

Integrating Abstract Caches with Symbolic Pipeline Analysis

Stephan Wilhelm¹ and Christoph Cullmann¹

¹ AbsInt Angewandte Informatik GmbH, Science Park 1; D-66123 Saarbrücken, Germany

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 space exploration which involves the abstract model and the program. Partial state space exploration is not sound. Symbolic methods using binary decision diagrams (BDDs) allow for a full state space exploration of the pipeline, thereby maintaining soundness. Caches are too large to admit an efficient BDD representation. On the other hand, invariants of the cache state can be computed efficiently using abstract interpretation. How to integrate abstract caches with symbolic-state pipeline analysis is an open question [11]. We propose a semi-symbolic domain to solve this problem. Statistical data from industrial-level software and WCET tools indicate that this new domain will enable an efficient analysis.

Digital Object Identifier 10.4230/OASlcs.WCET.2010.36

1 Introduction

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 [7, 9]. As a consequence, tools for safe WCET prediction have to cover all feasible program paths, inputs, and hardware states.

Static WCET analysis only becomes computationally feasible in practice by using abstraction, which is applied to both the modeling of processor and program behavior [5]. 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 explore all possibilities. This can lead to state explosion making an explicit enumeration of states infeasible due to memory and computation time constraints [10].

In [13] we presented a symbolic approach for pipeline analysis that avoids the explicit enumeration of reachable pipeline states, and showed its effectiveness in alleviating the state explosion problem in WCET analysis. The implementation cooperates efficiently with a framework of static analyses based on abstract interpretation. A commonality of these analyses is the fact that they run prior to pipeline analysis. Hence, cooperation boils down to importing statically available analysis results. In contrast, the abstract interpretation of caches [6] cannot be separated from pipeline analysis. The cache state depends on the order of memory accesses and therefore on the state of the pipeline. The pipeline state in turn is influenced by the latency of instruction and data fetches which depends on the cache state. Explicit-state implementations of pipeline analysis establish a one-to-one relationship



© Stephan Wilhelm and Christoph Cullmann;

licensed under Creative Commons License NC-ND

10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010).

Editor: Björn Lisper; pp. 36–43



OpenAccess Series in Informatics

OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

between pipeline and cache states, i.e., they combine each abstract cache state with a single abstract pipeline state. The pipeline state triggers an update of its associated cache state whenever the processor accesses a cached memory area.

Symbolic-state implementations cannot afford a one-to-one relation between pipeline and cache states without losing the advantages of symbolic state space exploration. We present experimental evidence that a one-to-one relation between pipeline and cache states is not required in practically relevant scenarios. Furthermore, we describe a semi-symbolic domain that efficiently integrates abstract interpretation based cache analysis with symbolic pipeline analysis while preserving a high analysis precision.

2 The Problem

Notation. The sets of natural numbers and Boolean values are denoted by \mathbb{N} and \mathbb{B} , respectively. $\mathbb{I} \subset \mathbb{N} \times \mathbb{N}$ is the set of intervals such that $\forall (l, u) \in \mathbb{I} : l \leq u$. We write $f \cdot g$ for conjunction, $f + g$ for disjunction, and \bar{f} for negation of Boolean functions and variables. A vector of Boolean values is written as \vec{x} .

Pipeline analysis [10, 4] computes upper bounds on the execution time of basic blocks using an abstract pipeline model. The model accounts for timing-relevant processor components, such as pipelining, speculation, and peripheral hardware. To reduce complexity, ALUs and register files are handled by a dedicated value analysis [3]. *Symbolic* pipeline analysis [13] uses BDD representations [2] of abstract pipeline models and sets of abstract states. The involved BDDs are directed, acyclic graphs that represent Boolean functions of type $\mathbb{B}^n \rightarrow \mathbb{B}$. An example BDD for the Boolean function $x_0 \cdot \bar{x}_1 \cdot \bar{x}_2 \cdot \bar{x}_3 \cdot \bar{x}_4 \cdot \bar{x}_5 + \bar{x}_0 \cdot (x_1 + \bar{x}_1 \cdot x_2)$ is depicted in Fig. 2.

The idea behind the symbolic approach is, that an abstract pipeline model corresponds to a finite state machine (FSM) with n Boolean state variables. Assignments of the state variables define pipeline states, e.g., in terms of different positions of instructions (identified by their addresses) in the pipeline. An FSM 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 $\mathbf{A} : \mathbb{B}^n \rightarrow \mathbb{B}$ where $\mathbf{A}(\vec{x}) = 1 \Leftrightarrow \vec{x} \in A$ and $\mathbf{T} : \mathbb{B}^n \times \mathbb{B}^n \rightarrow \mathbb{B}$ where $\mathbf{T}(\vec{x}, \vec{y}) = 1 \Leftrightarrow (\vec{x}, \vec{y}) \in T$. We say that $\mathbf{A} : \mathbb{B}^n \rightarrow \mathbb{B}$ is the *characteristic function* of the set A . The pipeline model is given in terms of its symbolic transition relation by the BDD $\mathbf{T}_{\mathcal{M}}$. Static program information, such as branch targets and intervals of register contents, are encoded into a BDD program relation $\mathbf{T}_{\mathcal{L}}$ that restricts the possible transitions of $\mathbf{T}_{\mathcal{M}}$. A set of pipeline states is represented by a BDD \mathbf{A} . State traversal is implemented by repeated application of a symbolic image operator $\mathbf{Img} : (\mathbb{B}^n \times \mathbb{B}^n \rightarrow \mathbb{B}) \times (\mathbb{B}^n \rightarrow \mathbb{B}) \rightarrow (\mathbb{B}^n \rightarrow \mathbb{B})$ [8]. The set of successor states for the states in \mathbf{A} is computed by $\mathbf{Img}(\mathbf{T}_{\mathcal{M}} \cdot \mathbf{T}_{\mathcal{L}}, \mathbf{A})$.

Cache analysis [6] operates on abstract representations of cache states. The abstract representation allows to trade precision for efficiency. Soundness is maintained by losing information only on the safe side, i.e., the result over-approximates the concrete cache states but it never misses a reachable cache state. The interface of the cache analysis comprises functions to query and update abstract caches with intervals of memory addresses. It also features a join operator for joining two cache states into another cache state that over-approximates both. The join operation may lose precision. There are two possibilities for integrating a symbolic-state implementation of pipeline analysis with a cache representation:

1. Including the cache into the symbolic representation of pipeline states.

2. Associating an abstract cache representation with a symbolic representation of pipeline states.

Let us consider the first approach. Even small caches are too large to admit a straightforward BDD representation for symbolic state traversal. In [12] we proposed an alternative symbolic representation for caches. Compactness was achieved by losing the correlation between the abstract cache cells; the resulting BDD is no longer the characteristic function of a set of hardware states. Unfortunately, it seems that – despite its compactness – the proposed representation does not allow for an efficient state traversal. So far, attempts to design efficient image operators, i.e., operators that avoid an exhaustive enumeration of the encoded states, have not been successful.

The second possibility seems equivalent to the approach taken by explicit-state implementations. However, symbolic-state implementations cannot afford a one-to-one relation between pipeline and cache states without losing the advantages of symbolic state space exploration. The explicit handling of caches would require the same explicit enumeration of pipeline states that the symbolic representation is trying to avoid. The next section presents a domain that is based on this second possibility, but uses a more favorable relation between pipeline and cache states.

3 Proposed Domain

We propose a semi-symbolic domain that integrates an abstract cache representation with a symbolic representation of pipeline states. The explicit enumeration problem is avoided by maintaining an efficient relation between pipeline and cache states. The basic idea is that we combine a set of pipeline states (represented symbolically by a BDD) with a single abstract cache state. The product of the pipeline and cache domains is thus based on an *n-to-one* relation. This allows us to preserve the benefits of the symbolic representation by manipulating sets of pipeline states symbolically.

Let $\widehat{\mathcal{C}}$ and $\mathbb{B}^n \rightarrow \mathbb{B}$ denote the abstract cache domain and the symbolic pipeline domain, respectively. A partition of abstract hardware states is a tuple of type $(\mathbb{B}^n \rightarrow \mathbb{B}) \times \widehat{\mathcal{C}}$ and \mathcal{H} denotes the set of all partitions. The proposed domain \mathcal{D} is the power set of \mathcal{H} excluding the empty set.

3.1 Updating partitions of abstract hardware states

We show the update of a single partition $(\mathbf{A}, \widehat{a}) \in \mathcal{H}$, where $\mathbf{A} : \mathbb{B}^n \rightarrow \mathbb{B}$ is a BDD representing a set of pipeline states and $\widehat{a} \in \widehat{\mathcal{C}}$ is an abstract cache state. Let \mathcal{A}_C be the set of all addresses in cached memory that are accessed by the analyzed program. For the remainder of this paper we assume that *all* memory accesses address cached memory regions. The pipeline model then needs $m = \log_2(|\mathcal{A}_C|)$ state variables for addressing memory. We require that these variables appear first in the BDD representation. The addressed interval can be obtained by a function $acc : (\mathbb{B}^n \rightarrow \mathbb{B}) \rightarrow \mathbb{I}$ that inspects the first m BDD variables. Its implementation is discussed in Sec. 3.3.

The classification function $cl : \widehat{\mathcal{C}} \times \mathbb{I} \rightarrow \{(0, 1), (1, 0), (1, 1)\}$ of the abstract cache domain determines whether an access results in a cache hit (0, 1) or miss (1, 0). Note that the result of this query can also be *undecided* (1, 1) if precise information has been lost due to abstraction or if the interval comprises both, cache hits and misses. The result of $cl(\widehat{a}, acc(\mathbf{A}))$ can be encoded as a symbolic relation \mathbf{T}_C by a function $enc : \mathbb{B}^2 \rightarrow (\mathbb{B}^n \times \mathbb{B}^n \rightarrow \mathbb{B})$. The computed relation restricts the possible transitions of the model relation $\mathbf{T}_{\mathcal{M}}$. It only allows

$$\begin{aligned}
step(\mathbf{A}, \hat{a}) = & \\
& \text{let } I = acc(\mathbf{A}) \text{ in} \\
& \text{let } \mathbf{T}_C = enc(cl(\hat{a}, I)) \text{ in} \\
& (\mathbf{Img}(\mathbf{T}_M \cdot \mathbf{T}_L \cdot \mathbf{T}_C, \mathbf{A}), up(\hat{a}, I))
\end{aligned}$$

■ **Figure 1** Implementation of the update function $step : \mathcal{H} \rightarrow \mathcal{H}$.

for transitions that correspond to the result of the cache query. This is analogue to the construction of \mathbf{T}_L from statically available program information.

Let $up : \hat{\mathcal{C}} \times \mathbb{I} \rightarrow \hat{\mathcal{C}}$ denote the update function for abstract cache states. The update of a single partition (\mathbf{A}, \hat{a}) can then be computed by a function $step : \mathcal{H} \rightarrow \mathcal{H}$ as depicted in Fig. 1. The $step$ function first determines the interval I of memory addresses that is accessed by the pipeline states in \mathbf{A} . It then queries the cache domain to determine whether the access hits or misses the cache and – based on this information – constructs the BDD \mathbf{T}_C for restricting the reachable pipeline states. The constructed BDD is conjoined with the BDDs \mathbf{T}_M and \mathbf{T}_L to obtain the effective transition relation for the next update. By application of the image operator on the computed transition relation and the set of pipeline states \mathbf{A} , it computes the set of successor pipeline states. The next cache state is obtained by application of the cache domain update function on the current cache \hat{a} and the accessed interval I .

3.2 Balancing pipeline and cache states

In order to maintain a favorable n-to-one relation between pipeline and cache states, we introduce a balancing operation to be applied in each round of the state traversal. The balancing operation involves two steps: partition and join. The partition step is based on the decomposition of the BDD of pipeline states. Decomposition of a BDD $f : \mathbb{B}^n \rightarrow \mathbb{B}$ into its cofactors with respect to a variable x_n means computing subfunctions $g, h : \mathbb{B}^{n-1} \rightarrow \mathbb{B}$ such that $g(x_1, \dots, x_{n-1}) = f(x_1, \dots, x_{n-1}, 0)$ and $h(x_1, \dots, x_{n-1}) = f(x_1, \dots, x_{n-1}, 1)$. This decomposition, also known as Shannon expansion, is a very efficient operation on BDDs and the foundation of many basic BDD algorithms.

Let $(\mathbf{A}, \hat{a}) \in \mathcal{H}$ be a partition of abstract hardware states. The first m state variables in \mathbf{A} encode the accessed interval of memory addresses. We partition (\mathbf{A}, \hat{a}) by a function $part : \mathcal{H} \rightarrow \mathcal{D}$ that recursively decomposes \mathbf{A} into its cofactors with respect to the first m state variables. A new partition is created for each cofactor together with a copy of the cache state \hat{a} . The decomposition proceeds until all satisfying paths of the BDD pass through the variable $m + 1$. As a result, all pipeline states in a new partition $(\mathbf{A}_x, \hat{a}) \in part(\mathbf{A}, \hat{a})$ access the same interval of memory addresses. Note that $step(\mathbf{A}_x, \hat{a})$ yields a more precise successor cache state than $step(\mathbf{A}, \hat{a})$.

Excessive partitioning might lead us back to the explicit enumeration problem. In the worst case, each partition in a domain element $D \in \mathcal{D}$ encodes only a single pipeline state. We prevent this by applying a join operator to partitions of D . Let \sqcup denote the join operator for abstract caches [6]. The union of two sets of pipeline states is implemented by disjunction of their characteristic functions. Two partitions (\mathbf{A}, \hat{a}) and (\mathbf{B}, \hat{b}) are joined by a function $join : \mathcal{H} \times \mathcal{H} \rightarrow \mathcal{H}$:

$$join((\mathbf{A}, \hat{a}), (\mathbf{B}, \hat{b})) = (\mathbf{A} + \mathbf{B}, \hat{a} \sqcup \hat{b})$$

To minimize the loss of cache precision, we join only partitions whose pipeline states access the same interval of memory addresses. This restriction also prevents us from undoing the partitioning. The loss in cache precision could be limited further by joining only hardware states with similar caches. This however requires a similarity metric for abstract cache states. A simple but efficient similarity metric would be, to only join two cache states $\hat{a}, \hat{b} \in \hat{\mathcal{C}}$ if one of them already over-approximates the other, which is equivalent to

$$\hat{a} \sqcup \hat{b} = \hat{a} \quad \text{or} \quad \hat{a} \sqcup \hat{b} = \hat{b}$$

Besides balancing the relation between pipeline and cache states and optimizing the representation for an efficient implementation of the function $acc : \mathbb{B}^n \rightarrow \mathbb{I}$, the application of regular partition and join operators also ensures a canonical representation of hardware states; because of the balancing operations, a particular hardware state always ends up in exactly one partition of an element of \mathcal{D} . This property allows for an efficient equality check of data flow elements by pairwise invocation of the equality operators of the two underlying domains on the contained partitions. It is most efficient if the number of partitions is small.

3.3 State traversal and performance

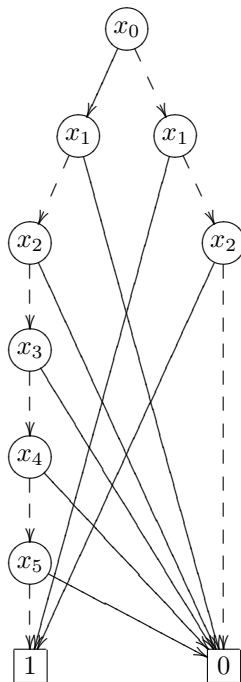
The state traversal for micro-architectural analysis on the domain \mathcal{D} is implemented by repeated application of the function $step : \mathcal{H} \rightarrow \mathcal{H}$ to all elements of a domain element $D \in \mathcal{D}$. Partition and join functions are applied in each round of the traversal for balancing pipeline and cache states before applying the $step$ function.

The proposed domain is most efficient if each cache state is associated with a large number of pipeline states. This allows for a small number of BDD operations which exploits the caching of intermediate results that is typical for BDD algorithms. Moreover, it significantly reduces the required number of cache updates since we perform a single cache update for all of the associated pipeline states. Note that a small number of partitions per domain element is also desirable.

A favorable relation between pipeline and cache states is maintained by the regular application of the join operator. The prior application of the partition operator minimizes the loss of cache precision and optimizes the BDD representation to allow for an efficient implementation of the function $acc : \mathbb{B}^n \rightarrow \mathbb{I}$. Its efficiency depends on the fact that

- the variables for addressing memory appear first in the BDD, and
- all encoded pipeline states access the same interval of addresses.

Hence, it suffices to enumerate the satisfying paths over the first m BDD variables. Let us consider the example depicted in Fig. 2. The example BDD shows only the first 6 state variables for accessing memory, i.e., we have $m = 6$. Note that in the full representation, the terminal node 1 would be replaced by a subgraph that represents the set of associated pipeline states. The BDD is evaluated by traversing the graph from the first variable node x_0 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. Nodes whose values do not influence the final result are omitted in the BDD (dont-care nodes). The terminal nodes represent the evaluation result. A path that ends at the terminal node 1 is called a satisfying path. It corresponds to one or several satisfying assignments of the variables. The satisfying paths over the example BDD of Fig. 2 are depicted in the first table of Fig. 3. To determine the interval that corresponds to a satisfying path, we set all dont-care nodes to 0 to obtain the lower bound (see table 3 in Fig. 3), and to 1 to obtain the upper bound (see table 2 in Fig. 3). Finally, we obtain the represented interval by taking the minimum and maximum of the intervals over all satisfying paths.



■ **Figure 2** BDD representation of the memory access interval $[8, 32]$. In the full representation, the terminal node 1 is replaced by a subgraph that represents the set of associated pipeline states.

x_0	x_1	x_2	x_3	x_4	x_5
1	0	0	0	0	0
0	1	-	-	-	-
0	0	1	-	-	-

1. satisfying paths

x_0	x_1	x_2	x_3	x_4	x_5	ub
1	0	0	0	0	0	32
0	1	1	1	1	1	31
0	0	1	1	1	1	15

2. upper bound

x_0	x_1	x_2	x_3	x_4	x_5	lb
1	0	0	0	0	0	32
0	1	0	0	0	0	16
0	0	1	0	0	0	8

3. lower bound

■ **Figure 3** Computing the lower and upper bounds of the intervals that correspond to the satisfying paths. The complete interval is then computed as $[\min\{32, 16, 8\}, \max\{32, 31, 15\}] = [8, 32]$.

The example shows that the interval can be computed from the BDD without enumerating all contained addresses. Note that the computational effort does not grow significantly if the interval shares a larger address prefix (using additional state variables x_6, x_7, \dots, x_m to address memory). The additional state variables either allow only for a single assignment, or most of them are dont-care nodes. The number of satisfying paths in the BDD will stay small.

4 Typical Cache Access Patterns

We experimented with 6 tasks of a commercial, safety-critical real-time software¹ to assess the correlation between memory accesses from different pipeline states. The tasks have been fully unrolled and annotated to avoid serious state explosion. The employed annotations specify ranges for register contents at selected program points to improve the precision of the value analysis and thereby reduce the reachable state space of the pipeline model. Note that full unrolling is not feasible for all software but required to obtain results with explicit-state implementations of very complex pipeline models. Otherwise, the analysis would not terminate in acceptable time because of state explosion. The following results have been obtained with the commercial, explicit-state pipeline model of the Motorola PowerPC

¹ Closed source and confidential.

755 [10, 1]. With full unrolling and annotations, all analyses terminate in less than 5 minutes running on an Intel i5 CPU at 2.67 GHz. We instrumented the pipeline model to print the following information for each access into a cached memory area:

1. Type of access, i.e., instruction or data.
2. Address and context of the currently analyzed basic block.
3. Cycle count since start of current basic block.
4. Accessed address or address range.

For each type of access, we collect all accesses with equal basic block address, analysis context, and cycle count. Symbolic pipeline analysis explores the model’s state space cycle-wise in breadth-first order. Hence, all accesses in one set are issued from pipeline states in the same exploration layer. We partition the sets depending on the accessed addresses to obtain the number of different memory accesses from the same layer. The following tables list the results of this experiment. For each task (numbered t_1, \dots, t_6) the first row gives the results for instruction cache accesses, whereas the second row reports the same information for data cache accesses.

■ **Table 1** Avg. number of partitions per cycle. ■ **Table 2** Max. number of partitions per cycle.

t_1	t_2	t_3	t_4	t_5	t_6	t_1	t_2	t_3	t_4	t_5	t_6
2.19	1.51	1.94	2.03	2.13	1.52	42	7	42	42	42	10
1.38	1.11	1.29	1.35	1.35	1.02	6	2	6	6	6	2

■ **Table 3** Avg. number of states per partition. ■ **Table 4** Max. number of states per partition.

t_1	t_2	t_3	t_4	t_5	t_6	t_1	t_2	t_3	t_4	t_5	t_6
17.24	35.35	25.49	25.61	35.63	19.96	4927	1311	8519	8190	8544	1091
10.82	28.07	20.76	25.94	25.61	7.87	1947	720	7140	7783	8115	268

According to Tab. 1, the average number of partitions is roughly 2. This number corresponds directly to the expected average number of partitions of an element of the proposed domain \mathcal{D} . Tab. 3 shows the average sharing, i.e., the number of pipeline states that can be encoded into a single BDD. The results indicate that the average relation between pipeline and cache states is roughly 18 : 1 (by dividing the average sharing of Tab. 3 by the average number of partitions of Tab. 1). The maximum number of partitions stays fairly small as shown by the results in Tab. 2. On the other hand, the maximum number of pipeline states per BDD can be quite large as shown by the results in Tab. 4.

All results indicate that the proposed domain operates on tuples with typical pipeline-cache relations between 1 : 1 and 8544 : 1, with an average of 18 : 1. These numbers hold under the assumption that the analysis maintains maximum cache precision. The proposed domain allows higher numbers of pipeline states per partition if caches are joined more aggressively. Larger numbers of pipeline states per partition can also be expected when the analysis encounters cases of imprecise information, e.g., about memory accesses.

5 Conclusion

We presented a new domain that integrates a symbolic exploration of abstract pipeline states with an abstract interpretation based domain for computing invariants of the cache state. Sets of pipeline states are stored in BDDs and manipulated symbolically using BDD operations. Abstract cache states are associated with sets of symbolically encoded pipeline

states. Partition and join steps balance the representation to preserve a high analysis precision while avoiding an explicit enumeration of pipeline and cache states. Statistical data indicates that it is possible to maintain a favorable relation between pipeline and cache states, which allows us to reap the benefits of symbolic state traversal for pipeline analysis.

Acknowledgements

We would like to thank the anonymous reviewers for their comments. We thank Daniel Kästner and Reinhold Heckmann for proof-reading this paper and Marc Schlickling for providing information about the Motorola PowerPC 755 pipeline model.

References

- 1 AbsInt. aiT WCET Analyzers. <http://www.absint.com/ait>, 2000.
- 2 R.E. Bryant. Graph based algorithms for boolean function manipulation. In *IEEE Transactions on Computers*, 1986.
- 3 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 Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Los Angeles, California, 1977.
- 4 Jakob Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, 2002.
- 5 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.
- 6 Christian Ferdinand. *Cache Behavior Prediction for Real-Time Systems*. PhD thesis, Saarland University, 1997.
- 7 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), 2003.
- 8 R. Ranjan, A. Aziz, R. Brayton, B. Plessier, and C. Pixley. Efficient BDD Algorithms for FSM Synthesis and Verification, 1995.
- 9 Jan Reineke. *Caches in WCET Analysis*. PhD thesis, Saarland University, 2008.
- 10 Stephan Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, 2004.
- 11 Reinhard Wilhelm, Sebastian Altmeyer, Claire Burguière, Daniel Grund, Jörg Herter, Jan Reineke, Björn Wachter, and Stephan Wilhelm. Static timing analysis for hard real-time systems. In *VMCAI*, pages 3–22. Springer Verlag, 2010.
- 12 Stephan Wilhelm and Björn Wachter. Towards symbolic state traversal for efficient WCET analysis of abstract pipeline and cache models. In *Proceedings of Seventh International Workshop on Worst-Case Execution Time Analysis*, July 2007.
- 13 Stephan Wilhelm and Björn Wachter. Symbolic state traversal for WCET analysis. In *International Conference on Embedded Software*, pages 137–146, October 2009.