Static Timing Analysis for Hard Real-Time Systems*

Reinhard Wilhelm, Sebastian Altmeyer, Claire Burguière,
Daniel Grund, Jörg Herter, Jan Reineke,
Björn Wachter, and Stephan Wilhelm

Saarland University, Saarbrücken, Germany

Abstract. Hard real-time systems have to satisfy strict timing constraints. To prove that these constraints are met, timing analyses aim to derive safe upper bounds on tasks' execution times. Processor components such as caches, out-of-order pipelines, and speculation cause a large variation of the execution time of instructions, which may induce a large variability of a task's execution time. The architectural platform also determines the precision and the complexity of timing analysis.

This paper provides an overview of our timing-analysis technique and in particular the methodological aspects of interest to the verification community.

1 Introduction

Hard real-time systems have to satisfy strict timing constraints. Traditionally, measurement has been used to show their satisfaction. However, the use of modern high-performance processors has created a severe problem. Processor components such as caches, out-of-order pipelines, and all kinds of speculation cause a large variability of the execution times of instructions, which induces a potentially high variability of whole programs' execution times. For individual instructions, the execution time may vary by a factor of 100 and more. The actual execution time depends on the architectural state in which the instruction is executed, i.e. the contents of the caches, the occupancy of the pipeline units, contention on the busses etc.

Different kinds of timing analyses are being used today [1]; measurement-based/hybrid [2,3,4] and static analysis [5] being the most prominent. Both methods compute estimates of the worst-case execution times for program fragments like basic blocks. If these estimates are correct, i.e. they are upper bounds on the worst-case execution time of the program fragment, they can be combined to obtain an upper bound on the worst-case execution time of the task.

* The research leading to these results has received funding from the following projects (in alphabetical order): the Deutsche Forschungsgemeinschaft in SFB/TR 14 *AVACS*, the German-Israeli Foundation (GIF) in the *Encasa* project, and the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement number 216008 (*Predator*).

G. Barthe and M. Hermenegildo (Eds.): VMCAI 2010, LNCS 5944, pp. 3–22, 2010.

While using similar methods in the combination of execution times of program fragments, the two methods take fundamentally different approaches to compute these estimates:

- Static analyses based on abstract models of the underlying hardware compute invariants about the set of all execution states at each program point under *all* possible initial states and inputs and derive upper bounds on the execution time of program fragments based on these invariants.
- Measurement executes each program fragment with a subset of the possible initial states and inputs. The maximum of the measured execution times is in general an underestimation of the worst-case execution time.

If the abstract hardware models are correct, static analysis computes safe upper bounds on the WCETs of program fragments and thus also of tasks. However, creating abstract hardware models is an error-prone and laborious process, especially if no precise specification of the hardware is available.

The advantage of measurement over static analysis is that it is more easily portable to new architectures, as it does not rely on such abstract models of the architecture. On the other hand, soundness of measurement-based approaches is hard to guarantee. Measurement would trivially be sound if all initial states and inputs would be covered. Due to their huge number this is usually not feasible. Instead, only a subset of the initial states and inputs can be considered in the measurements.

This paper provides an overview of our state-of-the-art timing-analysis approach. In Section 2, we describe the architecture and the component functionalities of our framework for static timing analysis.

Section 3 is devoted to several aspects of the memory hierarchy, in particular caches. Memory-system performance often dominates overall system performance. This makes cache analysis so important for timing analysis. The most relevant property is *predictability* [6,7]. This notion—a hot topic of current research—is fully clarified for caches [8,9]. A second notion, exemplified for caches, is the *relative competitiveness* of different cache architectures. They allow one to use the cache-analysis results of one cache architecture to predict cache performance for another one. A third property of cache architectures is their *sensitivity to the initial state*. Results show that some frequently used cache replacement-strategies are highly sensitive. This has severe consequences for measurement-based approaches to timing analysis. Missing one initial cache state in a non-exhaustive set of measurements may lead to dramatically wrong results.

Data-cache analysis would fail for programs allocating data in the heap since the addresses and therefore the mapping to cache sets would be statically unknown. We approach this problem in two different ways, firstly, by converting dynamic to static allocation and using a parametric timing analysis, and secondly, by allocating in a cache-aware way.

Pipelines are much more complex than caches and therefore more difficult to model. The analysis of their behavior needs much more effort than cache analysis since a huge search space has to be explored and no abstract domain with a compact representation of sets of pipeline states has been found so far. Symbolic data structures as used in model checking offer some potential to increase the efficiency. A novel symbolic approach to timing analysis has shown promising results [10]. We give a short overview in Section 4.

This article centers around static timing analysis. An extended description of our approach can be found in [11]. A comprehensive survey of timing-analysis approaches is given in [1].

1.1 The Architectural Challenge—and How to Cope with It

Hard real-time systems need guarantees expressed in terms of worst-case performance. However, the architectures on which the real-time programs are executed are optimized for average-case performance. Caches, pipelines, and all kinds of speculation are key features for improving average-case performance. Caches are used to bridge the gap between processor speed and the access time of main memory. Pipelines enable acceleration by overlapping the executions of different instructions. The consequence is that the execution time of individual instructions, and thus the contribution to the program's execution time can vary widely. The interval of execution times for one instruction is bounded by the execution times of the following two cases:

- The instruction goes "smoothly" through the pipeline; all loads hit the cache, no pipeline hazard happens, i.e. all operands are ready, no resource conflicts with other currently executing instructions exist.
- "Everything goes wrong", i.e. instruction and/or operand fetches miss the cache, resources needed by the instruction are occupied, etc.

We will call any increase in execution time during an instruction's execution a *timing accident* and the number of cycles by which it increases the *timing penalty* of this accident. Timing penalties for an instruction can add up to several hundred processor cycles. Whether the execution of an instruction encounters

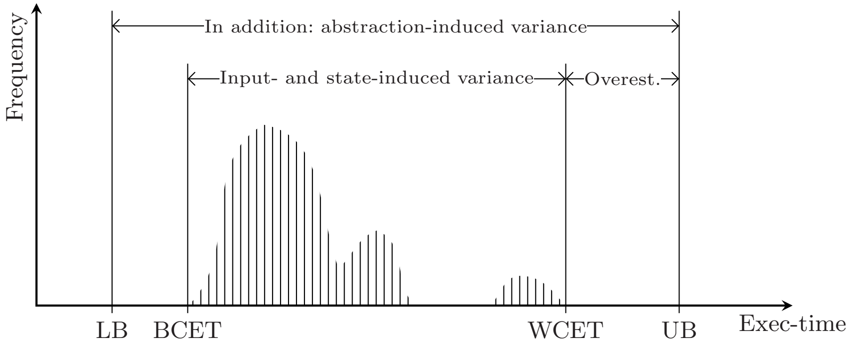


Fig. 1. Notions in Timing Analysis. Best-cast and worst-case execution time (BCET and WCET), and computed lower and upper bounds.

a timing accident depends on the architectural state, e.g. the contents of the cache(s), the occupancy of other resources, and thus on the execution history. It is therefore obvious that the attempt to predict or exclude timing accidents needs information about the execution history.

We use static analysis to compute invariants about the set of all possible architectural states at all program points. Indeed, due to abstraction, over-approximations of these sets are computed. They are used to derive safety properties of the kind: "A certain timing accident will not happen at this program point.". Such a safety property allows the timing-analysis tool to prove a tighter worst-case bound.

Some abstraction of the execution platform is necessary to make a timing analysis of the system feasible. These abstractions lose information, and thus are in part responsible for the gap between WCETs and upper bounds and between BCETs and lower bounds. How much is lost depends both on the methods used for timing analysis and on system properties, such as the hardware architecture and the analyzability of the software.

1.2 Timing Anomalies

Most powerful microprocessors have so-called *timing anomalies* [12]. Timing anomalies are contra-intuitive influences of the (local) execution time of one instruction on the (global) execution time of the whole program. Several processor features can interact in such a way that a locally faster execution of an instruction can lead to a globally longer execution time of the whole program. Hence, resolving uncertainty in the analysis by only assuming local worst-cases might be unsound.

One would assume that a cache miss is always the worst-case possibility for a memory access. However, the cache miss may prevent an expensive branch misprediction and, thus, globally be the better case. This was observed for the MCF 5307 [13,5]. Since the MCF 5307 has a unified cache and the fetch and execute pipelines are independent, the following can happen: A data access hitting in the cache is served directly from the cache. At the same time, the pipeline fetches another instruction block from main memory, performing branch prediction and replacing two lines of *data* in the cache. These may be reused later on and cause two misses. If the data access was a cache miss, the instruction fetch pipeline may not have fetched those two lines, because the execution pipeline may have resolved a misprediction before those lines were fetched.

The existence of timing anomalies forces the timing analysis to explore *all* successor states that cannot be excluded, not only the local worst-case ones. Besides the fact that timing penalties may partly mask each other out, timing anomalies are another reason why timing is *not* compositional.

2 A Timing Analysis Framework

Over roughly the last decade, a more or less standard architecture for timing-analysis tools has emerged. Figure 2 gives a general view of this architecture. First, one can distinguish four major building blocks:

- control-flow reconstruction
- static analyses for control and data flow
- micro-architectural analysis computing upper and lower bounds on the execution times of basic blocks
- global bounds analysis computing upper and lower bounds for the whole program

The following list presents the individual phases and describes their objectives and main challenges.

1. *Control-flow reconstruction* [14] takes a binary executable to be analyzed, reconstructs the program's control flow and transforms the program into a suitable intermediate representation. Problems encountered are dynamically computed control-flow successors, e.g. those stemming from switch statements, function pointers, etc.
2. *Value analysis* [15] computes an over-approximation of the set of possible values in registers and memory locations by an interval analysis and/or congruence analysis. The computed information is used for a precise data-cache analysis and in the subsequent control-flow analysis. Value analysis is the only one to use an abstraction of the processor's arithmetic. A subsequent pipeline analysis can therefore work with a simplified pipeline where the arithmetic units are removed. One is not interested in what is computed, but only in how long it will take.

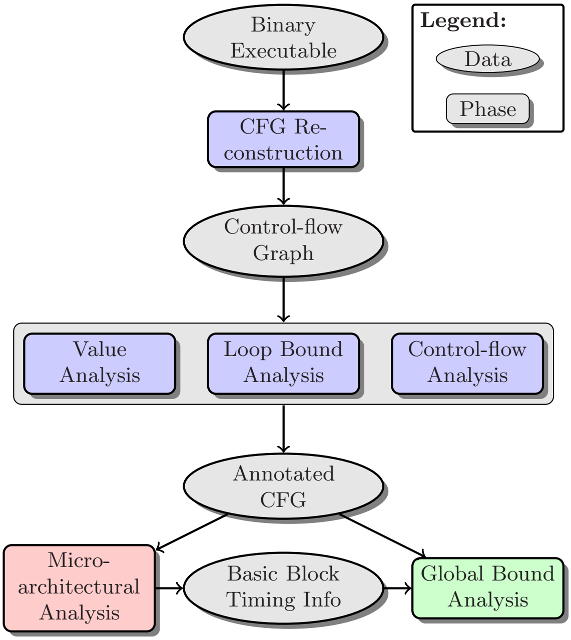


Fig. 2. Main components of a timing-analysis framework and their interaction

3. *Loop bound analysis* [16,17] identifies loops in the program and tries to determine bounds on the number of loop iterations; information indispensable to bound the execution time. Problems are the analysis of arithmetic on loop counters and loop exit conditions, as well as dependencies in nested loops.
4. *Control-flow analysis* [16,18] narrows down the set of possible paths through the program by eliminating infeasible paths or by determining correlations between the number of executions of different blocks using the results of value analysis. These constraints will tighten the obtained timing bounds.
5. *Micro-architectural analysis* [19,20,21] determines bounds on the execution time of basic blocks by performing an abstract interpretation of the program, combining analyses of the processor's pipeline, caches, and speculation. Static cache analyses determine safe approximations to the contents of caches at each program point. Pipeline analysis analyzes how instructions pass through the pipeline accounting for occupancy of shared resources like queues, functional units, etc.
6. *Global bounds analysis* [22,23] finally determines bounds on the execution times for the whole program by implicit path enumeration using an integer linear program (ILP). Bounds of the execution times of basic blocks are combined to compute longest paths through the program. The control flow is modeled by Kirchhoff's law. Loop bounds and infeasible paths are modeled by additional constraints. The target function weights each basic block with its time bound. A solution of the ILP maximizes the sum of those weights and corresponds to an upper bound on the execution times.

The commercially available tool `aiT` by AbsInt, cf. `http://www.absint.de/wcet.htm` implements this architecture. It is used in the aeronautics and automotive industries and has been successfully used to determine precise bounds on execution times of real-time programs [21,5,24,13]. The European Airworthiness Authorities have validated it for the certification of several avionics subsystems of the Airbus A380.

3 Cache Analysis

The goal of a static cache analysis is to statically predict the cache behavior of a program on a set of inputs with a possibly unknown initial cache state. As the cache behavior may vary from input to input and from one initial state to another, it may not be possible to safely classify each memory access in the program as a cache hit or a cache miss. A cache analysis is therefore forced to approximate the cache behavior in a conservative way if it shall be used to provide guarantees on the execution time of a task.

To obtain tight bounds on the execution time it is essential to use a precise cache analysis. Each excluded cache miss improves the provable upper bound on the worst-case execution time roughly by the cache miss penalty. Conversely, each guaranteed cache miss improves the provable lower bound on the best-case execution time.

WCET and BCET analyses need a classification of individual memory accesses in the program as cache hits or misses. For most architectures, it is not sufficient to determine upper and lower bounds on the number of misses for the execution of the entire program because caches interact with other architectural components such as pipelines. For instance, a cache reload may overlap with a pipeline stall. To precisely take such effects into account, a timing analysis needs to know where and when the cache misses happen.

One may compute *may* and *must* cache information in static cache analysis: *may* and *must* caches at a program point are upper and lower approximations, respectively, to the contents of all concrete caches that will occur whenever program execution reaches this program point. The *must* cache at a program point is a set of memory blocks that are definitely in each concrete cache at that point. The *may* cache is a set of memory blocks that may be in a concrete cache whenever program execution reaches that program point. We call the two analyses *may* and *must* cache analyses.

Must cache information is used to derive safe information about cache hits; in other words it is used to exclude the timing accident “cache miss”. The complement of the *may* cache information is used to safely predict cache misses.

3.1 Influence of the Cache Replacement Policy

Caches have a particularly strong influence on both the variation of execution times due to the initial hardware state and on the precision of static WCET analyses. A cache's behavior is controlled by its replacement policy. In [9], we investigate the influence of the cache replacement policy on

- the amount of *inherent uncertainty in static cache analysis*, i.e. cache misses that cannot be excluded statically but never happen during execution
- the *maximal variation in cache performance* due to the initial cache state
- the *construction of static cache analyses*, analyses that statically classify memory references as cache hits or misses

The following subsections explain the three problems in more detail and sketch our approaches and contributions.

Predictability Metrics—Limits on the Precision of Cache Analyses. Usually there is some uncertainty about the cache contents, i.e. the *may* and *must* caches do not coincide; there are memory blocks which can neither be guaranteed to be in the cache nor not to be in it. The greater the uncertainty in the *must* cache, the worse the upper bound on the WCET. Similarly, greater uncertainty in the *may* cache entails a less precise lower bound on the BCET.

There are several reasons for uncertainty about cache contents:

- Static cache analyses usually cannot make any assumptions about the initial cache contents. Cache contents on entry depend on previously executed tasks. Even assuming an empty cache may not be conservative [25].

- At control-flow joins, analysis information about different paths needs to be safely combined. Intuitively, one must take the intersection of the incoming *must* information and the union of the incoming *may* information. A memory block can only be in the *must* cache if it is in the *must* caches of all predecessor control-flow nodes, correspondingly for *may* caches.
- In data-cache analysis, the *value analysis* may not be able to exactly determine the address of a memory reference. Then the *cache analysis* must conservatively account for all possible addresses.
- Preempting tasks may change the cache state in an unpredictable way at preemption points [26].

Since information about the cache state may thus be unknown or lost, it is important to recover information quickly to be able to classify memory references safely as cache hits or misses. This is possible for most caches. However, the *speed* of this recovery greatly depends on the cache replacement policy. It influences how much uncertainty about cache hits and misses remains. Thus, the *speed of recovery* is an indicator of timing predictability.

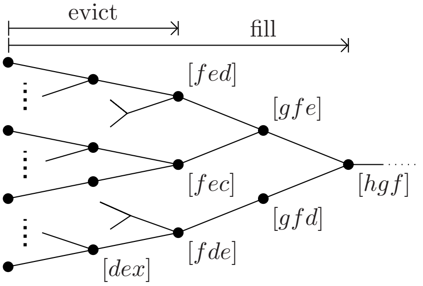


Fig. 3. Initially different cache sets converge when accessing a sequence $\langle a, b, c, d, e, f, g, h, \ldots\rangle$ of pairwise different memory blocks

The two metrics, *evict* and $fill$, indicate how quickly knowledge about cache hits and misses can be (re-)obtained under a particular replacement policy [8]. They mark a limit on the precision that *any* cache analysis can achieve, be it by abstract interpretation or any other sound method. Figure 3 illustrates the two metrics. *evict* tells us at which point we can safely predict that some memory blocks are no more in the cache, i.e. they are in the complement of may information. Any memory block not contained in the last *evict* accesses cannot be in the cache set. The greater *evict*, the longer it takes to gain may information. $fill$ accesses are required to converge to one completely determined cache set. At this point, complete may and must information is obtained, which allows to precisely classify each memory access as a hit or a miss. The two metrics mark a limit on *any* cache analysis: no analysis can infer any may information (complete must information) given an unknown cache-set state and less than *evict* ($fill$) memory accesses.

Under the two metrics, LRU is optimal, i.e. *may*- and *must*-information can be obtained in the least possible number of memory accesses. PLRU, MRU, and FIFO, perform considerably worse. Compared to an 8-way LRU, it takes more than twice as many accesses to regain complete *must*-information for equally-sized PLRU, MRU, and FIFO caches. As a consequence, it is *impossible* to construct cache analyses for PLRU, MRU, and FIFO that are as precise as known LRU analyses.

Relative Competitiveness of Replacement Policies. Developing cache analyses—analyses that statically determine whether a memory access associated with an instruction will always be a hit or a miss—is a difficult problem. Precise and efficient analyses have been developed for set-associative caches that employ the least-recently-used (LRU) replacement policy [21,27,28,29]. Other commonly used policies, like first-in-first-out (FIFO) or Pseudo-LRU(PLRU) are more difficult to analyze [8].

Relative competitive analyses yield upper (lower) bounds on the number of misses (hits) of a policy P relative to the number of misses (hits) of another policy Q. For example, a competitive analysis may find out that policy P will incur at most 30% more misses than policy Q and at most 20% less hits in the execution of any task.

The following approach determines safe bounds on the number of cache hits and misses by a task T under FIFO(k), PLRU(l)[1], or any another replacement policy [9]:

1. Determine competitiveness of the desired policy P relative to a policy Q for which a cache analysis exists, like LRU.
2. Perform cache analysis of task T for policy Q to obtain a cache-performance prediction, i.e. upper (lower) bounds on the number of misses (hits) by Q.
3. Calculate upper (lower) bounds on the number of misses (hits) for P using the cache analysis results for Q and the competitiveness results of P relative to Q.

Step 1 has to be performed only once for each pair of replacement policies.

A limitation of this approach is that it only produces upper (lower) bounds on the number of misses (hits) for the whole program execution. It does not reveal at which program points the misses (hits) will happen, something many timing analyses need. Relative competitiveness results can also be used to obtain sound *may* and *must* cache analyses, i.e. analyses that can classify individual accesses as hits or misses. Relative competitive ratios can be computed automatically for a pair of policies [30][2].

One of our results is that for any associativity k and any workload, FIFO(k) generates at least half the number of hits that LRU(k) generates. Another result is that *may* cache analyses for LRU can be safely used as *may* cache analyses for MRU and FIFO of other associativities.

Sensitivity of Replacement Policies. The sensitivity of a cache replacement policy expresses to what extent the initial state of the cache may influence the number of cache hits and misses during program execution [9]. Analysis results demonstrate that the initial state of the cache can have a strong impact on the number of cache hits and misses during program execution if FIFO, MRU, or PLRU replacement is used. A simple model of execution time demonstrates the impact of cache sensitivity on measured execution times. It shows that underestimating the number of misses as strongly as possible for FIFO, MRU,

[1] k and l denote the respective associativities of FIFO(k) and PLRU(l).

[2] See `http://rw4.cs.uni-sb.de/~reineke/relacs` for a corresponding applet.

$$\begin{aligned}
q_1 &= [\bot,\bot,\bot,\bot] \xrightarrow[\mathrm{M}]{a} [a,\bot,\bot,\bot] \xrightarrow[\mathrm{H}]{a} [a,\bot,\bot,\bot] \xrightarrow[\mathrm{M}]{b} [b,a,\bot,\bot] \xrightarrow[\mathrm{M}]{c} [c,b,a,\bot] = q_1' \\
q_2 &= [a,x,b,c] \xrightarrow[\mathrm{H}]{a} [a,x,b,c] \xrightarrow[\mathrm{H}]{a} [a,x,b,c] \xrightarrow[\mathrm{H}]{b} [a,x,b,c] \xrightarrow[\mathrm{H}]{c} [a,x,b,c] = q_2' \\
q_3 &= [x,y,z,a] \xrightarrow[\mathrm{H}]{a} [x,y,z,a] \xrightarrow[\mathrm{H}]{a} [x,y,z,a] \xrightarrow[\mathrm{M}]{b} [b,x,y,z] \xrightarrow[\mathrm{M}]{c} [c,b,x,y] = q_3' \\
q_4 &= [x,y,b,z] \xrightarrow[\mathrm{M}]{a} [a,x,y,b] \xrightarrow[\mathrm{H}]{a} [a,x,y,b] \xrightarrow[\mathrm{H}]{b} [a,x,y,b] \xrightarrow[\mathrm{M}]{c} [c,a,x,y] = q_4'
\end{aligned}$$

Fig. 4. Dependency of FIFO cache set contents on the initial state

and PLRU may yield worst-case-execution-time estimates that are dramatically wrong. Further analysis revealed that the "empty cache is worst-case initial state" assumption [2] is wrong for FIFO, MRU, and PLRU.

3.2 FIFO Cache Analysis

Precise and efficient analyses have been developed for the least-recently-used (LRU) replacement policy [21,27,28,29]. Generally, research in the field of embedded real-time systems assumes LRU replacement. In practice however, other policies like first-in first-out (FIFO) or pseudo-LRU (PLRU) are also commonly used. In [31], we discuss challenges in FIFO cache analysis. We identify a generic policy-independent framework for cache analysis that couples may- and must-analyses by means of domain cooperation. The main contribution is a more precise may-analysis for FIFO. It not only increases the number of predicted misses, but also—due to the domain cooperation—the number of predicted hits. We instantiate the framework with a canonical must-analysis and three different may-analyses, including the new one, and compare the resulting three analyses to the collecting semantics.

To see the difficulty inherent in FIFO, consider the examples in Figure 4. The access sequence $s = \langle a, a, b, c \rangle$ is carried out on different cache sets q_i of associativity 4. Although only 3 different memory blocks $\{a, b, c\}$ are accessed, some of the resulting cache sets q_i' do not contain all of the accessed blocks. In contrast, a k-way cache set with LRU replacement always consists of the k most-recently-used memory blocks, e.g. $\{a, b, c\}$ would be cached after carrying out s, independently of the initial state. This makes analysis of FIFO considerably more difficult than analysis of LRU.

To generalize, consider a FIFO cache set with unknown contents. After observing a memory access to a block a, trivial must-information is available: One knows that a must be cached, but the position of a within the cache set is unknown. For example the access to a could be a hit to the second position:

$$[?,?,?,?] \xrightarrow[hit]{a} [?,a,?,?] \xrightarrow[hit]{b} [?,a,?,b]$$

$$[?,?,?,?] \xrightarrow[hit]{a} [?,?,?,a] \xrightarrow[miss]{b} [b,?,?,?]$$

However, as in the second case, the access to a could also be a hit on the first-in (i.e., rightmost) position. Hence, a second access to a different block b may already evict the first accessed block a. Thus, without additional information about the accesses it is not possible to infer that two or more blocks are cached, i.e. one can only derive must information of poor quality.

However, there are means to gain more precise information: If one can classify the access to a as a miss for example, then the second access to a different block b cannot evict a because one knows that a was inserted at the last-in position.

$$[?, ?, ?, ?] \xrightarrow[miss]{a} [a, ?, ?, ?] \xrightarrow[miss]{b} [b, a, ?, ?]$$

On a more abstract level, what this actually means is that may-information can be used to obtain precise must-information. To do so however, one needs to realize information flow between may- and must-analyses. This gives rise to the policy-independent cache-analysis framework explained below that can couple different analyses to improve analysis precision.

Must- and May-analyses for FIFO. Here, we only describe the ideas behind the abstract domains and kindly refer the interested reader to [31] for details. The must-analysis borrows basic ideas from LRU-analysis [21]. For each memory block b, it infers an upper bound on the number of cache misses since the last insertion of b into the cache set. If the bound for b is smaller than the associativity, cache hits can be soundly predicted for b. Analogously, to predict cache misses, the may-analysis infers lower bounds on the number of cache misses to prove eviction. By distinguishing between hits and misses and taking into account the order in which they happen, we improve the may-analysis, thereby increasing the number of predicted cache misses. Through the cooperation of the two analyses in the generic framework, this also improves the precision of the must-analysis.

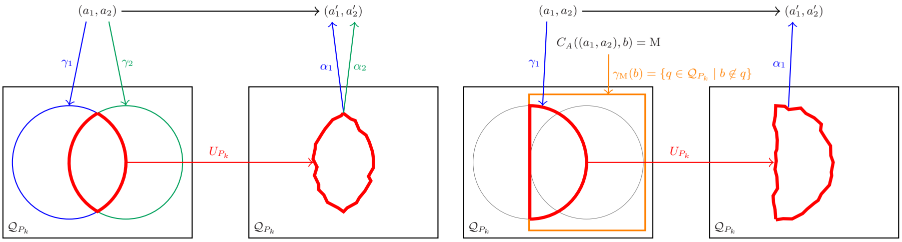
Cache Analysis Framework. As motivated above, for FIFO there needs to be some information flow between may- and must-analyses to obtain precise information. Indeed, this is not restricted to FIFO and can be generalized: This section presents a *policy-independent* cache analysis framework, in which any number of *independent* cache analyses can *cooperate*. The goal is to obtain information that is more precise than the best information obtained by any of the individual analyses. The only prerequisite is that the individual analyses implement a very small interface. Given correct analyses, the framework realizes the cooperation between these and guarantees correctness of the resulting analysis.

The framework constructs a cache analysis (A, C_A, U_A, J_A), with abstract domain A, classification function C_A, abstract transformer U_A, and join function J_A, given any number of cache analyses $(A_i, C_{A_i}, U_{A_i}, J_{A_i})$ for the same concrete cache set type $\mathcal{Q}_{P_k}$. The domain of the constructed analysis is the cartesian product

$$A := A_1 \times \ldots \times A_n$$

To classify a cache access to some memory block $b \in \mathcal{B}$, the classification function, $C_A : A \times \mathcal{B} \rightarrow \{\text{hit}, \text{miss}\}^\top$, combines the classifications of all individual analyses:

$$C_A((a_1, \ldots, a_n), b) := \prod_i C_{A_i}(a_i, b) \tag{1}$$



(a) Best abstract transformer would correspond to $\alpha_j(U_{P_k}(\bigcap_i \gamma_i(a_i), b))$ for all j.

(b) Abstract transformer using reduction by classification corresponds to $\alpha_i(U_{P_k}(\gamma_i(a_i) \cap \gamma_{C_A(a,b)}(b), b))$ for all i.

Fig. 5. Information flow between analyses by update reduction

Since each individual analysis is sound, these classifications cannot contradict each other, i.e. their meet ($\sqcap$) is always defined.

In abstract interpretation, the term *reduction* refers to the process of refining information encoded in a domain by other, external, information. For an example, consider value analysis using the *Interval*- and *Parity*-domain: Assume the interval domain infers $n \in [2, 4]$ and the parity domain provides $\text{isOdd}(n)$. Then, using the latter, one can *reduce* the interval to $n \in [3, 3]$.

However, in abstract domains for cache analysis, information expressible in one domain is not necessarily expressible in another one: The syntactical structure of constraints in domain A_1 does not allow to encode information provided by constraints of domain A_2. For example, a must-analysis maintains *upper* bounds on the number of cache misses while a may-analysis maintains *lower* bounds on that number. In such a case, a reduction on the abstract states would be ineffective.

Nonetheless, it is possible to use the information provided by other abstract domains to reduce the abstract transformers. First, consider the two extremes: On the one hand, the *independent* update of all A_i, and on the other hand a *best* abstract transformer. In an independent update of an A_i *no* information of the other domains is used. In a best abstract transformer, which is depicted in Figure 5(a), *all* information of the other domains is used: It would correspond to taking the intersection of all concretizations (sets of cache sets), updating them in the concrete, and then abstracting to the domains again. However, best abstract transformers counteract the wish to implement and prove correct individual domains *independently* and mostly are computationally expensive, anyway.

The update reduction of our framework lies in between these two extremes: The reduced abstract transformers are more precise than independent updates. And the information exchange is abstract enough such that it can be realized without knowledge about the participating domains, i.e. domains can be plugged in without changing the update functions of other domains. The update reduction of the framework uses the classification of the current access. Figure 5 shows this at hand of the domain A_1. Assume that some domain A_i can classify the access to block b as a miss, e.g. $C_{A_2}(a_2, b) = \mathrm{M}$. Then the overall classification, which depends on all individual classifications, will be $C_A((a_1, a_2), b) = \mathrm{M}$, too. With this information, one can further restrict the concretization $\gamma_{A_1}(a_1)$ to cache sets that additionally do not contain the accessed block $\{q \in \mathcal{Q}_{P_k} \mid b \notin q\}$. Using this additional information, one can define abstract transformers that are more precise than independent ones.

In an implementation, the update reduction amounts to refine the update functions of each domain by an additional parameter to pass the classification of the current access. Hence, the update function $U_A : A \times \mathcal{B} \rightarrow A$ is defined as:

$$U_A((a_1, \ldots, a_n), b) := \left(U_{A_1}(a_1, b, cl), \ldots, U_{A_n}(a_n, b, cl)\right), \tag{2}$$

where $cl := C_A((a_1, \ldots, a_n), b)$.

Finally, the join function J_A is simply defined component-wise.

3.3 Context Switch Costs

Previous timing analyses assume tasks running to completion, i.e. assuming non-preemptive execution. Some task sets, however, are only schedulable preemptively. For such systems, we also need to bound the context-switch costs in addition to the WCET. In case of preemption, cache memories may suffer interferences between memory accesses of the preempted and of the preempting task. These interferences lead to some additional reloads, which are referred to as cache-related preemption delay (CRPD). This CRPD constitutes the major part of the context switch costs.

Upper bounds on the CRPD are usually computed using the concept of useful cache blocks (UCB). Memory blocks are considered useful at a program point if they may be in cache before the program point and may be reused after it. When a preemption occurs at that point the number of additional cache-misses is bounded by the number of useful cache blocks. However, some cache accesses are taken into account as misses as part of the WCET bound anyway. These accesses do not have to be accounted for a second time as part of the CRPD bound [26].

A memory block m is called a definitely-cached UCB (DC-UCB) at program point P if (a) m must be cached before the preemption point and (b) m may be reused at program point Q, which may be reached from P, and must be cached along the path to its reuse. Using the notion of definitely-cached UCB, one computes the number of additional cache misses due to preemption that are not already taken into account as a miss by the timing analysis. This number does not bound the CRPD, but the part of the CRPD that is not already included in the WCET bound. Hence, the global bound on WCET+CRPD can be significantly improved.

The DC-UCB analysis uses information computed by a preceding cache analysis. In the following, we denote instruction j of basic block i as B_i^j and use $\mathit{Access}(B_i^j)$ to denote the memory block accessed by instruction B_i^j.

To determine the set of definitely-cached UCBs, we use a backward program analysis over the control flow graph. A memory block m is added to the set of

DC-UCBs of instruction B_i^j, if m is element of the must cache at B_i^j (computed by a preceding cache analysis) and if instruction B_i^j accesses m. The domain of our analysis is the powerset domain of the set of memory blocks M: $\mathbb{D} = 2^M$ The following two equations determine the data-flow value before (DC-UCB$_{in}$) and after (DC-UCB$_{out}$) instruction B_i^j:

$$\text{DC-UCB}_{in}(B_i^j) = gen(B_i^j) \cup (\text{DC-UCB}_{out}(B_i^j) \setminus kill(B_i^j)) \tag{3}$$

$$\text{DC-UCB}_{out}(B_i^j) = \bigcup_{successor B_k^l} \text{DC-UCB}_{in}(B_k^l) \tag{4}$$

where the gen/kill sets are defined as follows:

$$gen(B_i^j) = \begin{cases} \{Access(B_i^j)\} & \text{if } Access(B_i^j) \in Must\_Cache(B_i^j) \\ \emptyset & \text{otherwise} \end{cases} \tag{5}$$

$$kill(B_i^j) = M \setminus Must\_Cache(B_i^j) \tag{6}$$

Equation (4) combines the flow information of all successors of instruction B_i^j. Equation (3) represents the update of the flow information due to the execution of the instruction. First, all memory blocks not contained in the must cache at B_i^j are removed from the set of DC-UCBs (6)—only a memory block that is element of the must cache all along the way to its reuse is considered useful by our definition. Then, the accessed memory block of instruction B_i^j is added in case it is contained in the must cache at the instruction (5).

Using these equations, the set of UCBs can be computed via fixed-point iteration (see [15]). The initial values at instruction B_i^j are defined by DC-UCB$_{in}(B_i^j)$ $= gen(B_i^j)$ and DC-UCB$_{out}(B_i^j) = \emptyset$.

The analysis obtains a set of memory blocks at each program point P access that might cause an additional miss upon access in case of preemption at P. The program point P with the largest DC-UCB set determines an upper-bound on the number of additional misses for the whole task. In contrast to the former UCB analysis, the DC-UCB analysis only takes those misses into account that are not part of the WCET bound. Evaluation shows that up to 80% of the accesses to a UCB were also considered to be misses in the WCET analysis. The DC-UCB analysis omits these UCBs. Hence, the analysis derives much better bounds on the CRPD when used in the context of timing analysis.

3.4 Heap-Allocating Programs

Static timing analyses rely on high cache predictability in order to achieve precise bounds on a program's execution time. Such analyses, however, fail to cope with programs using dynamic memory allocation. This is due to the unpredictability of the cache behavior introduced by the dynamic memory allocators. Using standard allocators, the cache sets to which a newly allocated memory block is mapped to are statically unknown. This does not only prohibit a cache analysis to derive hits or misses for accesses to dynamically allocated objects. It also forces such analyses to conservatively treat an access to a dynamically allocated block as an access to all cache sets. In turn, information about the cache derived from an access sequence to statically allocated objects may easily be lost. Allocators normally traverse some internal structure of free memory blocks in order to find a suitable block to satisfy an allocation request or reinsert newly deallocated memory blocks. These statically unpredictable traversals have the same negative effect on static cache analyses. Additionally, the response times of allocators can in general not be tightly bounded.

We investigate two approaches to enable precise worst-case execution time analysis for programs that use dynamic memory allocation.

The first approach automatically transforms the dynamic memory allocation into a static allocation with comparable memory consumption [32]. Hence, we try to preserve the main advantage of dynamic memory allocation, namely the reduction of memory consumption achieved by reusing deallocated memory blocks for subsequent allocation requests. Ending up with a static allocation allows for using existing techniques for timing analyses. However, the techniques for transforming dynamic to static allocation as presented in [32] have limitations. In particular, the number and sizes of dynamically allocated blocks need to be statically known. Although this might be reasonable in the hard real-time setting, ongoing research addresses this problem by investigating a parametric approach to automatically precompute memory addresses for otherwise dynamically allocated memory.

The second approach replaces the unpredictable dynamic memory allocator by a predictable dynamic memory allocator [33]. Our predictable memory allocator takes an additional—possibly automatically generated—argument specifying the cache set newly allocated memory shall be mapped to. It further gives guarantees on the number of cache lines per cache set that may be *touched* during (de)allocation. It also features constant response times by managing free blocks in multi-layered segregated lists.

Both approaches rely on precise information about the dynamically allocated heap objects and data structures arising during program execution. This information could be obtained by shape analysis [34] or data structure analysis. However, these analyses are not allocation-site aware, i.e. they only know about the shape of a data structure and are ignorant of the allocation requests that created the nodes of that structure. If we want to modify allocation requests, either by adding an additional cache set argument to a call to malloc or by replacing malloc by some function returning a sequence of static addresses, we rely on this missing information. Our current approach extends the shape analysis framework via three-valued logic by adding information about allocation sites to the logical representatives of heap-allocated objects.

4 Symbolic Representation of Pipeline Domains

Microarchitectural analysis explores execution traces of the program with respect to a pipeline model. In the pipeline model, information about register values is needed to determine addresses of memory accesses and information about cache content is needed to predict cache hits and misses. To make the analysis computationally feasible, abstractions of register and cache content of the processor are used. These abstractions may lose information.

Value analysis is invoked prior to microarchitectural analysis. It computes information about register content, which is later on used in microarchitectural analysis. For example, for a specific load instruction, value analysis computes a range of the possible memory addresses that contains the possible values for *all* executions that reach the load instruction. Microarchitectural analysis may, on the other hand, distinguish different traces ending at the load instruction. However, it uses the less specific approximation of register content from value analysis and may thus be unable to classify the address.

Further, instead of a more precise and expensive set representation of values, abstract domains like intervals and congruences are used in value analysis. This incurs additional loss of information. Similarly, cache analysis employs abstract domains which also sacrifice precision for efficiency.

Thus, at the level of the pipeline model, the inevitable use of abstraction incurs uncertainty about memory accesses and cache content. Furthermore, program inputs are not statically known. The (abstract) pipeline model has to cope with this lack of information by offering non-deterministic choices. Existence of timing anomalies forces the pipeline analysis to exhaustively explore all of them. In certain cases, state explosion can make explicit enumeration of states infeasible due to memory and computation time constraints [20].

We address the state explosion problem in static timing analysis by storing and manipulating pipeline states in a more efficient data structure based on Ordered Binary Decision Diagrams (OBDDs) [35]. Our work is inspired by BDD-based symbolic model checking [36]. Symbolic model checking has been successfully applied to components of processors. Its success sparked a general interest in symbolic representations.

4.1 Symbolic Domain and Analysis

A pipeline model can be regarded as a large, non-deterministic finite state machine (FSM). Pipeline analysis determines sets of reachable pipeline states by a fixed-point iteration over the control-flow graph, which involves the computation of abstract execution traces on the basic block level. WCET bounds for basic blocks are derived from the lengths of their execution traces.

Sets of pipeline states as well as the transition relation of the model can be represented implicitly using BDDs. Execution traces are computed by repeated application of a symbolic image operator to an incoming set of pipeline states.

We account for required program information by translating them into symbolic relations that restrict the non-deterministic choices of the pipeline model.

The resulting symbolic state traversal proceeds in breadth-first search order where one traversal step corresponds to a particular execution cycle of the processor. Savings in memory consumption and running time, compared to explicit-state analysis, result from the more efficient representation, in particular for large sets of states with many redundancies, and from completely avoiding the explicit enumeration of states.

4.2 Optimizations and Performance

To arrive at an efficient symbolic analysis that scales to pipeline models of real-life processors and industrial-size programs, we incorporate well-known optimizations from symbolic model checking, e.g. the image computation methods of [37], and novel domain-specific optimizations that leverage properties of the processor and the program. The processor-specific optimizations follow the general pattern of

- reducing representation size of components by omitting information that is not timing-relevant and
- statically precomputing information.

For example, the prefetch buffer of the Infineon TriCore processor uses 16 bytes in hardware, while the timing-relevant information can be stored in only 16 bits. For the same processor, conditions for pipeline stalls in case of unresolved data dependencies can be precomputed by a data flow analysis. The symbolic representation then requires only one state bit per pipeline to encode such stalls.

Properties of the analyzed program are exploited to achieve an efficient handling of the many 32 bit instruction and data addresses used by pipeline models. The optimizations are based on two observations:

- Each program typically uses only a small fraction of the address space.
- The computation of execution traces for a single basic block requires only a bounded amount of information about neighbouring blocks.

Based on the first observation, we compactly enumerate all addresses used in the program and then encode these addresses using a number of state bits logarithmic in the size of the set of used addresses. This significantly reduces the required number of state bits.

However, the size of the symbolic representation still depends on the size of the analyzed program. This dependence can be eliminated using the second observation. For the symbolic computation of abstract execution traces we enumerate only the addresses within range of the current basic block. The resulting incompatible address encoding in pipeline states of different basic blocks can be translated during the fixed-point iteration using symbolic image computation.

We enhance the existing framework for static timing analysis with a symbolic representation of abstract pipeline models. Our prototype implementation is integrated into the commercial WCET analysis tool `aiT` and employs the model of a real-life processor, the Infineon TriCore. The model was developed and tested within `aiT`. This enables a meaningful performance comparison between the two implementations, which produce the same analysis results. Experiments with a set of industrial benchmarks show that the symbolic domain significantly improves the scalability of the analysis [10].

5 Conclusion and Ongoing Work

Computer architects have, for a long time, optimized processor architectures for average-case performance. This article has given an overview of the problems created by ignoring the needs of embedded systems, which often need guarantees for their worst-case performance. Formal methods have been described for the derivation of timing guarantees. Two architectural components have received a detailed treatment, caches and pipelines. Caches have nice abstractions, i.e. compact abstract domains and efficient abstract update functions. Static analysis of the cache behavior by abstract interpretation is therefore quite fast. Pipelines seem to require powerset domains with a huge state space. It has been described how to use symbolic representations popular in model checking to compactly represent sets of pipeline states. What is still missing is the interaction between the abstract-interpretation-based cache analysis and the BDD-based pipeline analysis. The tools (and the communities behind them) don't talk to each other.

Several important notions, e.g. predictability, sensitivity, and relative competitiveness have been clarified. Similar notions have to be found for non-cache like architecture components. Future architectures for use in safety-critical and time-critical applications should be designed under the design goals, high predictability of the timing behavior and low sensitivity against small changes of the execution state.

References

1. Wilhelm, R., et al.: The worst-case execution-time problem—overview of methods and survey of tools. Trans. on Embedded Computing Sys. 7(3), 1–53 (2008)
2. Petters, S.M.: Worst-Case Execution-Time Estimation for Advanced Processor Architectures. PhD thesis, Technische Universität München, Munich, Germany (2002)
3. Bernat, G., Colin, A., Petters, S.M.: WCET analysis of probabilistic hard real-time systems. In: Proceedings of the 23rd IEEE Real-Time Systems Symposium, Washington, DC, USA, p. 279. IEEE Computer Society, Los Alamitos (2002)
4. Wenzel, I.: Measurement-Based Timing Analysis of Superscalar Processors. PhD thesis, Technische Universität Wien, Vienna, Austria (2006)
5. Ferdinand, C., Heckmann, R., Langenbach, M., Martin, F., Schmidt, M., Theiling, H., Thesing, S., Wilhelm, R.: Reliable and precise WCET determination for a real-life processor. In: Henzinger, T.A., Kirsch, C.M. (eds.) EMSOFT 2001. LNCS, vol. 2211, pp. 469–485. Springer, Heidelberg (2001)
6. Thiele, L., Wilhelm, R.: Design for timing predictability. Real-Time Sys. 28, 157–177 (2004)
7. Wilhelm, R., Grund, D., Reineke, J., Schlickling, M., Pister, M., Ferdinand, C.: Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. IEEE Transactions on CAD of Integrated Circuits and Systems 28(7), 966–978 (2009)
8. Reineke, J., Grund, D., Berg, C., Wilhelm, R.: Timing predictability of cache replacement policies. Real-Time Sys. 37(2), 99–122 (2007)
9. Reineke, J.: Caches in WCET Analysis. PhD thesis, Saarland University, Saarbrücken, Germany (2008)

10. Wilhelm, S., Wachter, B.: Symbolic state traversal for WCET analysis. In: International Conference on Embedded Software, pp. 137–146 (2009)
11. Wilhelm, R.: Determining bounds on execution times. In: Zurawski, R. (ed.) Handbook on Embedded Systems, pp. 14–23. CRC Press, Boca Raton (2005)
12. Reineke, J., Wachter, B., Thesing, S., Wilhelm, R., Polian, I., Eisinger, J., Becker, B.: A definition and classification of timing anomalies. In: Proceedings of 6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis (2006)
13. Heckmann, R., Langenbach, M., Thesing, S., Wilhelm, R.: The influence of processor architecture on the design and the results of WCET tools. Real-Time Sys. 91(7), 1038–1054 (2003)
14. Theiling, H.: Control-Flow Graphs For Real-Time Systems Analysis. PhD thesis, Saarland University, Saarbrücken, Germany (2002)
15. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 238–252. ACM Press, New York (1977)
16. Ermedahl, A., Gustafsson, J.: Deriving annotations for tight calculation of execution time. In: Lengauer, C., Griebl, M., Gorlatch, S. (eds.) Euro-Par 1997. LNCS, vol. 1300, pp. 1298–1307. Springer, Heidelberg (1997)
17. Healy, C., Sjödin, M., Rustagi, V., Whalley, D., van Engelen, R.: Supporting timing analysis by automatic bounding of loop iterations. Real-Time Sys., 129–156 (2000)
18. Stein, I., Martin, F.: Analysis of path exclusion at the machine code level. In: Proceedings of the 7th Intl. Workshop on Worst-Case Execution-Time Analysis (2007)
19. Engblom, J.: Processor Pipelines and Static Worst-Case Execution Time Analysis. PhD thesis, Dept. of Information Technology, Uppsala University (2002)
20. Thesing, S.: Safe and Precise WCET Determinations by Abstract Interpretation of Pipeline Models. PhD thesis, Saarland University, Saarbrücken, Germany (2004)
21. Ferdinand, C., Wilhelm, R.: Efficient and precise cache behavior prediction for real-time systems. Real-Time Sys. 17(2-3), 131–181 (1999)
22. Li, Y.T.S., Malik, S.: Performance analysis of embedded software using implicit path enumeration. In: Proceedings of the 32nd ACM/IEEE Design Automation Conference, pp. 456–461 (1995)
23. Theiling, H.: ILP-based interprocedural path analysis. In: Sangiovanni-Vincentelli, A.L., Sifakis, J. (eds.) EMSOFT 2002. LNCS, vol. 2491, pp. 349–363. Springer, Heidelberg (2002)
24. Thesing, S., Souyris, J., Heckmann, R., Randimbivololona, F., Langenbach, M., Wilhelm, R., Ferdinand, C.: An abstract interpretation-based timing validation of hard real-time avionics software systems. In: Proceedings of the 2003 Intl. Conference on Dependable Systems and Networks, pp. 625–632. IEEE Computer Society, Los Alamitos (2003)
25. Berg, C.: PLRU cache domino effects. In: Mueller, F. (ed.) 6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany (2006)
26. Altmeyer, S., Burguière, C.: A new notion of useful cache block to improve the bounds of cache-related preemption delay. In: Proceedings of the 21st Euromicro Conference on Real-Time Systems, pp. 109–118. IEEE Computer Society Press, Los Alamitos (2009)

27. White, R.T., Healy, C.A., Whalley, D.B., Mueller, F., Harmon, M.G.: Timing analysis for data caches and set-associative caches. In: Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium, Washington, DC, USA, p. 192. IEEE Computer Society, Los Alamitos (1997)
28. Ghosh, S., Martonosi, M., Malik, S.: Precise miss analysis for program transformations with caches of arbitrary associativity. In: Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 228–239 (1998)
29. Chatterjee, S., Parker, E., Hanlon, P.J., Lebeck, A.R.: Exact analysis of the cache behavior of nested loops. In: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, pp. 286–297. ACM Press, New York (2001)
30. Reineke, J., Grund, D.: Relative competitive analysis of cache replacement policies. In: Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, pp. 51–60. ACM, New York (2008)
31. Grund, D., Reineke, J.: Abstract interpretation of FIFO replacement. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 120–136. Springer, Heidelberg (2009)
32. Herter, J., Reineke, J.: Making dynamic memory allocation static to support WCET analyses. In: Proceedings of 9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis (2009)
33. Herter, J., Reineke, J., Wilhelm, R.: CAMA: Cache-aware memory allocation for WCET analysis. In: Caccamo, M. (ed.) Proceedings Work-In-Progress Session of the 20th Euromicro Conference on Real-Time Systems, pp. 24–27 (2008)
34. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. Trans. on Programming Languages and Sys. 24(3), 217–298 (2002)
35. Bryant, R.: Graph based algorithms for boolean function manipulation. IEEE Transactions on Computers (1986)
36. Burch, J., Clarke, E., McMillan, K., Dill, D., Hwang, J.: Symbolic model checking: 10^{20} states and beyond. In: Proceedings of the 5th Annual Symposium on Logic in Computer Science. IEEE Comp. Soc. Press, Los Alamitos (1990)
37. Ranjan, R., Aziz, A., Brayton, R., Plessier, B., Pixley, C.: Efficient BDD Algorithms for FSM Synthesis and Verification. In: Proceedings of IEEE/ACM International Workshop on Logic Synthesis, Lake Tahoe, USA (1995)