Infineon TriCore. The model was developed and tested within `aiT`. This enables a meaningful performance comparison between the two implementations, which produce the same analysis results.
To arrive at an efficient symbolic analysis that scales to industrial-size programs, we have not only incorporated well-known optimizations from symbolic model checking but also novel domain-specific optimizations that leverage properties of the processor and the program. The experimental results are promising, showing that our analysis is able to handle very large state spaces efficiently.

Our Contribution.

We present the first WCET analysis of abstract pipeline models using a symbolic representation. We combine static WCET analysis based on abstract interpretation with symbolic representations from the world of model checking, thus gaining efficiency without compromising safety. We give detailed information on the efficient integration of `aiT`'s program analysis framework with a symbolic state exploration engine. We further describe a generic optimization exploiting program locality to make the symbolic representation independent of the size of the analyzed programs. Besides, we sketch several optimizations for reducing the size of abstract pipeline models. Note that *all* of these optimizations preserve the safety and precision of the analysis. We have evaluated our approach considering a real-life architecture, the Infineon TriCore, used in the automotive domain, and an industrial engine control software.

2. RELATED WORK

Wilhelm [38] discusses the benefits of static WCET analysis based on abstract interpretation and ILP, and argues why techniques that do not employ abstraction are not able to tackle the complexity of the problem. As an example for an inadequate technique, the paper sketches how binary search and symbolic model checking could be used for WCET analysis. Thereby the symbolic model checker is used to check, for a given potential execution time bound, if there exists a longer execution, and the whole process is repeated until a smallest such bound is found. Subsequently, Metzner [26] took up this idea. He argued that analysis precision could be improved by using model checking rather than static program analysis. However, this analysis has yet to be extended to realistic programs and processor models with features like pipelining and speculation.
Logothetis and Schneider [22] proposed an analysis that obtains hardware-independent execution time bounds for synchronous programs making the unit assumption: every instruction takes one time unit. However, the unit assumption does not hold for today's microarchitectures.

3. STATIC WCET ANALYSIS

Static WCET analysis [39] employs data flow analyses based on abstract interpretation [11] to statically determine timing relevant information, e.g. the possible target addresses of memory accesses and possible hardware states and cache contents.

Data Flow Analysis.

Data flow analyses compute invariants for all program points by fixed point iteration over the control flow graph of the program. An analysis is defined in terms of:

- a *domain*, in which program invariants are expressed,
- a *transfer function* for updating information at program points,
- an *equality test* to check for fixed points of the transfer function,
- a *least upper bound operator* for combining information at control flow joins. The operator is typically associated with a loss of precision.

To obtain precise information about the executed machine code, control flow graphs for static WCET analysis are reconstructed from fully linked executables. Strictly linear sequences of machine instructions, i.e. sequences where control joins only before the first instruction and branches only after the last instruction, are folded into *basic blocks*. The resulting control flow graph is also called *basic block graph*. For interprocedural data flow analyses, the basic block graphs of all procedures are merged into a single *supergraph* that represents the entire program.
The manual implementation of interprocedural data flow analyses is intricate and error-prone. The program analyzer generator `PAG` [24] allows to generate interprocedural analyzers from an analysis specification in terms of domain, transfer function, equality test and least upper bound operator. Further, it offers flexible means to distinguish different call contexts and loop iterations by virtual inlining and unrolling [25].

The aiT WCET Analyzer.

A successful tool flow for static WCET analysis is the `aiT` WCET analyzer. Fig. 1 shows an overview of `aiT`'s architecture. `aiT` employs several data flow analyses, generated with `PAG`. *Control flow reconstruction* [34] decodes a binary executable and reconstructs its control flow and basic block graph. *Value analysis* [11] computes an overapproximation of possible register values by an interval analysis. The results are used to determine loop bounds, to eliminate infeasible paths through the program, and to determine addresses of memory accesses.
The focus of this paper is *microarchitectural analysis* [13, 14, 35], which computes upper bounds on the execution time of basic blocks. Microarchitectural analysis is a data flow analysis that performs an abstract interpretation of the program using a hardware model, i.e. it computes an overapproximation of the set of states of the hardware model that can

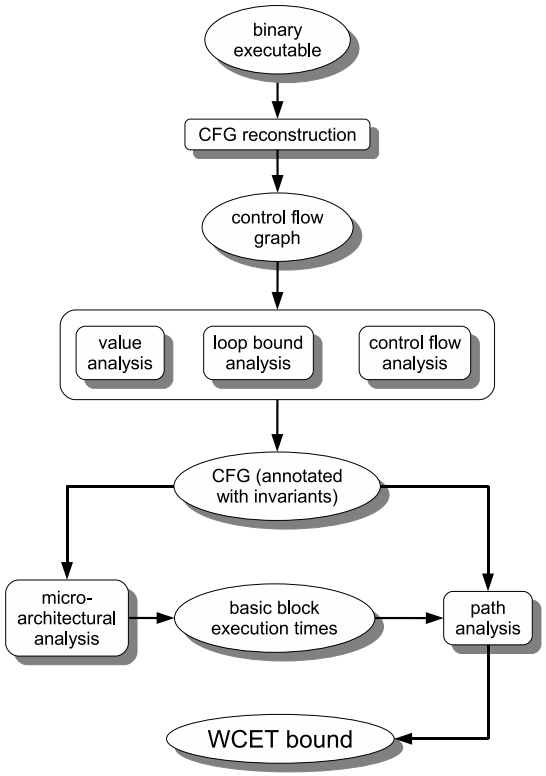

Figure 1: Architecture of the `aiT` WCET analyzer.

occur at a basic block. The hardware model accounts for timing-relevant processor components, such as pipelining, speculation, and peripheral hardware. To reduce complexity, the arithmetic in the processor and register contents are factored out into value analysis.
Finally, the results of microarchitectural analysis are passed to the final phase, *path analysis* [33] which uses implicit path enumeration [21], an ILP-based approach to determine an execution time bound for the whole program based on bounds on the execution times of basic blocks.

3.1 Microarchitectural Analysis

A microprocessor can be regarded as a very large state machine performing a state transition in every clock cycle. This state machine (or any complete and precise specification thereof) defines the concrete semantics. Each execution of a program on the hardware corresponds to a sequence of concrete states called a *trace*. The goal of the microarchitectural analysis is to determine execution time bounds for all basic blocks that hold for all executions of the program. The execution of a basic block is sensitive to execution history, i.e. it depends on the hardware state with which the block is entered. Without knowledge about the entering hardware state, a conservative analysis has to assume that any hardware state is possible, leading to exploration of unreachable states and overly pessimistic bounds. Therefore, in addition to upper bounds on basic block execution time, microarchitectural analysis computes for each basic block an overapproximation of the set of processor states that can reach the basic block.

Concrete Basic Block Execution.

Let Q be the set of concrete hardware states, $\mathcal{T}$ is the set of all possible traces and B is a set of basic blocks. We denote the *concrete* execution of a basic block $b \in B$ starting in state $a \in Q$ by a function $exec : B \times Q \to \mathcal{T}$ that computes a trace $t \in \mathcal{T}$. This trace corresponds to an interpretation of the instruction stream of b starting from a according to the concrete semantics. In a basic block graph, the execution of b's successor starts with the last state of t and b's execution time is given by the length of the trace.

Abstract Basic Block Execution.

Let $\widehat{Q}$ be the set of abstract states. Each abstract state represents a non-empty set of concrete states. We use the power set of $\widehat{Q}$ as the domain for microarchitectural analysis. We denote the *abstract* execution of a basic block b starting from a set of abstract states $\widehat{A} \subseteq \widehat{Q}$ by a function $\widehat{exec} : B \times 2^{\widehat{Q}} \to \widehat{\mathcal{T}}$ computing an abstract trace $\hat{t} \in \widehat{\mathcal{T}}$. An abstract trace is a sequence of sets of abstract states obtained by abstract interpretation of the instruction stream of basic block b starting from $\widehat{A}$. The length of trace $\hat{t}$ is an upper bound on the execution time of basic block b for any concrete start state corresponding to an abstract start state in the set $\widehat{A}$.
The analysis transfer function for updating domain elements at basic blocks is a function $\widehat{update} : B \times 2^{\widehat{Q}} \to 2^{\widehat{Q}}$. The function takes an incoming set of abstract states, constructs the abstract trace and returns its last element. Every possible successor block of b is then analyzed with a feasible subset of the obtained states.

Fixed Point Iteration.

At control flow joins, we combine sets of abstract states by set union. Since the power set of abstract pipeline states is a complete lattice and all employed update functions are monotone, the analysis finds a solution to the data flow problem by computing the least fixed point solution. If a basic block b is handled more than once during fixed point iteration, the resulting worst-case execution time for every edge leaving b is the maximum execution time bound of all abstract interpretations. The method yields safe results, i.e. overapproximations of the behavior of the real hardware, if the abstraction is sound w.r.t. the theory of abstract interpretation [11]. It has been shown that sound abstractions of the timing behavior can be constructed even for complex processors [35].

State Explosion.

Abstract states sometimes lack information about the precise state of some processor components, e.g. contents of buffers. Furthermore, value analysis information used for the classification of memory accesses can be imprecise, i.e. rather than one single address, the analysis may provide a range of possible addresses. In such cases, transitions in the abstract model depending on imprecise information become non-deterministic, i.e. an abstract state can have more than one successor. Unfortunately, we are not aware of an automatic way to decide locally which of these successors leads to the highest execution time [23, 30, 2]. Therefore pipeline analysis has to consider all possible successor states. We then say that the analysis performs so-called state splits.
Current tools for static WCET analysis employ an explicit

encoding of abstract pipeline states, i.e. states are stored individually in a list or a vector. Memory consumption and computation time for updating the set of abstract states at each program point grow linearly in the number of states. Since the number of states usually grows exponentially in the number of state splits, the analysis can quickly become infeasible [35].

4. SYMBOLIC REPRESENTATION

State explosion is a well-known problem in the area of model checking. Symbolic representations, usually based on BDDs, have significantly improved the situation because they admit both an implicit encoding of the transition system and of analysis information, like sets of states in state traversal. This has enabled the analysis of hardware designs with large state spaces [8]. We leverage a symbolic representation based on BDDs for state traversal of abstract pipeline models in WCET analysis, in order to avoid the explicit enumeration of abstract pipeline states and to improve scalability.

Fundamentals.

Let $\mathbf{f}, \mathbf{g} : \mathbb{B}^n \to \mathbb{B}$ be Boolean functions. We write $\mathbf{f} \cdot \mathbf{g}$ for conjunction, $\mathbf{f} + \mathbf{g}$ for disjunction, $\mathbf{f} \Rightarrow \mathbf{g}$ for logical implication, $\overline{\mathbf{f}}$ for negation and $(\exists x)[\mathbf{f}(x)]$ for existential quantification.

A finite state machine (FSM) with n state bits, consists of a set of states $Q \subseteq \mathbb{B}^n$, a set of initial states $S \subseteq Q$ and a transition relation $T \subseteq Q \times Q$. Each set of states $A \subseteq Q$, as well as the transition relation T, can be associated with a Boolean function $\mathbf{A} : 2^Q \to \mathbb{B}$ where $\mathbf{A}(x) = 1 \Leftrightarrow x \in A$ and $\mathbf{T} : 2^{Q \times Q} \to \mathbb{B}$ where $\mathbf{T}(x, y) = 1 \Leftrightarrow (x, y) \in T$.

Given a set of FSM states $A \subseteq Q$, we want to compute its set of successors with respect to the FSM transition relation, i.e. we want to compute the *image* of A under T denoted by the function $\mathbf{Img} : 2^{Q \times Q} \times 2^Q \to 2^Q$ where $\mathbf{Img}(\mathbf{T}, \mathbf{A})(y) = (\exists x)[\mathbf{T}(x, y) \cdot \mathbf{A}(x)]$. Image computation is the core operation of symbolic model checking algorithms and its efficient implementation has been the topic of many research papers, e.g. [27].

Symbolic model checking implements FSMs, sets of states, and image computation using BDDs and BDD operations. BDDs provide a compact, canonical representation of many Boolean functions of practical relevance. Fig. 2 shows an example of a BDD for a function with three variables. The BDD is evaluated by traversing the graph from the first variable node x_1 to one of the terminal nodes 1 or 0. Each variable node has two outgoing edges: the solid edge indicates that the variable has value 1, the dashed edge corresponds to the value 0. The terminal nodes represent the evaluation result. The path $x_1, x_2, x_3, 1$ in the example corresponds to the assignment $x_1 = 1$, $x_2 = 0$, $x_3 = 0$ for which the function evaluates to 1.

The complexity of BDD operations depends on the size, i.e., number of nodes, of the involved BDDs, which depends not only on the represented function, but also on the chosen variable ordering. Constructing a minimal BDD for a given function is an **NP**-hard problem.

4.1 Abstract Models

Abstract pipeline models can be specified as hardware designs in terms of latches and interconnecting wires. A design with n latches can be characterized by an FSM with state space $Q = \mathbb{B}^n$. Translating a hardware specification into its corresponding FSM is a well-known problem in hardware verification and solutions are readily available [9]. The *transition relation* of the FSM T is usually constructed by conjunction of the transition relations of the individual latches of the design [27]. For an FSM with n latches, we have

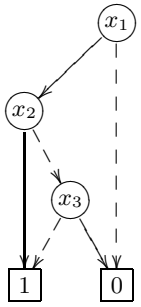

Figure 2: BDD for $f(x_1, x_2, x_3) = x_1 \cdot x_2 + x_1 \cdot \overline{x_2} \cdot \overline{x_3}$.

$$\mathbf{T}(x, y) = \prod_{j=0}^{n} \mathbf{T}_j(x, y)$$

Let $T_{\mathcal{M}}$ be the transition relation for an abstract pipeline model $\mathcal{M}$ with state space $\widehat{Q}$. We require an abstract model that is independent of any specific program. Such a model can be constructed by identifying variables, either single latches or groups of latches, for which some updates require information associated with program points (instructions). An example of such a variable is the current fetch address. On linear control flow, the fetch address is incremented by the pipeline model. On control flow redirections, e.g. by execution of branch instructions, program information is required to set the new fetch address to the branch target. In such states, the update of the fetch address is not constrained by $T_{\mathcal{M}}$. It is updated non-deterministically.

In general, in $T_{\mathcal{M}}$ we impose no restriction on the update of a variable v in states where the update of v requires program information. Thus, the following fixed point computation yields *all* abstract pipeline states reachable from an arbitrarily chosen set of start states $\widehat{\mathbf{Init}}(x)$:

$$\begin{aligned} \widehat{\mathbf{A}}_0(x) &= \widehat{\mathbf{Init}}(x) \\ \widehat{\mathbf{A}}_{k+1}(y) &= \widehat{\mathbf{A}}_k(x) + \mathbf{Img}(\mathbf{T}_{\mathcal{M}}, \widehat{\mathbf{A}}) \end{aligned} \tag{1}$$

4.2 Program Representation

Let V be the set of variables of model $\mathcal{M}$. $\bar{V} \subseteq V$ is the subset of variables whose updates require program information and are thus not completely determined by $T_{\mathcal{M}}$. We regard a program $\mathcal{L}$ for pipeline analysis as a set of *attributed instructions* uniquely identified by their physical addresses. The effect of executing instruction $l \in \mathcal{L}$ in a state of $\mathcal{M}$ updating a variable $\bar{v} \in \bar{V}$ can be expressed as a relation $T_{l,\bar{v}}$ between the states of $\mathcal{M}$. Intuitively, $T_{l,\bar{v}}$ is a *restriction* for $T_{\mathcal{M}}$. The relation $T_{l,\bar{v}}$ can be computed by a function $\delta_{\bar{v}} : \mathcal{L} \times 2^V \to \widehat{Q} \times \widehat{Q}$ which takes an instruction and a set of variables $D \subseteq V$ on which the update of $\bar{v}$ depends: $\delta_{\bar{v}}(l, D) = T_{l,\bar{v}}$. Each relation $T_{l,\bar{v}}$ constrains *only* the next-state value of $\bar{v}$. A relation for the whole program $\mathcal{L}$ is constructed as the conjunction of the relations for the in-

structions:

$$\mathbf{T}_{\mathcal{L}}(x,y) = \prod_{l \in \mathcal{L}} \prod_{\bar{v} \in \bar{V}} \mathbf{T}_{l,\bar{v}}(x,y)$$

Reachability analysis (as in Eq. 1) using the transition relation $\mathbf{T}^{\mathcal{L}}_{\mathcal{M}} = \mathbf{T}_{\mathcal{M}} \cdot \mathbf{T}_{\mathcal{L}}$ yields the set of all abstract pipeline states of $\mathcal{M}$ that are reachable by execution of $\mathcal{L}$.

Example.

Modeling the Infineon TriCore requires information about instruction fetches, i.e. the number and type of fetched instructions. On the rising edge of the clock signal `clk` this information is read into a variable `data` whenever the signal `valid` indicates that data is arriving on the bus and the fetch address `addr` holds a legal program address. In such states, `data` is updated non-deterministically in $T_{\mathcal{M}}$. Let `data`$'$ denote the next-state instance of the variable `data`. The instruction attribute $l.fetchData$ contains information about the instruction fetch at program point l. We define $T_{l,data}$ such that if the clock is rising, the data is valid, and the l is fetched, the next value of *data* is $l.fetchData$, i.e. $T_{l,data} = (\overline{clk} \cdot valid \cdot (addr = l)) \Rightarrow (data' = l.fetchData)$.

Other Required Program Information.

Beside control flow and instruction type information, updates of variables in the set $\bar{V}$ may also require information about data dependencies (for detecting pipeline stalls) and address ranges of memory accesses (for determining memory access latencies). The latter can be obtained from instruction attributes computed by the preceding value analysis. Data dependencies can also be precomputed and stored as instruction attributes.

4.3 Symbolic WCET Analysis

We present a microarchitectural WCET analysis which uses only symbolic computations on BDDs. It is based on the symbolic representation of abstract pipeline models and programs. Note that the symbolic implementation is equivalent to the explicit state approach in all but the representation of states. Due to our symbolic representation, explicit enumeration of states of the abstract pipeline model is completely avoided. This improves the performance of microarchitectural WCET analysis while preserving the soundness and precision of the analysis.

Abstract Basic Block Execution.

Let b be a basic block and $last(b)$ the last instruction in b. For a set of abstract pipeline states, $pred(x) : \mathbb{B}^n \to \mathbb{B}^n$ computes all predecessor states. R_b is the set of all states where $last(b)$ has just left the pipeline model, which can be characterized by the function $\mathbf{R}_b : \mathbb{B}^n \to \mathbb{B}$ given by $\mathbf{R}_b(x) = 1 \Leftrightarrow last(b)$ *left the pipeline in* $pred(x)$. The empty set of pipeline states is represented by $\mathbf{Empty} : \mathbb{B}^n \to \{0\}$.
Let $\widehat{A}_{in}$ be the set of incoming states at block b. The set of outgoing states $\widehat{A}_{out}$, i.e. the last element in the abstract trace of b starting with $\widehat{A}_{in}$, can be computed by Alg. 1 as $\widehat{update}(\mathbf{T}^{\mathcal{L}}_{\mathcal{M}}, \widehat{\mathbf{A}}_{in}, \mathbf{R}_b)$. This is an implementation of the update function for microarchitectural analysis introduced in Sec. 3. Its first argument (the basic block) is implicitly contained in $\mathbf{T}^{\mathcal{L}}_{\mathcal{M}}$ and $\mathbf{R}_b$.
Note that the algorithm implicitly constructs the abstract trace $\hat{t} = \widehat{exec}(b, \widehat{A}_{in})$. Line 3 of the algorithm computes the successor states in the next cycle. Line 4 adds the retired states to $\widehat{A}_{out}$ and line 5 removes them from $\widehat{A}$. The number of execution cycles of b equals the number of loop iterations in $\widehat{update}(\mathbf{T}^{\mathcal{L}}_{\mathcal{M}}, \widehat{\mathbf{A}}_{in}, \mathbf{R}_b)$.

1: $\widehat{\mathbf{A}}_{out} = \mathbf{Empty}$
2: **while** $\widehat{\mathbf{A}} \neq \mathbf{Empty}$ **do**
3: $\quad \widehat{\mathbf{A}} = \mathbf{Img}(\mathbf{T}, \widehat{\mathbf{A}})$
4: $\quad \widehat{\mathbf{A}}_{out} = \widehat{\mathbf{A}}_{out} + (\widehat{\mathbf{A}} \cdot \mathbf{U})$
5: $\quad \widehat{\mathbf{A}} = \widehat{\mathbf{A}} \cdot \overline{(\widehat{\mathbf{A}} \cdot \mathbf{U})}$
6: **end while**

Algorithm 1: $\widehat{update}(\mathbf{T}, \widehat{\mathbf{A}}, \mathbf{U})$

1: $\widehat{\mathbf{A}}_{out} = \widehat{update}(\mathbf{T}, \widehat{\mathbf{A}}, \mathbf{D}_b)$
2: **for all** e leaving b **do**
3: $\quad \widehat{\mathbf{A}}_{out,e} = \widehat{update}(\mathbf{T}, \mathbf{L}_{b,e} \cdot \widehat{\mathbf{A}}_{out}, \mathbf{R}_b)$
4: **end for**

Algorithm 2: DoBlock$(\mathbf{T}, \widehat{\mathbf{A}}, b)$

Control Flow Sensitive Timing Effects.

A basic block b can have more than one successor in the basic block graph, e.g. if b contains a conditional branch. The previous $\widehat{update}$ algorithm only computes the effect of a single control flow edge starting in a particular basic block. We now give an algorithm that uses the $\widehat{update}$ algorithm to deal with the general case of blocks with multiple outgoing edges. For each outgoing edge of the block, it computes the set of outgoing pipeline states and an execution time bound for the respective edge.
Let us assume that if b contains a branch instruction, then this instruction is always $last(b)$. In the case of a conditional branch, we allow for two successor states in the pipeline model (branch taken, not taken). If this state split occurs while decoding $last(b)$, we define the function $\mathbf{D}_b : \mathbb{B}^n \to \mathbb{B}$; $\mathbf{D}_b(x) = 1 \Leftrightarrow$ *last(b) has been decoded in pred(x).* If we run $\widehat{update}(\widehat{\mathbf{A}}_{in}, \mathbf{D}_b)$, the resulting set $\widehat{A}_{out}$ contains all states that have decoded $last(b)$ in the last cycle. For an outgoing edge e of block b, we characterize the set containing all states that can leave b over e by the function $\mathbf{L}_{b,e} : \mathbb{B}^n \to \mathbb{B}$; $\mathbf{L}_{b,e}(x) = 1 \Leftrightarrow$ *x may leave b via e.*
E.g. if $last(b)$ is a conditional branch and e is a true edge, then the conditional branch must have been taken on any state x leaving b via e. We can compute the outgoing set $\widehat{A}_{out,e}$ of states for every edge leaving b using Alg. 2. Let $|a|$ denote the number of loop iterations of algorithm a. The number of execution cycles for block b on the path via edge e is then given by

$$|\, \widehat{update}(\mathbf{T}^{\mathcal{L}}_{\mathcal{M}}\,,\, \widehat{\mathbf{A}}\,,\, \mathbf{D}_b)\,| \;+\; |\, \widehat{update}(\mathbf{T}^{\mathcal{L}}_{\mathcal{M}}\,,\, \mathbf{L}_{b,e} \cdot \widehat{\mathbf{A}}_{out}\,,\, \mathbf{R}_b)\,|$$

Fixed Point Iteration.

Fixed point iteration on the basic block graph should also operate directly on the symbolic representation. The required operations are set union and checking for equality of sets. Union of two sets is implemented by disjunction of BDDs. Equality checks are constant-time operations due to properties of BDDs [7].

5. OPTIMIZATIONS

A straightforward implementation of the analysis presented in the preceding section, does not scale to realistic microprocessors and programs. To scale, we not only leverage standard techniques from model checking. We also use knowledge about the processor to keep the processor model small and heavily exploit program structure in the symbolic representation of both program information and buffer contents in the processor. All of the presented optimizations improve the analysis performance without affecting the precision of the computed WCET bounds.

Building the transition relation for an FSM as a monolithic BDD by taking the conjunction of the relations for all latches is usually infeasible but for the smallest models. Therefore we apply conjunctive partitioning both to the FSM and the program relation $\mathbf{T}_{\mathcal{L}}$ which is also a conjunction of relations for the instructions. As this is a standard technique, we refer to [27] for a description and instead focus on an optimization specific to our approach: *address compression*.

Abstract pipeline models store many (program) addresses. A direct encoding would be to bit-blast the addresses, i.e. a 32-bit address takes 32 state bits. This would be inefficient, since BDD size, and therefore performance, is very sensitive with respect to the number of state bits. However, a program typically uses only a small fraction of the address space. Exploiting information from the basic block graph and from value analysis, one can *compactly enumerate* all addresses used in the program and then encode these addresses using a number of state bits logarithmic in the size of the set of used addresses.

5.1 Processor-Specific Optimizations

We apply several processor-specific optimizations. These follow the general pattern of:

1. reducing representation size of components by omitting information which is not timing-relevant.

2. statically precomputing information.

For illustration, we give two specific examples of such optimizations which are used in our TriCore implementation described in Sec. 6.

Compact Buffer Representation.

The prefetch buffer of the TriCore holds up to 8 instructions. Updating buffers and dispatching instructions into the correct pipelines requires type and size information for each instruction. Since the TriCore has two different instruction sizes and two major pipelines, this information can be represented by 2 bits per instruction. We thus represent the timing-relevant buffer contents by 16 bits compared to 16 bytes in the actual processor.

Precomputing Stall Conditions.

TriCore features a set of rules that define pipeline stalls in case of unresolved data dependencies. We precompute such dependencies by a data flow analysis and store the results as instruction attributes. The model then requires only one single bit per pipeline to encode pipeline stalls due to unfulfilled data dependencies. Updates of the stall bits are encoded in $\mathbf{T}_{\mathcal{L}}$ depending on the positions of the instructions in the pipelines.

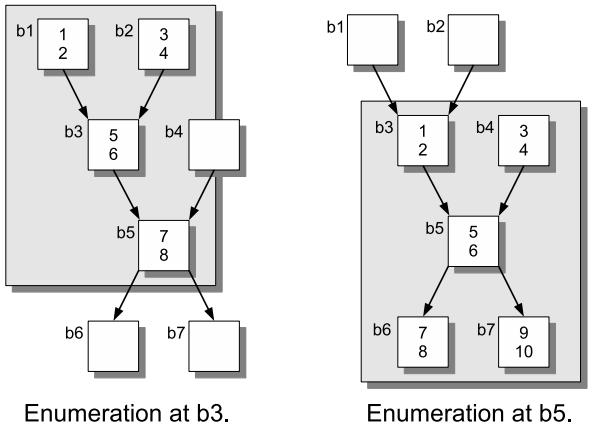

Figure 3: Illustration of Program Decomposition.

5.2 Program Decomposition

The aforementioned optimizations make the construction of the symbolic representations feasible. However, the required number of state bits heavily depends on the size of the program because all instructions must be uniquely identified in the pipeline model. Additionally, context-sensitive analysis, which is indispensable to obtain sufficient analysis precision, increases the number of individual instructions even further by virtual inlining and unrolling. Since BDD performance depends on the number of state bits, the performance of *every single* analysis operation depends on the size of the analyzed program and on the degree of context-sensitivity. We have developed an optimization that removes this undesired dependence. It is based on two observations:

1. There is an upper bound on the number of instructions that a pipeline (or abstract pipeline model) can process concurrently due to parallel execution, prefetching and speculation.

2. Pipelines perform out-of-order execution and yet guarantee in-order completion, i.e. even if some instruction l_2 can be executed before some other instruction l_1, it will not leave the pipeline before l_1.

Let V_l be the set of abstract model variables that reference instructions. It exists a partial ordering on V_l, ordering the variables with respect to their distance from the pipeline entry. E.g. variables in the decode stage are further away from the pipeline entry than variables in the fetch unit. We call the longest ascending chain of elements in this lattice an *overlap bound*.

We define two functions $pre, post : \mathcal{L} \times \mathcal{L} \to \mathbb{N}$ where $pre(i, j)$ returns k if j is the k-th predecessor of i and 0 otherwise. Further, $post(i, j)$ returns k if j is the k-th successor of i and 0 otherwise. The set of all predecessor or successor instructions for a given distance is obtained by $preds, succs : \mathcal{L} \times \mathbb{N} \to 2^{\mathcal{L}}$ where $preds(i, k) = \{j : 0 < pre(i, j) \leq k\}$ and $succs(i, k) = \{j : 0 < post(i, j) \leq k\}$. Based on a given overlap bound $\bar{c}$, a set of *relevant instructions* at a basic block b can be defined as $R(b) = b \cup preds(first(b), \bar{c}) \cup succs(last(b), \bar{c})$ where $first(b)$ and $last(b)$ are the first and the last instruction of b. Fig. 3 shows an example for two blocks $b3$ and $b5$, assuming that $\bar{c} = 2$. The numbers in

$\lceil \widehat{\mathbf{A}}_{out} \rceil_b = \widehat{update}(\lceil \mathbf{T} \rceil_b, \lceil \widehat{\mathbf{A}} \rceil_b, \lceil \mathbf{D}_b \rceil_b)$
for all e leaving b **do**
$\lceil \widehat{\mathbf{A}}_{out,e} \rceil_{tgt(e)} = \mathbf{Img}(\mathbf{T}_{(b,tgt(e))}, \widehat{update}(\lceil \mathbf{T} \rceil_b, \lceil \mathbf{L}_{b,e} \rceil_b \cdot \lceil \widehat{\mathbf{A}}_{out} \rceil_b, \lceil \mathbf{R}_b \rceil_b))$
end for

Algorithm 3: $\mathsf{DoBlock}^+(\lceil \mathbf{T} \rceil_b, \lceil \widehat{\mathbf{A}} \rceil_b, b)$

the blocks correspond to the enumeration of the relevant instructions at $b3$ and $b5$ respectively.
In general, we require fewer state bits for the unambiguous enumeration of $R(b)$ than for enumerating all instructions in $\mathcal{L}$. Furthermore, $|R(b)|$ is independent of $|\mathcal{L}|$. We denote this compaction of the symbolic representation by $\lceil \, \rceil_b$.

Translation Between Basic Blocks.

The encoding of instructions is no longer globally the same for all blocks and must therefore be translated before propagating states along control flow edges. We observe that the number of overlapping instructions between two adjacent blocks is bounded by the processor-specific constant $\bar{c}$. Let block s be a direct successor of block p. A symbolic relation $\mathbf{T}_{(p,s)}$ between different representations $\lceil \widehat{\mathbf{A}} \rceil_s$ and $\lceil \widehat{\mathbf{A}} \rceil_p$ of the same set of abstract pipeline states $\widehat{A}$ can be constructed by tabulating both instruction enumerations. For the relevant overlapping instruction numbers of Fig. 3 this yields the relation

$$\{(5,1),(6,2),(7,5),(8,6)\}$$

between the enumerations at $b3$ and $b5$.
Translating sets of abstract pipeline states between s and p can be implemented symbolically using image computation. We include this translation into Alg. 3 ($\mathsf{DoBlock}^+$), an improved version of Alg. 2 ($\mathsf{DoBlock}$). Abstract basic block execution for a compatibly encoded set of states $\lceil \widehat{\mathbf{A}} \rceil_b$ can now be performed efficiently by

$$\mathsf{DoBlock}^+(\lceil \mathbf{T}_{R(b)} \rceil_b \cdot \lceil \mathbf{T}_{\mathcal{M}} \rceil_b, \lceil \widehat{\mathbf{A}} \rceil_b, b)$$

Note that for a block b with several predecessors, all incoming states are translated into the same range $\lceil \, \rceil_b$. Therefore, the instruction numbering is consistent over all incoming edges and incoming states can be safely combined by computing the disjunction of the Boolean functions before proceeding with the state traversal.
The additional cost for translating between different basic blocks is amortized by the savings achieved by reducing BDD size during state traversal. Moreover, $\lceil \mathbf{T} \rceil_b$ conjoins fewer relations than $\mathbf{T}_{\mathcal{L}}$ which further improves the performance of image computations.

6. EXPERIMENTAL RESULTS

We have implemented a symbolic pipeline analysis framework that is integrated into the `aiT` tool chain and uses the code base of the model checker VIS [6]. All `aiT` analysis phases, except microarchitectural analysis, have been left unchanged. The pipeline model is specified in Verilog and compiled into a symbolic transition relation by VIS. A setup phase enumerates the instruction addresses in all analysis contexts and initializes tables for mapping between both representations. Based on this mapping, a *relation generator* builds the program transition relation. The generator is implemented as a processor-specific plugin that has to be specified together with the model's Verilog description. The plugin also comprises generators for the retirement functions $\mathbf{R}$ and $\mathbf{D}$ required by Alg. 2. The data flow analysis framework [24] of `aiT` has been instantiated to perform fixed point iteration on the symbolic representation. Thus, the integration is processor-independent, i.e. analyzing a new target requires only a Verilog model specification and the plugin for generating the program relation and retirement functions. To enable debugging and result checking, the framework provides a graphical representation of the least fixed point of abstract pipeline states.

6.1 Pipeline Modeling

The Infineon TriCore is used in hard real-time systems in the automotive industry, e.g. for engine control. It features two major pipelines: the Integer pipeline (I) handles data arithmetic and conditional jumps. The Load/Store pipeline (L/S) handles loads/stores, address arithmetic, unconditional jumps and calls. Both pipelines have decode, execute and writeback stages. The shared fetch unit reduces instruction fetch latencies using a 16 byte prefetch buffer holding up to 8 instructions which are either 2 or 4 bytes wide. A minor pipeline for handling zero-overhead loop instructions shares its decode stage with the L/S pipeline. TriCore can issue one instruction per cycle into each of the major pipelines. For improved performance, the architecture also features static branch prediction.
A commercial TriCore model is available in `aiT`. The model includes all relevant peripheral hardware such as buses, flash modules, buffers and caches. Its representation requires 500 bytes [1] per abstract pipeline state plus 196 bytes for the state of the peripheral components (not counting the dynamically sized abstract cache). Analysis of the TriCore model exhibits two sources of state splits:

1. imprecise information about data memory accesses.
2. imprecise information about the state of peripheral components.

A single unclassified memory access in this model causes up to 64 state splits. A series of unclassified memory accesses may quickly multiply this number. Imprecise information about the state of caches, flash modules and buffers may cause further state splits.
Our symbolic TriCore model has been modeled after the existing explicit state model. It conservatively approximates the timing of the full pipeline core. This has been verified by comparison with hardware traces of programs running from the on-chip scratch pad memory and with analysis traces of the explicit state analysis. Using the presented compression techniques, we arrive at a very compact symbolic representation. Its size ranges from 163 to 333 state bits, depending on the analyzed program.

6.2 Experimental Setup

Without caches and flash buffers, the pipeline model for TriCore is mostly deterministic since the processor core can be modeled accurately. To study the effects of state explosion,

[1] Explicit state pipeline models represent instructions and contexts as pairs of 32 bit pointers/integers. The space required to store these pairs dominates the size of abstract pipeline states.

	$\|\mathcal{L}\|$	sym bits	sym$^+$ bits
dhry 2.1	3361	12	7
edn	69462	17	8
task A	3214	12	8
task B	28606	15	9
task C	1035	11	8

Figure 4: Program sizes.

dhry 2.1	7	15	31	63	127
explicit	3	19	160	1286	9524
sym stat	815	1379	2609	5373	7748
sym dyn	329	580	774	2256	4090
sym$^+$ stat	455	796	1466	2869	4632
sym$^+$ dyn	256	522	926	1849	2788
edn	7	15	31	63	127
explicit	75	480	3666	40266	> 48 h
sym$^+$ dyn	2169	4033	7965	15245	19561

Figure 5: Analysis times in secs.

we introduce additional state splits by losing precision on the latencies of instruction fetches. This forces the analysis to explore the different possible prefetch buffer states and interleavings of instructions in the pipeline. The resulting behavior of the analysis corresponds closely to the analysis behavior in cases of unknown data memory accesses or cache misses. The commercial TriCore model has been modified accordingly, i.e. the behavior of both models is identical. This setup allows us to study state explosion of pipeline states in isolation, without dealing with the additional complexity of cache and bus models. Further, it gives us some control over the amount of state splits performed by the analysis. We use it to investigate how both implementations scale up in cases of state explosion.

Benchmark Programs.
We compare the performance of the explicit- and symbolic-state implementations by analyzing the following programs:

dhry 2.1 is the Dhrystone [37] integer CPU performance benchmark.

edn comprises DSP algorithms [12] like filters, matrix multiplication and FFT.

EC is a closed source automotive software for engine control, one of the main application areas for TriCore. We analyze its 3 major tasks which we call A, B and C.

Fig. 4 lists size information for all of the described programs. The first column gives the number of instructions in all analysis contexts, i.e. after virtual inlining and loop unrolling. The edn benchmark is analyzed with full virtual inlining and unrolling, hence the large number of instructions. The second column lists the number of bits required for global enumeration of all instructions including padding addresses. The last column gives the same information for the optimized implementation of Sec. 5.1.

6.3 Performance Comparison

Fig. 5 and 6 show a comparison of the analysis runtimes for the different implementations running on an Intel Core

task A	7	15	31	63	127
explicit	7	56	432	3450	> 5 h
sym stat	2352	4706	9425	19296	> 5 h
sym dyn	1131	1879	2368	4502	> 5 h
sym$^+$ stat	1296	2480	4879	9748	16069
sym$^+$ dyn	675	1174	2851	3848	5905
task B	7	15	31	63	127
explicit	64	459	3336	26090	> 48 h
sym$^+$ dyn	9529	14947	28141	56122	101658
task C	7	15	31	63	127
explicit	1	9	72	549	4198
sym stat	230	383	685	1564	2034
sym dyn	146	179	277	374	692
sym$^+$ stat	328	515	1155	2367	3967
sym$^+$ dyn	227	506	649	1205	1860

Figure 6: Analysis times for EC in secs.

2 Duo at 2.66 GHz. The first row of each table shows the runtimes for the explicit-state implementation whereas the following rows list the results for its symbolic-state counterpart in the following configurations:

sym symbolic implementation of Sec. 4.3.

sym$^+$ optimized symbolic implementation of Sec. 5.1.

stat initial static variable order of VIS [6].

dyn dynamic variable reordering by converging window permutation [16, 18].

Each program is analyzed with an increasing number of possible instruction fetch latencies starting from 7 up to 127. Thus, the average number of concurrently analyzed states grows exponentially from left to right. At the same time, the summarized maximum latencies on every path through the program grow by a factor of 2. The reported analysis times are in seconds and include all setup costs. The analysis times of the explicit-state implementation reflect the exponential growth in the number of states whereas the runtimes of the symbolic-state implementation only grow linearly. The explicit-state implementation is significantly faster if the number of states is small. However, the symbolic-state implementation catches up if the average number of states exceeds a certain threshold which appears to be independent of the analyzed program. The best symbolic-state analysis typically reaches the same order of magnitude as the explicit-state analysis at 63 splits per fetch and outperforms it at 127 splits per fetch.
We would like to point out that the considered numbers of state splits are within the typical range for pipeline models of medium complexity. E.g. the commercial TriCore model performs at most 64 splits per unclassified memory access. For more complex architectures, such as the Motorola PowerPC family of processors, the maximum number of state splits per unclassified memory access can be as high as 1000. For such architectures, the presented approach can be expected to enable significant performance improvements in cases of state explosion.
A comparison of the different configurations of the symbolic-state implementation shows that the optimized implementation of Sec. 5.1 is clearly superior to the global enumeration

approach of Sec. 4.3 except for very small programs as EC task C. Moreover, for large programs as EC task B and edn, the global enumeration becomes practically infeasible. We therefore show only the results for explicit and sym$^+$ for EC task B and edn.
The results also show that the initial static BDD variable ordering chosen by VIS is suboptimal for our application. We get better results using dynamic reordering. Reordering was invoked between 5 and 30 times per analysis. In most cases, analyses using global instruction enumeration performed about twice the number of reordering steps compared to the optimized configuration sym$^+$. The selected reordering algorithm, using window permutation [16, 18], was chosen for its speed and consistently good results. However, VIS implements a large number of reordering algorithms that we did not explore exhaustively. It is likely that other advanced reordering algorithms such as sifting [29] will yield similar results.

7. CONCLUSION

We have presented a tight integration of a symbolic state-exploration engine into a static WCET analysis tool. The approach combines the advantages of static program analysis, in particular the possibility to decompose a very hard analysis problem (WCET computation) into several simpler subproblems (value, cache, pipeline, and path analysis), with the strengths of symbolic state traversal. The use of symbolic methods significantly improves the scalability of the pipeline analysis while maintaining soundness. Combining an abstract-interpretation-based static analysis with a symbolic state-exploration engine is not an easy task. We described some of the difficulties that had to be solved on the way. Let us summarize our major contributions:

1. We describe a *novel* and *efficient* integration of a symbolic state traversal engine into an abstract interpretation based static analysis framework. Model size is kept relatively small since arithmetic is handled by the value analysis. Further, no cycle counter is required within the symbolic representation.

2. The integration comprises a *generic* solution for the interaction between a symbolic domain and other static analyses. The interaction is based on an explicit program representation as a control flow graph and supports advanced context-sensitive analysis. This enables interprocedural analysis and the precise analysis of loops.

3. We present a solution to scale the symbolic analysis to *realistic* program sizes. State traversal costs only depend on the size of the pipeline model, not on the program size or the number of analysis contexts.

4. We give experimental *evidence* showing that symbolic state traversal for pipeline analysis is feasible in practice. Since BDD representations are sensitive to the size of the employed models, it is crucial that we are able to keep the size of realistic pipeline models within acceptable limits. Further, our experimental data indicate that the built-in automatic variable reordering heuristics of a state-of-the-art model checker work well for our application.

The reported results confirm that the approach is suitable to alleviate the state explosion problem and demonstrate that the approach scales to industrial programs thanks to several optimizations. The symbolic computation of single cycle updates is relatively expensive for small sets of FSM states. Explicit-state implementations are therefore better suited for analyses where state explosion does not occur. Future research should consider a hybrid approach that admits switching between explicit-state and symbolic-state representations. Another direction for future research is to consider more complex architectures, e.g. Motorola PowerPC 755 [35]. Finally, integration of cache analyses remains future work.

Acknowledgements.
We would like to thank the anonymous reviewers for their comments. We thank Daniel Kästner and Reinhard Wilhelm for proof-reading preliminary versions of this paper. The second author is supported by the Deutsche Forschungsgemeinschaft as part of the Transregional Collaborative Research Center SFB/TR 14 AVACS.

8. REFERENCES

[1] AbsInt. aiT WCET Analyzers. http://www.absint.com/ait, 2000.

[2] C. Berg. PLRU cache domino effects. In *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*. Schloss Dagstuhl, Germany, 2006.

[3] M. Berndl, O. Lhoták, F. Qian, L. J. Hendren, and N. Umanee. Points-to analysis using BDDs. In *PLDI*, pages 103–114, 2003.

[4] A. Betts, G. Bernat, R. Kirner, P. Puschner, and I. Wenzel. WCET Coverage for Pipelines. Technical report, 2006.

[5] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker Blast. *STTT*, 9(5-6):505–525, 2007.

[6] R. K. Brayton, G. D. Hachtel, A. L. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. A. Edwards, S. P. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa. VIS: A System for Verification and Synthesis. In *CAV*, pages 428–432, 1996.

[7] R. Bryant. Graph based algorithms for boolean function manipulation. In *IEEE Transactions on Computers*, 1986.

[8] J. Burch, E. Clarke, K. McMillan, D. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. IEEE Comp. Soc. Press, 1990.

[9] S.-T. Cheng. Compiling Verilog into Automata. Technical report, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, 1994.

[10] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ Cache Coherence Protocol. In *CHDL*, pages 15–30, 1993.

[11] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of*

Programming Languages, Los Angeles, California, 1977.

[12] EDN. DSP Benchmarks. In *EDN - Electronic Design, Strategy, News*, Sept. 1988.

[13] J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, 2002.

[14] C. Ferdinand. *Cache Behavior Prediction for Real-Time Systems*. PhD thesis, Saarland University, 1997.

[15] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and Precise WCET Determination for a Real-Life Processor. In *Proceedings of EMSOFT 2001, LNCS 2211*, 2001.

[16] M. Fujita, Y. Matsunaga, and T. Kakuda. On variable ordering of binary decision diagrams for the application of multi-level logic synthesis. In *Proceedings of the conference on European design automation*, 1991.

[17] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 91(7), 2003.

[18] N. Ishiura, H. Sawada, and S. Yajima. Minimization of binary decision diagrams based on exchanges of variables. In *ICCAD*, pages 472–475, 1991.

[19] R. Jhala and K. L. McMillan. Microarchitecture Verification by Compositional Model Checking. In *CAV*, pages 396–410, 2001.

[20] D. Kästner, R. Wilhelm, R. Heckmann, M. Schlickling, M. Pister, M. Jersak, K. Richter, and C. Ferdinand. Timing Validation of Automotive Software. In *3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA) 2008*, Communications in Computer and Information Science (CCIS). Springer, 2008.

[21] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *DAC*, pages 456–461, 1995.

[22] G. Logothetis and K. Schneider. Exact High Level WCET Analysis of Synchronous Programs by Symbolic State Space Exploration. In *DATE*, pages 10196–10203, 2003.

[23] T. Lundquist and P. Stenström. Timing Anomalies in Dynamically Scheduled Microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, 1999.

[24] F. Martin. PAG - An Efficient Program Analyzer Generator. *STTT*, 2(1), 1998.

[25] F. Martin, M. Alt, R. Wilhelm, and C. Ferdinand. Analysis of Loops. In K. Koskimies, editor, *Proceedings of the 7th International Conference on Compiler Construction*, *LNCS 1383*, pages 80–94, Berlin, 1998. Springer.

[26] A. Metzner. Why Model Checking Can Improve WCET Analysis. In *CAV*, 2004.

[27] R. Ranjan, A. Aziz, R. Brayton, B. Plessier, and C. Pixley. Efficient BDD Algorithms for FSM Synthesis and Verification, 1995.

[28] J. Reineke. *Caches in WCET Analysis.* PhD thesis, Saarland University, 2008.

[29] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *ICCAD '93: Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 42–47. IEEE Computer Society Press, 1993.

[30] J. Schneider. *Combined Schedulability and WCET Analysis for Real-Time Operating Systems*. PhD thesis, Saarland University, 2003.

[31] J. Souyris, E. Le Pavec, G. Himbert, V. JÃ©gu, G. Borios, and R. Heckmann. Computing the Worst Case Execution Time of an Avionics Program by Abstract Interpretation. In *Proceedings of the 5th Intl Workshop on (WCET) Analysis*, 2005.

[32] L. Tan. The Worst-case Execution Time Tool Challenge 2006. *International Journal on Software Tools for Technology Transfer (STTT)*, 11(2):133 – 152, 2009.

[33] H. Theiling. ILP-based Interprocedural Path Analysis. In *Proceedings of the Workshop on Embedded Software*, Grenoble, France, 2002.

[34] H. Theiling. *Control Flow Graphs for Real-Time System Analysis*. PhD thesis, Saarland University, 2003.

[35] S. Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, 2004.

[36] S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand. An Abstract Interpretation-Based Timing Validation of Hard Real-Time Avionics Software. In *Proceedings of the International Conference on Dependable Systems and Networks*. IEEE Computer Society, 2003.

[37] R. Weicker. Dhrystone benchmark: rationale for version 2 and measurement rules. *SIGPLAN Notices*, 23(8):49–62, 1988.

[38] R. Wilhelm. Why AI + ILP is good for WCET, but MC is not, nor ILP alone. In *In Verification, Model Checking and Abstract Interpretation (VMCAI), LNCS 2937*, 2004.

[39] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Determination of Worst-Case Execution Times—Overview of the Methods and Survey of Tools. 7(3), 2008. ACM Transactions on Embedded Computing Systems (TECS).