# Why Model Checking Can Improve WCET Analysis

Alexander Metzner

Safety critical Embedded Systems group
Department of Computer Science
Carl von Ossietzky University Oldenburg
metzner@informatik.uni-oldenburg.de

**Abstract.** Calculating predictions for an upper bound of the execution time of real-time tasks in embedded systems is a necessary step in designing such systems. There exist successful analysis methods, based on abstract interpretation and integer linear programming (ILP) for that problem. In [12] it is stated, that model checking *is not adequate* for this task. The approach presented in this paper shows that model checking *is adequate* and, furthermore, can improve the results. This is done by defining an automaton based semantic for control flow graphs of programs for abstract and concrete instruction cache analysis. A binary search based bunch of model checker runs is used to calculate the upper bound of execution time.

## 1 Introduction

Schedulability analysis is one of the key factors in the design of a hard real-time system, that has to fulfill timing constraints stemming from the interaction with a physical environment. Particularly, in safety critical systems, as used in the domains of automotive, avionics or power-plant electronics, the guarantee to meet given deadlines is a stringent property of the system. To prove this property, common schedulability analysis methods[6] need an upper bound for the execution time of programs, so-called worst case execution time (WCET). This bound has to be safe and accurate: it will never under-estimating the real behaviour. Furthermore it should be as close as possible to the real maximal execution time of the program. Deriving such execution time bounds from software is a challenging task, since modern processor and cache architectures have to be taken into account. The dynamics of programs, ie. which paths through a program are valid or not, also influence the WCET, independent from the architecture. Therefore, usually the analysis of WCET is separated into two tasks, namely the low level analysis (dealing with architectural features) and the high-level analysis (dealing with program paths)[11]. In this paper, we focus on the first one, but give some hints how possibly to combine this with a high-level analysis.

There exist several approaches to tackle the low-level analysis with respect to caches and pipelines. In general, the low level analysis consists of two subproblems. Firstly, the timing behaviour of instructions of a program must be

R. Alur and D.A. Peled (Eds.): CAV 2004, LNCS 3114, pp. 334–347, 2004.
© Springer-Verlag Berlin Heidelberg 2004

predicted. Secondly, the longest path through the program must be calculated by using the predicted instruction timing. In [10] an abstract cache behaviour classification is defined, that statically predicts the worst case behaviour of each memory reference. [11] derives a similar classification of memory accesses using an abstract interpretation for cache behaviour prediction and integer linear programming for finding the longest path by a method called implicit path enumeration[7]. Only ILP is used in [8] for pipeline and simple cache analysis, whereas [3] uses data-flow analysis methods for pipeline and cache analysis.

Both tasks, the behaviour prediction and the longest path finding problem, can be solved by a model checker[9]. Since a model checker traverses the entire state space of a model and therefore returns a concrete path in the program, better results can be obtained than all mentioned approaches above, because of their abstract nature. However, [12] claims model checking to be an inadequate technique for this analysis task. The approach presented in this paper will show, that model checking *can* be used for WCET analysis and furthermore it could be a benefit to use it. In order to prove this, we have implemented an experimental framework for an instruction cache analysis within the OFFIS verification environment[1] based on the VIS model checker[4].

## 2 Basic Modelling

Calculations of the WCET for programs are performed on the control flow graph. In this section we define the construction of the control flow graph of a program from the program code. Further, we define an automaton based representation of the control flow graph that is used in the next chapter to build the semantics for WCET analysis.

The starting point of a WCET analysis in a low level manner are executables, object code or assembler code. Our approach analyses timing behaviour of a program at assembler code level. For effectiveness reasons, the control flow graph is built from blocks of assembler instructions instead of single instructions. The idea is, to merge strongly connected instructions to so called basic blocks[10], that consists of the longest sequence of instructions with only one entry and one exit.

**Definition 1 (Basic Block).**
*A basic block is constructed from a control flow of an assembler program by merging sequential instructions to blocks of instructions. Only the first instruction of a block may be target of a jump and the last instruction must be a jump or the last instruction of the program. There are no other jumps within this block.*

**Definition 2 (Basic Block Graph (BBG)).**
*A basic block graph is a tuple* $\mathcal{B} = (BB, T_{\mathcal{B}}, \mathcal{L})$ *with:*

– $BB$ *set of basic blocks of a program*
– $T_{\mathcal{B}} \subseteq BB \times BB$ *transition relation according to the control flow*

- $\mathcal{L} \subseteq BB \times BB \times \mathbb{N}_0 \times \mathbb{N}^+$ *the labelling relation for loops (minimal and maximal iteration count)*

*For each basic block $b_i \in BB$ that is a head of a loop the following attributes are defined:*

- $backjump : BB \rightarrow 2^{BB}$*: All source basic blocks that jump backwards to $b_i$*
- $exits : BB \rightarrow 2^{BB}$*: All basic blocks which are outside the loop headed by $b_i$ and which are targets of basic blocks inside the loop*
- $cost : BB \rightarrow \mathbb{N}_0$ *the number of processor cycles of an execution of each basic block in the worst case*

Note, that the assembler instructions of each basic block are considered in the *cost* attribute. The attribute *backjump* denotes all basic blocks that lead to a further iteration of loops, whereas *exits* denotes all blocks that finish the loop. The labelling relation must be defined by the user or a high level analysis in order to avoid unbounded loops and defines the possible interval of iterations for each loop. It has to be defined for each backjump basic block to the loop head.

For simplicity reasons, we restrict our analysis to the prediction of instruction cache related timing, so an analysis of the pipeline behaviour with respect to timing is assumed to be given with the *cost* attribute. (Below, we will give a hint, how the integration of pipeline analysis is possible, too).

The basic block graph can be statically derived from the assembler program and represents a control flow graph of the program. In order to calculate a safe upper bound of the execution time by using model checking techniques, we represent the basic block graph by a finite state machine, that contains all possible paths through the program.

**Definition 3 (Basic Block Automaton (BBA)).**
*The Basic Block Automaton is a tuple $BBA = (s_0, S_T, S, \mathcal{I}, V, T)$ with:*

- $S$ *the set of states*
- $s_0 \in S$ *the initial state*
- $S_T \subseteq S$ *the set of termination states*
- $\mathcal{I}$ *the set of inputs (used to resolve non-determinism)*
- $V$ *the set of variables (used for loop counters, cache contents and consumed time)*
- $T \subseteq S \cup \{\bot\} \times S$ *the transition relation. The $\bot$ element is used to define the init transition.*

*For each transition $t \in T$ the following functions are defined:*

- $guard : T \rightarrow BEXPR^{V \cup \mathcal{I}}$ *a guarding condition of type boolean expression*
- $action : T \rightarrow 2^{ASSIGN_V}$ *with* $ASSIGN_V = \{v := e \mid v \in V \wedge e \in EXPR^{V \cup \mathcal{I}}\}$ *an action on each transition to conditionally update the system variables. EXPR is defined as usual, with the notation of* $\mathrm{c?e_1 : e_2}$ *for conditional expressions.*

We will define the semantics of a BBG in terms of a BBA. The main idea is to represent all paths through a BBG by an automaton that jumps each step from basic block to basic block. A sequence of states represents a path in the BBG. On each step the emerging costs of the successor state are added to a global variable, which contains the consumed time (from the init state to the actual state), ie. the runtime on the actual path. In the next chapter we will show for two different kinds of semantics, how this is done with respect to the cache architecture and the BBG.

## 3 Two Different Semantics for BB Graphs

The well known method from [11] to calculate safe and accurate WCETs uses a mixture of abstract interpretation (AI) for cache behaviour classification and integer linear programming to derive the worst case path through a BBG. To handle this problem by model checking, one can think of mainly two different approaches: First, model checking can replace the integer linear programming in order to solve only the subtask of finding the longest path. Secondly, it can replace both subtasks, the path finding as well as the prediction of cache behaviour, to solve the entire problem. Changing only the method of critical path finding will not lead to more accurate WCETs with respect to ILP, but it will help to avoid ILP specific problems. In section 3.1 the semantic of this approach is described.

On the other hand, employing the model checker for precise cache behaviour prediction can improve the results due to the avoidance of over-approximations of the abstraction methods of AI. We describe the semantics for precise cache behaviour prediction in section 3.2.

### 3.1 Model Checking for Path Finding

If the task for model checking is reduced to finding the longest path in a BBG, we have to assume a pre-calculated prediction of the behaviour of cache accesses. This is done by the prediction of the worst case hit/miss behaviour and is derived statically by an implementation of the abstract interpretation formalisation found eg. in [11]. The idea is to collect abstract states of the cache contents in order to compute an over-approximation of timing. In the next section, when we show why model checking can lead to more accurate results, this method will be explained in more detail. Here we are only interested in the classification for all memory accesses. Each memory access is predicted to behave like *always-miss (am)*, *always-hit (ah)* or *first-miss (fm)*. Memory accesses of type “am” will produce a cache miss each time. “fm” means that only the first access leads to a cache miss, all further accesses will produces cache hits. Finally, “ah” indicates a cache hit for every access to this memory location.

Since the approach presented in this paper is restricted to the behaviour of instruction caches, this analysis is performed for each instruction in a basic block $b_i$. For efficiency reasons, the cache behaviour prediction of instructions

within one basic block are merged and the syntax of a BB is extended by the two mappings (all accesses that are not handled by these two mappings are treated as always hit):

- $get_am : BB \rightarrow \mathbb{N}_0$ number of always miss in a BB
- $get_fm : BB \rightarrow \mathbb{N}_0$ number of first miss in a BB

Exploiting this knowledge we can abstract from the concrete cache architecture. We only have to consider the penalty of cache hits and misses (in processor cycles). For pipeline architectures we assume, that each instruction can be executed within one cycle, including the hit penalty of the instruction memory access (pipelined RISC concept).

The main idea of the semantic for path finding is to sum up the emerging costs (in processor cycles) of all basic blocks on a concrete path through the BBA in a variable *cycles*. This is performed at the transitions to each block. The addend is constant for the fixed costs of instruction execution in the pipeline and the cache accesses that are predicted as “am” and “ah”. In order to handle the category first-miss, we have to introduce one boolean variable $f_{b_i}$ for each basic block $b_i$, if there exist “fm” accesses within this block. The content of the $f_{b_i}$ variable indicates, whether this is the first time of access or a subsequent one. Consequently, the cost that has to be added to *cycles* on entering $b_i$ contains the miss and the hit penalty, respectively.

In order to resolve the non-determinism that occurs in each branch in the BBG (conditional jumps in the assembler program are the source of these non-determinism, because at this low level analysis we do not care about the conditions in the source program), an input $n_{cond}$ is required. Since an input, determining the chosen path of a branch, is independent from all other branches in each step, one input $n_{cond}$ is sufficient for this task.

Loop iteration bounds are used to avoid unbounded unrolling of loops. As always the case in real-time systems, software with divergent loops is not allowed and for the worst case analysis we need the interval of loop iterations for each loop.

Therefore, for each loop a local loop counter $l_{b_i}$ is inserted, that contains the actual number of loop unrolls for the current path. Figure 1 shows the transformation from a loop in the BBG to the BBA. For each transition to a loop head, that is not a backward jump from within the loop, the loop counter is initialised with 0. On each backward jump towards the loop head, the loop counter is incremented to indicate a new iteration. Exiting the loop is performed by taking a transition to a basic block outside the loop, which we called exit block. The path to an exit block always uses branching blocks[1], thus the transformation has to follow the rule of

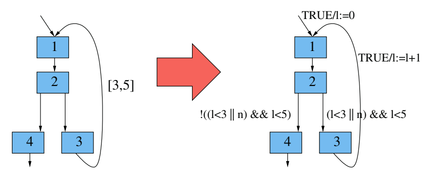

**Fig. 1.** Semantic transformation of loops

[1] Otherwise it would be no exit block, because if a block within a loop has only one successor, this target block must be within the loop, too.

introducing an input variable $n_{cond}$. However, this alone would allow unbounded loop iterations, hence we have to take the iteration bounds into account. This is done by an extended guard at the transition of branching blocks, that leads to a loop exit $(\neg((l_{b_i} < m_d \vee n_{cond}) \wedge l_{b_i} < m_u))$. Exiting the loop by using the input variable $n_{cond}$ is only possible, if the number of iterations is greater than the minimal iteration count ($m_d$) and is enforced if the maximal iteration count ($m_u$) is exceeded (see figure 1)[2]. This leads to the following definition:

**Definition 4 (BBA semantics of a BBG).**
*The semantics* $BBA = (s_0, S_T, S, \mathcal{I}, V, T)$ *of a given basic block graph* $\mathcal{B} = (BB, T_{\mathcal{B}}, \mathcal{L})$ *is defined by the following rules:*
*Let* $S_{head} = \{b \mid b \in S \wedge \exists b_i \in BB, \exists l = (b_i, b, \cdot, \cdot) \in \mathcal{L}\}$ *be the set of all loop heads within* $\mathcal{B}$.

- $S = BB$
- $s_0 = b$ *with* $\exists b_i \in BB, t_{\mathcal{B}} \in T_{\mathcal{B}} : t_{\mathcal{B}} = (b, b_i) \wedge \forall b_j \in BB : \nexists t_{\mathcal{B}} \in T_{\mathcal{B}} : t_{\mathcal{B}} = (b_j, b)$
- $S_T = \{b \mid b \in BB \wedge \forall b_i \in BB : \nexists t_{\mathcal{B}} \in T_{\mathcal{B}} : t_{\mathcal{B}} = (b, b_i)\}$
- $\mathcal{I} = \{n_{cond}\}$ *with* $n_{cond} \in BOOL$
- $V = \{cycles\} \cup \{l_{b_j} \mid b_j \in S_{head}\} \cup \{f_{b_j} \mid b_j \in BB : get_fm(b_j) > 0\}$
- $T = T_{\mathcal{B}} \cup \{t_{init}\}$ *with* $t_{init} = (\perp, s_0)$

*With a transition* $t = (b_1, b_2) \in T$ *it holds:*

$$guard(t) = \begin{cases} n_{cond} \in \mathcal{I} & \text{if } \exists b_i \neq b_2 \in BB, \exists t_i = (b1, b_i) \in T_{\mathcal{B}} \\ & \quad \wedge\ guard(t_i) = \neg n_{cond} \wedge b_i \notin S_{head} \\ \neg n_{cond} \in \mathcal{I} & \text{if } \exists b_i \neq b_2 \in BB, \exists t_i = (b1, b_i) \in T_{\mathcal{B}} \\ & \quad \wedge\ guard(t_i) = n_{cond} \wedge b_i \notin S_{head} \\ \neg((l_{b_i} < m_d \vee n_{cond}) & \\ \quad \wedge l_{b_i} < m_u) & \\ \text{with } l_{b_i} \in V \wedge n_{cond} \in \mathcal{I} & \text{if } \exists b_i \in S_{head} : b_2 \in exits(b_i) \\ & \quad \wedge \exists \lambda = (\cdot, b_i, m_d, m_u) \in \mathcal{L} \\ (l_{b_i} < m_d \vee n_{cond}) & \\ \quad \wedge l_{b_i} < m_u & \\ \text{with } l_{b_i} \in V \wedge n_{cond} \in \mathcal{I} & \text{if } \exists b_i \in S_{head}, \exists b_j \in BB, \\ & \quad \exists t_j \in T_{\mathcal{B}}, \exists \lambda \in \mathcal{L} : \\ & \quad b_j \neq b_2 \wedge b_j \in exits(b_i) \\ & \quad \wedge t_j = (b_1, b_j) \\ & \quad \wedge \lambda = (\cdot, b_i, m_d, m_u) \\ TRUE & \text{else} \end{cases}$$

[2] Using the first exit in case of bound exceeding is a structural property of assembler code for loops that is constructed by common compilers.

*For each basic block $b_j$ let $c_{am}(b_j) = get_am(b_j) \times MISS_PENALTY$ be the cost of a cache miss of type "am". Let $c_{fm} = get_fm(b_j) \times MISS_PENALTY$ the similar attribute for "fm" classified misses. Let $action_n$ (assignment for cache and pipeline costs) be defined as follows:*

$$action_n(t) = \begin{cases} \{cycles := cost(s_0) + c_{am}(s_0) + c_{fm}(s_0)\} & \\ \cup\{f_{b_j} := FALSE \mid \forall f_{b_j} \in V : b_j \neq s_0\} & \\ \cup\{f_{s_0} := TRUE\} & \text{if } t = t_{init} \\ \{cycles := cost(b_2) + c_{am}(b_2) & \\ \qquad + (f_{b_2})?c_{fm}(b_2) : 0\} & \\ \cup\{f_{b_2} := TRUE\} & \text{if } get_fm(b_2) > 0 \\ & \quad \wedge\ t \neq t_{init} \\ \{cycles := cycles + cost(b_2) + c_{am}(b_2)\} & else \end{cases}$$

*To avoid unbound unrolling of loops, $action_l$ is defined:*

$$action_l(t) = \begin{cases} \{l_{b_2} := 0\} & \text{if } b_2 \in S_{head} \wedge b_1 \notin backjump(b_2) \\ \{l_{b_2} := l_{b_2} + 1\} & \text{if } b_2 \in S_{head} \wedge b_1 \in backjump(b_2) \\ \emptyset & else \end{cases}$$

*With this two parts, the action function can be constructed by*

$$action(t) = action_n(t) \cup action_l(t)$$

This BBA of a given BBG can now be translated to a model for a model checker. What we can do with model checking techniques, is the proof of $M_{BBA} \models \mathbf{EF}(\phi_N)$, where $\phi_N$ is the state formula:

$$\phi_N = \Big(\bigvee_{t:b_t \in S_T} b_t\Big) \wedge \text{cycles} > N$$

The target of this proof is a path to one of the termination blocks whose cost (in processor cycles) is greater than $N$. Since searching for WCET is an optimisation problem with the objective function of $max(cycles)$, we must execute a bunch of model check runs to converge to the real WCET. This is performed in a binary search manner and we do an update on $N$ each run: If $M_{BBA} \models \mathbf{EF}(\phi_N)$ fails, the value of $N$ has to be decreased, otherwise it has to be increased[3]. If we have found a proof for $M_{BBA} \models \mathbf{EF}(\phi_{N_i-1})$ and the proof of $M_{BBA} \models \mathbf{EF}(\phi_{N_i})$ fails, we have reached the maximum for the variable *cycles* and thus we have found, that the WCET equals $N_i$.

### 3.2 Model Checking for Cache Behaviour Prediction

As mentioned above, the former approach can compete with other techniques, like ILP based methods presented in [11]; it shows that with model checking techniques a low level analysis of WCET is possible as well. Furthermore, the

[3] Note, that with very slightly changes the BCET can easily be analysed with the same technique.

ILP method is known to suffer from numerical instabilities that can produce non-valid solutions. In contrast, model checking based techniques by construction do not suffer from problems like this and always calculate valid solutions.

To get some improvements on WCET analysis, we try to transfer the prediction of cache accesses from the statical way to the dynamic way under the control of a model check run. This increases complexity, but, however, we can avoid too pessimistic over-approximations of the static analysis.

Static analysis, such as AI based methods, do not consider concrete paths through a program but work with abstract ones. The prerequisite for this abstraction is a safe over-approximation of the cache behaviour. Since model checking considers a concrete path, at some points in a control flow graph the pessimistic prediction of abstract cache behaviour can be improved. These points are basic blocks, that are targets of more than one predecessor block. Regarding the abstract cache behaviour, the abstract analysis method is not able to determine, which predecessor is used (because this is only known in a concrete path). Therefore, at these points so-called *join*-functions merge all possible paths in a safe way with respect to WCET.

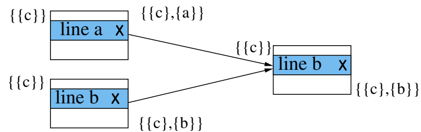


**Fig. 2.** Merging paths in an abstract cache behaviour analysis. Crosses mark cache miss prediction.

For example, in figure 2 the lower block loads a cache line, that is used in the successor block, but the upper block does not use this line. In order to give safe predictions, the *join-function* does some kind of set intersection of abstract cache states with the result, that in an "am" or "fm" analysis this cache accesses in both blocks leads to a cache miss for cache line $b$. This also holds, if the concrete path uses the block that pre-loads cache line $b$.

To overcome this drawback, we give in this section a BBA based semantic for BBGs with dynamic cache analysis. We will reuse parts of the above defined semantic, namely most of the construction rules and the whole *guard* function. To model the cache behaviour of concrete paths, we introduce new variables to represent the contents of the lines of an instruction cache with given size, associativity and other parameters.

In order to model dynamic cache contents on a concrete path, the whole cache memory must be taken into account. Each access to instruction memory is deterministically mapped into one set of the cache and belongs to a so called "program line" [10]. Program lines are the set of instruction memory locations in a program that are mapped to the same cache set with the same tag (see figure 3). If we assume a critical word first loading strategy

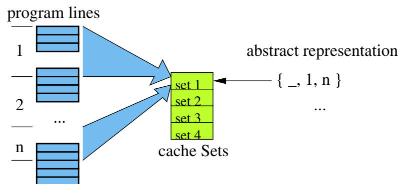


**Fig. 3.** Abstract cache content representation.

(modern processors usually implement this strategy), only the first access of instructions within one program line to a cache set leads to a miss. Hence, we can abstract without loss of precision from collecting each memory location access by collecting only the program lines. While traversing a concrete path, we collect the program lines per cache line and set, that are loaded, and we delete the program lines that are replaced (depending on the replacement strategy). This is performed in the sequence of memory accesses on the concrete path. Therefore, the definition of a basic block graph is extended by the set of program lines of each BB:
$pl : BB \rightarrow 2^{\mathbb{N}^+}$ denotes all program lines of a basic block
$set : N^+ \rightarrow \mathbb{N}^+$ gives the cache set that belongs to program line p
Note, that we assume implicitly, that the size of the cache is sufficient to record all program lines of a basic block[4]. With this additional information we can now define the BBA-based semantic for a basic block graph with cache behaviour prediction.

**Definition 5 (BBA semantic of BBG with cache behaviour prediction).**
*The semantics* $BBA = (s_0, S_T, S, \mathcal{I}, V, T)$ *of a given basic block graph* $\mathcal{B} = (BB, T_{\mathcal{B}}, \mathcal{L})$ *is defined by the following rules:*

- *Let* $S_{head}, s_0, S_T, \mathcal{I}, T$ *be defined as in definition 4.*
- $V = \{cycles\} \cup \{l_{b_j} \mid b_j \in S_{head}\} \cup \{IC_{g,h} \mid IC_{g,h} \in \mathbb{N} \cup \{\bot\} \wedge \mathrm{g} \in \{1, \ldots, SETS\} \wedge \mathrm{h} \in \{1, \ldots, A\}\}$ *with SETS is the number of cache sets, A is the associativity of the cache and* $\bot$ *denotes an invalid cache line*
- *Let guard*$(t)$ *and action*$_l(t)$ *be defined as in definition 4*

*Let the cost of memory accesses of a basic block b on a cache with associativity A be defined as*

$$c_{cache}(b) = \sum_{p \in pl(b)} \left( \bigvee_{i=\{1,\ldots,A\}} (IC_{set(p),i} = p) \right) ?0 : MISS_PENALTY$$

*and the cost of entering the first basic block by transition* $t_{init}$ *as*

$$c_{init} = \sum_{p \in pl(s_0)} MISS_PENALTY$$

*With* $t = (b_1, b_2) \in T$*, the assignments for cache and pipeline costs are defined:*

$$action_n(t) = \begin{cases} \{cycles := cost(s_0) + c_{init}\} & \text{if } t = t_{init} \\ \{cycles := cycles + cost(b_2) + c_{cache}(b_2)\} & \text{else} \end{cases}$$

*The assignments for dynamic cache prediction action*$_c$ *is defined with*

$$hit(p) = \bigvee_{i=\{1,\ldots,A\}} (IC_{set(p),i} = p)$$

[4] This is the standard case in modern processors, since basic blocks are typically small.

$$action_c(t) = \begin{cases} \{IC_{p,a} := \bot \mid p \in \{1,\ldots,SETS\} \backslash \{set(pl(s_0))\} \\ \qquad\qquad \wedge\ a \in \{1,\ldots,A\}\} \ \cup \\ \bigcup_{p \in pl(s_0)} \{(hit(p))?AGE(p) : LOAD(p)\} & \textit{if } t = t_{init} \\ \bigcup_{p \in pl(b_2)} \{(hit(p))?AGE(p) : LOAD(p)\} & \textit{else} \end{cases}$$

*With these three parts, the action function can be constructed by*

$$action(t) = action_n(t) \cup action_l(t) \cup action_c(t)$$

The concrete cache contents (ie. the program line actually stored in the cache lines) are represented in $(SETS \cdot A)$ $IC_{s,a}$ variables, that are updated on each memory reference. The cost of an access depends on the hit/miss state and is calculated by checking whether the program lines of the target block are within the cache variables or not. In the init transition a completely invalidated cache is assumed (worst case scenario).

The real cache behaviour of loading lines is hidden by the $AGE(p)$ and $LOAD(p)$ macros, because they depend on the used cache architecture. For a two-way associative cache ($A = 2$) with the popular LRU strategy, $AGE(p)$ and $LOAD(p)$ are defined with $s = set(p)$ (the :=: notation denotes swapping the contents):

$$AGE(p) \equiv (IC_{s,2} = p)?IC_{s,2} :=: IC_{s,1} \ : < null >$$

$$LOAD(p) \equiv IC_{s,2} := IC_{s,1} \ ; \ IC_{s,1} := p$$

For higher degrees of the associativity this is a bit more complex, but the basic scheme is the same, even if we have more complicated and less predictable (in an abstract analysis) replacement schemes, like the pseudo-LRU used in the Embedded Pentium[5].

## 4 Evaluation and Discussion

For an evaluation of the two described approaches, first, the design-flow and construction of the model from an assembler program is described. A very short sketch of our model checking environment is given, too. Using this framework, the result of an evaluation of three example case studies will be demonstrated and discussed. As mentioned above, we restrict the evaluation on the behaviour of an instruction cache and assume a “perfect” pipelined RISC architecture with a priori generated costs of one cycle per instruction.

### 4.1 Design-Flow

Starting point of the analysis is a C program with user annotations to bound the number of loop iterations. This is transfered to assembly language whereby the annotations are preserved. From this assembler program, we build the basic block graph and generate the BBA, which itself is a C program that represents the given semantic as a finite state machine. This is the base for the use of the

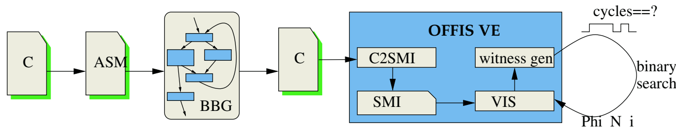


**Fig. 4.** The design-flow of WCET analysis using model checking techniques provided by the OFFIS verification environment.

OFFIS verification environment[1], in which the C program is transformed[2] to an input language for the model checker core. As model checker engine we use the VIS[4]. Figure 4 gives an overview over our design-flow for WCET analysis.

For the inquiry whether $M \models \mathbf{EF}(\phi_N)$, we use the *drive to property* checking of the OFFIS VE (technically an invariance check) and produce a witness in case of a result of true. The witness contains the value of the *cycles* variable for the concrete path to one of the termination basic blocks. The result is used to increase or decrease the bound $N$, until we reach the termination point, where it holds $M_{BBA} \models \mathbf{EF}(\phi_{N_i - 1}) \wedge M_{BBA} \not\models \mathbf{EF}(\phi_{N_i})$.

### 4.2 Experimental Results

For evaluation of our approach we take three case studies as inputs for the above described design-flow to calculate the WCET. The case studies are derived from running projects and are located in the domain of embedded systems, partly implemented with typical CASE tools of an embedded systems development process (see table 1 for details).

**Table 1.** Case studies with number of instructions, description and source of the code.

| *case study* | *instructions* | *comment* |
|---|---|---|
| robot | $\sim 250$ | control of a LEGO mindstorm to find a light source and react on collisions, hand written C code |
| collision | $\sim 1100$ | collision detection and avoidance for a satellite control unit, derived C code from STATEMATE specification |
| flight | $\sim 2500$ | flight control unit to compensate side drift, generated C code from SCADE |

As cache parameters we choose an instruction cache of 128 sets with 8 instructions (4 bytes per instruction) per line and an associativity of two (altogether a cache size of 8KB; the size is typical for processors used in todays embedded systems). The evaluation machine was a SUN BLADE 2000 (900 MHz UltraSparc III processor, 2 GB main memory) and the results are summarised in the table 2 below. The times in *time(A)* and *time(C)* are the time to find the WCET,

including witness generation. As mentioned above, the results of *WCET(A)* are identical to the WCET the approach in [12] would calculate. Hence we can directly compare [12] and the results from *WCET(C)* to highlight the improvement of our approach. The time of one single model check run ranges from 8 seconds for *robot* to 21 seconds for *flight* with abstract cache analysis and from 13 seconds for *robot* to 315 seconds for *flight* with concrete cache analysis. However, the results have shown, that WCET analysis with model checking techniques is possible and can partly improve the results of the abstract analysis on case studies, that are typical in their size and complexity for tasks in embedded systems.

To limit the number of iterations in the binary search procedure for the abstract cache analysis, we determine an interval, that was derived at the lower bound by a reachability analysis[5] and on the upper bound by a rule of thumb. For the concrete cache analysis we use the result of the abstract analysis and the number of joins to decrease the number of iterations (shown by *iter. (A/C)* in table 2).

**Table 2.** Evaluation of the three benchmarks with results and runtime for the abstract and the concrete cache behaviour analysis and maximal iterations of the binary search. The letter $A$ denotes the abstract, $C$ the concrete cache analysis. *bits* denotes the state bits of the problem.

| *case study* | *WCET(A)* | *bits(A)* | *WCET(C)* | *bits(C)* | *time(A)* | *time(C)* | *iter. (A/C)* |
|---|---|---|---|---|---|---|---|
| robot | 387 | 35 | 376 | 75 | 0:46 min | 0:36 min | 9/4 |
| collision | 502 | 38 | 478 | 135 | 2:00 min | 4:56 min | 10/5 |
| flight | 1782 | 40 | 1749 | 362 | 2:16 min | 15:24 min | 11/4 |

### 4.3 Discussion and Perspectives

As mentioned in the last chapter, the improvement of the concrete cache analysis compared to abstract cache analysis is founded in the more accurate cache behaviour prediction at merging points in the control flow graph. Analysing the number of these points can give a good hint whether a concrete cache analysis could be profitable or not. In our case studies, we determine that 33% to 50% of the joins lead to additional cache miss predictions in the abstract cache analysis with respect to the concrete one. As a whole this results in an improvement of 1.5% to 5% of the predicted WCET, if only instruction caches are taken into consideration.

The investigation of concrete paths increases the complexity, as shown by *bits* in table 2. The used operations (addition and comparison with constants) are not expected to lead to state explosion problems. Thus the main factor for complexity are the $IC_{s,a}$ variables which depend on the cache architecture and the size of the program. Since we are dealing with tasks (parts of functions), program sizes greater than the flight examle are quite unusual. Furthermore,

[5] The reachability analysis checks for the reachability of one termination block and returns a witness for the *cycles* variable within a few seconds on each of the models.

cache lines, that are not used, are sliced from the BBA during construction. This could also be possible for the associativity: For cache lines that are known to store only one program line there is no need for more than one cache line in the BBA, thus we can assume for this lines an associativity of one). This is currently not implemented but would decrease the number of state bits.

Regarding pipeline analysis, there occurs the same problem at merging points, since without a concrete path the analysis method cannot determine which of the predecessor blocks is on the path. Hence there is an over-approximation by taking the maximal delay, too. The consequences to the accuracy of the execution time computation are hard to predict, because this deeply depends on the used processor architecture. In well balanced pipelines there are no great differences in the delay of two predecessor blocks, but in pipelines with instructions of various execution time intervals there can possibly be ten's of cycles difference (for example using floating point execution units is very popular but yields exactly this problem). We conjecture, that with modern processor architectures the improvement of a concrete analysis could reach the same magnitude as for the cache analysis. Note, that for the pipeline analysis we do not need a pipeline behaviour analysis under control of the model checker. A pre-analysis can provide path specific delays for each basic block that can be integrated in the BBA in a similar way as we do this for caches at the moment. Thus, the complexity of the problem should not grow dramatically. If we take the same for data caches into account, the use of model checking can reach an improvement of about 10% in average.

More over, the described technique will allow for a more accurate path validation of WCET analysis by combining it with a high level analysis. In a high level analysis, for example, the mutual exclusiveness of parts of the paths can be proven. This can simply be exploited in our analysis by introducing new inputs $n_i$ at the branching blocks instead of the one input used now. By formulating an assumption over these inputs, the impossible paths through the control flow graph can be avoided and a more accurate WCET prediction can be achieved.

## 5 Conclusion

As a conclusion, our experimental environment for WCET analysis using model checking techniques and the evaluations with some typical embedded systems examples have shown, that this approach not only is able to produce valid execution times, but also can improve the results of the analysis with respect to the well known abstract analysis. Furthermore, for the abstract cache analysis, our approach always produces correct runtime predictions, different from the ILP approach that sometimes suffer from numerical instabilities.

Altogether we have shown, that model checking can be used quite well to solve the problem of determining execution time bounds. The application in a verification framework for the development of embedded systems opens a wide area of coupling the WCET analysis directly to the design in a CASE tool and promises further use cases, for example the combination with property checking for a high level path analysis, that can support the low level analysis, or the combination with automatic code generators.

## References


1. T. Bienmüller, W. Damm, and H. Wittke. The STATEMATE Verification Environment – Making it real. In *12th international Conference on Computer Aided Verification*, number 1855 in LNCS, 2000.
2. W. Damm, C. Schulte, M. Segelken, H. Wittke, U. Higgen, and M. Eckrich. Formale Verifikation von ASCET Modellen im Rahmen der Entwicklung der Aktivlenkung. *Lecture Notes in Informatics*, P-34, 2003.
3. S.-S. Lim et al. An accurate worst case timing analysis for RISC processors. *Software Engineering*, 21(7):593–604, 1995.
4. The VIS Group. VIS : A System for Verification and Synthesis. In *8th international Conference on Computer Aided Verification*, number 1102 in LNCS, 1996.
5. Intel Corp. *Intel Embedded Pentium Processor Family Dev. Manual*, 1998.
6. M. Joseph, editor. *Real-time Systems Specification, Verification and Analysis*. Prentice Hall International, London, 1996.
7. Y. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Workshop on Languages, Compilers and Tools for Real-Time Systems*, pages 88–98, 1995.
8. Y. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *IEEE Real-Time Systems Symposium*, 1995.
9. K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
10. F. Mueller. *Static Cache Simulation and its Applications*. PhD thesis, Dept. of CS, Florida State University, 1994.
11. H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2/3):157–179, 2000.
12. R. Wilhelm. Why AI + ILP is good for WCET, but MC is not, nor ILP alone. *Lecture Notes in Computer Science*, 2937, 2003.