Towards Symbolic State Traversal for Efficient WCET Analysis of Abstract Pipeline and Cache Models *

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

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

Abstract

Static program analysis is a proven approach for obtaining safe and tight upper bounds on the worst-case execution time (WCET) of program tasks. It requires an analysis on the microarchitectural level, most notably pipeline and cache analysis. In our approach, the integrated pipeline and cache analysis operates on sets of possible abstract hardware states. Due to the growth of CPU complexity and the existence of timing anomalies, the analysis must handle an increasing number of possible abstract states for each program point. Symbolic methods have been proposed as a way to reduce memory consumption and improve runtime in order to keep pace with the growing hardware complexity. This paper presents the advances made since the original proposal and discusses a compact representation of abstract caches for integration with symbolic pipeline analysis.

1. Introduction

Finding the worst-case execution time (WCET) for all tasks of a software is an important requirement in the design of hard real-time systems. A proven approach for obtaining tight upper bounds of the WCET, based on *abstract interpretation* (AI), has been presented in [9]. It employs several semantics-based static program analyses on the assembly level control flow graph (CFG) of the input program. First, the *value analysis* computes possible register contents for each program point in order to determine the address ranges for instructions accessing memory. Then, an integrated *pipeline and cache analysis*, operating on safe approximations of the possible pipeline and cache states, computes a WCET bound for each basic block of the CFG. Safe approximation means, that the analysis might only consider too many states, i. e. the WCET state is always included. The correctness of this approach has been proven [8] [15]. Finally, a *path analysis* computes the global worst-case path using the WCET bounds for basic blocks determined by the pipeline and cache analysis [14]. The AI approach has been used very successfully for various complex, real-life architectures [16] [13].

Unfortunately, CPUs using modern techniques for reducing the average execution time, such as caches, pipelined execution, branch prediction, speculative execution, and out-of-order execution, are often subject to *timing anomalies*. A timing anomaly is a local worst-case behavior, e. g. a cache miss, that does not contribute to the global worst-case [11]. As a consequence, abstract interpretation of the pipeline behavior must consider a large number of possible abstract pipeline states for each program point. This problem is also known as *state explosion*. In certain cases, the analysis can even become infeasible because of the increase in memory consumption and computation time [15].

Measurement-based approaches for computing the WCET avoid the high cost of microarchitectural modeling and are therefore not affected by the problem of state explosion. However, measurement-based approaches are often not suitable for safety critical applications since an underestimation of the WCET cannot be excluded [2]. Furthermore, using measurement-based methods for complex architectures can lead to a high overestimation of the WCET [7].

Symbolic methods have been proposed recently as a way to handle sets of abstract pipeline states efficiently, reducing memory consumption and improving runtime, in order to keep pace with the growing hardware complexity [17]. In the past, such methods have been used successfully for similar problems in model checking [5]. This paper presents an extension of the AI approach for pipeline analysis, using a symbolic representation of abstract pipeline states.

*This work has been funded in part by the ARTIST2 Network of Excellence (http://www.artist-embedded.org/) and by the Transregional Collaborative Research Center 14 AVACS (http://www.avacs.org/).

ECRTS 2007
7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis
http://drops.dagstuhl.de/opus/volltexte/2007/1190

Our Contribution. We propose an improved algorithm for symbolic state traversal of abstract pipeline models that overcomes limitations of the algorithm presented in [17]. Furthermore, we discuss a compact encoding of abstract caches that admits an integration of pipeline and cache analysis.

The paper is organized as follows. Section 2 briefly introduces the required terminology and section 3 gives an informal overview of the abstraction techniques used for deriving abstract pipeline models. Then, we review the algorithm for computing state transitions for sets of abstract pipeline states, using a small example (section 4). Using the same example, section 5 shows that the algorithm does not handle all cases correctly and proposes a solution for this problem. Section 6 discusses the performance of the approach and section 7 presents the current state of our implementation and gives first performance numbers. Finally, section 8 discusses the integration with a cache analysis.

2. Background

Given a finite state machine (FSM) with a set of states Q, a set of input values I, and a transition relation T, each set of FSM states $A \subseteq Q$ can be associated to its *characteristic function* $\mathbf{A} : Q \to \{0, 1\}; \mathbf{A}(x) = 1 \Leftrightarrow x \in A$. In the same way, the *transition relation* T can be associated to the function $\mathbf{T} : Q \times I \times Q \to \{0, 1\}; \mathbf{T}(x, i, y) = 1 \Leftrightarrow (x, i, y) \in T$. It is common practice to represent FSM state sets and the FSM transition relation by their characteristic functions encoded as *binary decision diagrams* (BDDs). BDDs are usually more compact than explicit representations and efficient implementations of useful operations such as negation, conjunction and existential quantification exist [4].

Given a set of FSM states $A \subseteq Q$, the *image* of A, $\text{Img}(A) \subseteq Q$, is the set of states that is reachable from A under T. Image computation is the core operation of symbolic model checking algorithms and can be efficiently implemented using BDDs [12].

Pipeline analysis is a static program analysis which performs a fixed point iteration on the domain of abstract pipeline states. The least fixed point (LFP) is the solution to the data flow problem containing all states that are reachable for a given program point including the WCET state. The result is a maximum number of cycles for each basic block.

Symbolic pipeline analysis is an implementation of pipeline analysis using BDDs for representing sets of abstract pipeline states (an abstract pipeline model is an FSM). In addition to the standard BDD operations provided by any BDD library, it uses the image computation engine from a model checker for efficiently computing the reachable abstract pipeline states.

3. Abstract Pipeline Models

Implementation of an abstract pipeline model requires detailed knowledge about the internal working and timing behavior of a CPU. This knowledge can be obtained from written documentation or hardware traces of bus signals or from the original Verilog or VHDL implementation (although the latter is often not available). Using any of this information, an abstract pipeline model can be derived by

1. Omission of timing-irrelevant implementation details.
2. Omission of data paths.
   Operations on the register file are already handled by the value analysis. Possible register values are therefore available during pipeline analysis without modeling register file and ALU.
3. Use of instruction addresses in all pipeline stages.

The last two points require a detailed discussion because they are crucial for understanding the difference between model checking and pipeline analysis. Model checking explores the state space of a model in order to prove or reject a statement in temporal logic. All information is contained in the model and state space exploration is guided by the transition relation and by the logic statement. In contrast, pipeline analysis explores the model's state space guided by the transition relation and the structure of a program and by additional information from other analyses. Thus, the model does *not* contain all information because we have delegated some of it, e. g. to the value analysis. In order to obtain this information during the analysis, we need to establish a relationship between abstract states and program points. This is achieved by the use of instruction addresses in all pipeline stages (see item 3 above). The same relationship can be used to annotate the analysis result, i. e. the WCET for each basic block.

Reading the delegated information during state transition is trivial for implementations operating on an explicit representation of abstract states, because the transition is computed individually for each abstract state [15]. On the other hand, symbolic pipeline analysis gains efficiency by using BDD operations on sets of abstract states. Explicitly extracting and recoding single states has to be avoided or minimized. An algorithm for computing state transitions on sets using image computation and encoding the delegated information using BDD operations has been presented in [17]. In the next section, we will illustrate the key ideas using a simplified abstract pipeline model.

4. An Illustrated Example

Figure 1 shows a simplified extract from our abstract pipeline model for a subset of the Infineon Tricore [18]. The

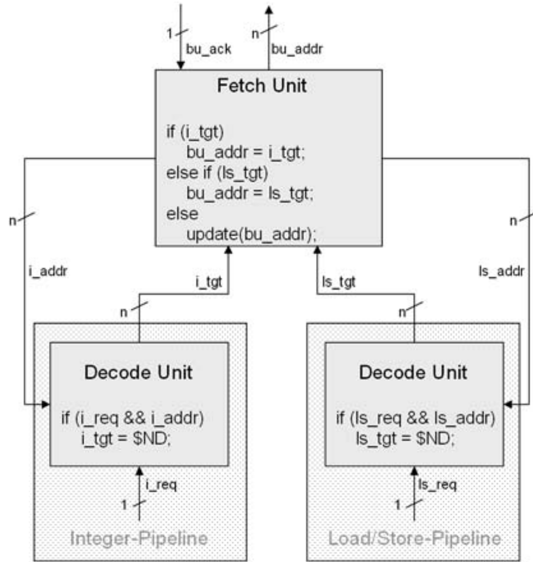

Figure 1: Extract from Tricore model.

architecture features two pipelines, called *Integer-Pipeline* (IP) and *Load/Store-Pipeline* (LSP), which share the same fetch unit. Each pipeline has its dedicated decode unit. [1] The state of each unit is held in a few variables which are updated in each cycle. The Verilog code below the unit's name shows the update for three selected variables. The current value of each variable is communicated to other units using `1`- or `n`-bit wide wires, where `n` is the number of bits used for encoding instruction addresses.

Let us examine what happens if we analyze this subset of our abstract model. The fetch unit sends instruction addresses to the bus unit via `bu_adr` which returns the signal `bu_ack` when the instruction data arrives. The fetch unit controls an 8 byte prefetch buffer (not shown in the example) and dispatches its contents to the decode units. The fetch unit sends only instruction addresses via `i_addr` and `ls_addr` instead of sending instruction data (remember that data paths have been removed). The decode units are responsible for detecting structural pipeline hazards (not shown here) and for computing targets of control-flow instructions. Target addresses are stored in the `i_tgt` and `ls_tgt` buffers where the special value `0` indicates that the buffer does not contain a valid target. Whenever the fetch unit is about to request a new address from the bus unit, it checks whether any of the two buffers contains a valid address and in that case it overwrites `bu_addr` with that address and redirects the next fetch to the branch, return or call target. Otherwise, the new value of `bu_addr` is calculated by the function `update` depending on the state of the prefetch buffer.

[1] The third pipeline for handling zero-overhead loops shares its decode stage with the LSP.

```
0xd4000056  mov d15, +21649
0xd400005a  j 0xd4000068
```

Figure 2: Tricore assembly.

Pipeline analysis starts from the initial state where `bu_addr` contains the start address and `i_tgt` and `ls_tgt` are set to zero. The reachable pipeline states are computed by repeated image computation, but at certain points we require some of the delegated information (from now on, we will refer to such cases as *external requests*). E. g. in a state q where the condition `(i_req && i_addr)` holds, we require the possible control-flow targets for the instruction at address `i_addr` in order to update the variable `i_tgt`. The update rule [2] for this variable, `i_tgt = $ND`, states that `i_tgt` may adopt any possible value in the next cycle (`$ND` stands for non-deterministic values). Thus, the image of q is a set of 2^n states that differ only in the value of `i_tgt`.

Consider the assembly code of figure 2 and let us assume that the value of `i_addr` in state $q \in Q$ is `0xd400005a`. A lookup of this address from the assembly program shows that it is an unconditional jump to address `0xd4000068`. The key idea presented in [17] is to partition a set of abstract states according to different external requests. Computing the image of each partition yields all possible successor states for that request, which can be restricted using the external information. The set of successor states for the next analysis cycle is the disjunction of the restricted images for each partition.

5. Concurrent External Requests

Consider again the assembly code of figure 2. The Tricore fetch unit can issue such a pair of instructions concurrently by assigning the move instruction to the IP and the unconditional jump to the LSP. If the signals `i_req` and `ls_req` are active during the next cycle, then both conditions `(i_req && i_addr)` and `(ls_req && ls_addr)` hold. This means that we have two external requests in the same state $r \in Q$ and $\texttt{Img}(r)$ produces 2^{2n} successor states, because *both* target buffers may adopt any value. The original algorithm as proposed in [17] handles each external request individually and therefore fails to create a correct restriction for this case. Thus, if restrict$(A, \texttt{v})$ denotes the restriction of variable v in the subset $A \subseteq Q$, then the algorithm effectively computes

$$\text{restrict}(\text{Img}(\{r\}), \texttt{i_tgt}) \cup \text{restrict}(\text{Img}(\{r\}), \texttt{ls_tgt})$$

This computation yields $2 \cdot 2^n$ states instead of the single state where `i_tgt` equals `0` (the move has no branch target) and `ls_tgt` equals `0xd4000068`.

[2] Depicted in the left decode unit in figure 1.

We therefore propose a slightly different algorithm for partitioning the set of abstract pipeline states, in order to simplify finding all restrictions for concurrent external requests. It involves the computation of all *cofactors* of the variables controlling external requests. Computing a cofactor means restricting a binary variable to either 1 or 0. We define an ordering on the decision variables for external requests, e. g.

$$\begin{array}{ccccc} \texttt{i_req} & \underset{1}{\rightarrow} & \texttt{i_addr[n]} & \rightarrow \ \ldots & \texttt{i_addr[0]} \\ \downarrow & & & & \\ \texttt{ls_req} & \underset{1}{\rightarrow} & \texttt{ls_addr[n]} & \rightarrow \ \ldots & \texttt{ls_addr[0]} \end{array}$$

All decision variables for a single external request appear in a single line and their ordering is indicated by arrows. Using this ordering, we compute a tree of cofactors as shown in figure 3. The leaves of the tree are the required partitions for external requests. In some cases, we do not need to consider the cofactors of all variables in a line. E. g. we skip cofactoring the variables i_addr[n] . . . i_addr[0] for cofactors where i_req = 0 because the expression

$$\texttt{i_req} \wedge (\texttt{i_addr[n]} \vee \ldots \vee \texttt{i_addr[0]})$$

evaluates to false for all assignments of i_addr[n] . . . i_addr[0]. This shortcut is indicated by the 1 below the ordering arrow between i_req and i_addr[n]. Shortcuts can be found easily because of the regular structure of external requests. They usually rely on the state of very few signals (like i_req) and the address part must be different from 0 (invalid address). The partitions depicted in figure 3 are

(A)	States without external requests.
(B)	States requesting i_tgt.
(C, D)	States requesting i_tgt and ls_tgt.

The states (C) and (D) differ in the instruction address ls_addr for the external request of ls_tgt. In practice, most of the possible cofactors will be empty, which means that it suffices to consider a fraction of the possible paths through the partitioning tree. The efficiency of the partitioning can be further improved by choosing a variable ordering that allows for the definition of many early shortcuts. Furthermore, the use of shortcuts avoids the partitioning of states with different addresses but without active external requests, e. g. for i_addr if i_reg = 0. A tree walk from the root to a leaf defines an assignment of all decision variables for all active external requests in that partition. Using this information, we can lookup the results for all active external requests of a partition and restrict the possible successor states as described in section 4.

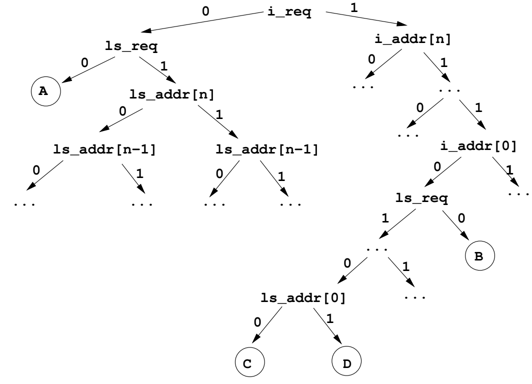

Figure 3: Partitioning tree.

6. Performance Considerations

The performance of BDD-based algorithms depends on the number of BDD variables and on their ordering. The number of BDD variables is equivalent to the number of bits needed for all variables in the abstract model. Therefore, introducing instruction addresses into all pipeline stages creates a serious problem if we use all 31 bits needed for addressing the whole address space (for Tricore, we can omit 1 bit because of alignment restrictions). We can significantly reduce the number of bits by enumerating all instructions used in the program. For external requests, we translate the numbers back using a simple table lookup. The same method can be used for compactly encoding data addresses.

The problem of finding a good variable ordering for obtaining small BDDs has been studied extensively and many heuristics exist [10]. However, finding a good ordering requires careful engineering and knowledge about the implementation of a concrete abstract model. Thus, it must be determined for each abstract pipeline model.

The number of partitions generated by our traversal algorithm depends on the number and placement of external requests in the abstract pipeline model. A large number of external requests can lead to *fragmentation* of the symbolic representation, e. g. if each state issues a different external request, we have to separate all states into singular partitions. However, we believe that the worst case is unlikely in practice and that efficient designs can be found.

7. Implementation and Experimental Results

The presented approach for symbolic pipeline analysis has been implemented in the *aiT*-framework [1], using the *VIS* [3] model checker for image computation and BDD operations. An abstract pipeline model for a subset of the Infineon Tricore has been implemented in Verilog. It comprises the shared fetch unit (fully implemented) and simplified im-

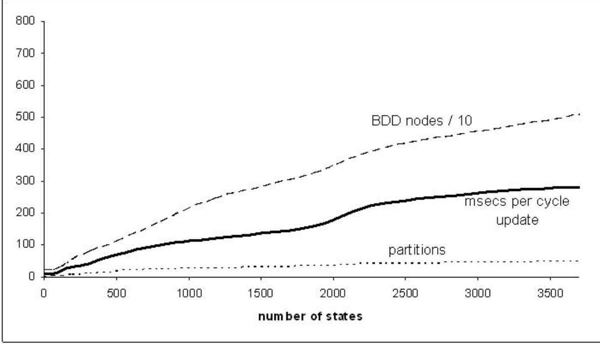

Figure 4: Cycle update times compared to number of states.

plementations of the two main pipelines (altogether about 500 lines of Verilog). The Verilog specification is translated to a netlist for VIS, using the *vl2mv* compiler [6]. The interface to aiT's program analysis framework, including the handling of external requests, is implemented in C++. The analysis has been tested successfully on several small benchmarks, e. g. dhrystone.

In order to assess the efficiency of our approach, we have increased the number of abstract states considered by the analysis by assuming different latencies for instruction fetches. This is similar to real-world problems since state explosion is often caused by uncertain information about the timing of memory accesses (e. g. cache-hit or miss). Using 63 possible latencies for each instruction fetch, we obtained the results presented in figure 4 by analyzing a program for calculating prime numbers. The solid bold curve indicates the time required for computing a cycle update for all possible abstract states at a program point. The dotted curve below shows the number of partitions generated by our algorithm and the dashed curve shows the number of BDD nodes (divided by 10). Considering that the representation of a single state requires 192 BDD nodes and approximately 10 msecs per cycle update, figure 4 shows that memory consumption (number of required BDD nodes) and computation time for cycle updates only grow slowly with the number of states. We expect that the symbolic approach will outperform an explicit implementation, as soon as the number of states reaches a break-even point.

8. Integrating Cache Analysis

We have mentioned that the AI approach uses an *integrated* cache and pipeline analysis (see section 1). In fact, the analysis operates on tuples of abstract pipeline- and cache states because of their interdependence, i. e. the order of memory accesses depends on pipeline effects and the timing of memory accesses depends on the cache state. We propose a symbolic implementation of an integrated pipeline and cache analysis such that the extra number of BDD variables for representing abstract caches remains small. Let us start by recalling some notions of cache analysis as defined in [8].

Abstract Cache Model. We deal with A-way associative caches. Let A denote the associativity of the cache. We denote by $\mathcal{M}$ the set of memory locations that are mapped into the cache. Let s be a cache set and $\mathcal{S}$ the set of cache sets. Each memory location falls into a particular cache set. The set $\mathcal{M}$ decomposes into disjoint sets $\mathcal{M} = \dot{\bigcup}_{s\in\mathcal{S}}\mathcal{M}_s$ where $\mathcal{M}_s$ are the memory locations falling into set s. Abstract caches are typically based on the concept of age. A memory location is either not in the cache, i.e. has age $\bot$, or it is in one of the A many lines of its cache set, i.e. has age k where $k \in \{1,\ldots,A\}$. The age of a memory location is an element of $\mathcal{A} = \{1,\ldots,A,\bot\}$. An abstract state of a cache set is a total function $[\mathcal{M}_s \to \mathcal{A}]$. An abstract cache state is the collection of the states of its sets:

$$\widehat{Cache} = \biguplus_{s\in\mathcal{S}} [\mathcal{M}_s \to \mathcal{A}]$$

Symbolic Representation. We exploit knowledge about the program and approximations to arrive at a compact symbolic encoding. As noted earlier, we know the memory locations that will be accessed by the program in advance, as they can be determined by value analysis. In general, the set of accessed memory locations $\mathcal{M}$ is significantly smaller than the full address space and depends on the program under analysis. Note that we encode functions that keep track of the age of particular memory locations. A straightforward encoding would be to extend the state vector by $|\mathcal{M}|$ many entries, one for each memory location, that record the age of each memory location. An advantage of this encoding is that it can be implemented using off-the-shelf image computation. However, the number of required state bits would be too high, as e. g. a data cache typically may contain tens of thousands of memory locations.

The problem with the naive encoding is that it unrolls the domain of the functions that represent cache state. In our encoding, we exploit that BDDs can encode functions directly and without unrolling, so that the number of required boolean variables is logarithmic in the number of memory locations rather than linear. For simplicity, we can assume that only one cache state is stored per pipeline state. States that agree in terms of pipeline state can be merged to one state by using the join operator for abstract caches. The symbolic domain consists of partial functions from pipeline to cache state:

$$\widehat{Pipe} \hookrightarrow \widehat{Cache}$$

An element of this domain represents a set of abstract states and is stored in a single BDD. The BDD contains the

boolean variables of the pipeline state plus boolean variables that address a particular cache set, a memory location, and age, respectively. One can use binary encoding for the set of cache sets, the sets of memory locations and ages, respectively. One obtains the following number of required boolean variables in the BDD:

$$\|\widehat{Pipe}\| + \underbrace{\lceil \log_2 \mathcal{S} \rceil + \lceil \log_2 \max_{s \in \mathcal{S}} |\mathcal{M}_s| \rceil + \lceil \log_2 |\mathcal{A}| \rceil}_{\|\widehat{Cache}\|} \quad (1)$$

where $\|\widehat{Pipe}\|$ denotes the number of bits required for pipeline state.

As mentioned earlier, BDDs typically represent characteristic functions in symbolic state traversal [5]. This is not the case for the proposed symbolic representation and therefore a modified image computation algorithm is required. However, space precludes us from discussing this algorithm.

Let us briefly assess the compactness of the representation on a real-life example, a task set obtained from an industry project. The task set was compiled for a PowerPC 755 processor featuring a 2-way associative cache with 128 sets and 32 bytes line size. We obtained (an overapproximation of) the set of accessed memory locations $\mathcal{M}$ by value analysis. The maximum number of memory locations that fall into a cache set is 94 in this example. Overall there are about ten thousand memory locations. Using formula 1, we see that the extra number of bits required for the data cache is 16 ($|\mathcal{S}| = 128$, $\max_{s \in \mathcal{S}} |\mathcal{M}_s| = 94$,$|\mathcal{A}| = 3$). In the example, the instruction cache can potentially contain only about a thousand locations. Thus we only need one more bit to switch between instruction and data cache in the representation (analogously to how we use bits to address different sets in a cache). The overall number of bits required for the cache is thus 17.

9. Conclusion

We have shown that symbolic pipeline analysis is a static program analysis which uses the same abstractions as proposed in [15]. The use of a symbolic representation for abstract pipeline states requires algorithms for efficiently encoding delegated information when computing state transitions. We have illustrated our solution using the notion of external requests (sections 4 and 5) and described its effects on the performance of the analysis.

Currently, our implementation only supports a pipeline analysis. We have outlined ideas that admit an integrated cache and pipeline analysis without significantly increasing the number of state bits compared to stand-alone pipeline analysis (section 8). In the future, we will implement the integrated analysis.

References

[1] http://www.absint.com/aiT/.

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

[3] 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.

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

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

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

[7] A. Colin and S. M. Petters. Experimental Evaluation of Code Properties for WCET Analysis. In *RTSS '03: Proceedings of the 24th IEEE International Real-Time Systems Symposium*, page 190, Washington, DC, USA, 2003. IEEE Computer Society.

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

[9] 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.

[10] S.-W. Jeong, B. Plessier, G. Hachtel, and F. Somenzi. Variable Ordering for FSM Traversal. In *Proceedings of the International Conference on Computer-Aided Design*, 1991.

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

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

[13] J. Souyris, E. Le Pavec, G. Himbert, V. Jgu, 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 Worst-Case Execution Time (WCET) Analysis*, pages 21–24, 2005.

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

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

[16] 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 2003 International Conference on Dependable Systems and Networks (DSN 2003)*, pages 625–632. IEEE Computer Society, 2003.

[17] S. Wilhelm. Efficient Analysis of Pipeline Models for WCET Computation. In *Proceedings of the 5th Intl. Workshop on Worst-Case Execution Time Analysis*, 2005.

[18] S. Zarnescu. *TriCore Pipeline Behaviour & Instruction Execution Timing*. Infineon Technologies, 2001.