# Scope-based Method Cache Analysis*

**Benedikt Huber[1], Stefan Hepp[2], and Martin Schoeberl[3]**

1 **Institute of Computer Engineering**
**Vienna University of Technology**
benedikt@vmars.tuwien.ac.at
2 **Institute of Computer Languages**
**Vienna University of Technology**
hepp@complang.tuwien.ac.at
3 **Department of Applied Mathematics and Computer Science**
**Technical University of Denmark**
masca@dtu.dk

## Abstract

The quest for time-predictable systems has led to the exploration of new hardware architectures that simplify analysis and reasoning in the temporal domain, while still providing competitive performance. For the instruction memory, the method cache is a conceptually attractive solution, as it requests memory transfers at well-defined instructions only. In this article, we present a new cache analysis framework that generalizes and improves work on cache persistence analysis. The analysis demonstrates that a global view on the cache behavior permits the precise analyses of caches which are hard to analyze by inspecting cache state locally.

**1998 ACM Subject Classification** B.8.2 Performance Analysis and Design Aids

**Keywords and phrases** Real-Time Systems, Cache Analysis, Time-predictable Computer Architecture

**Digital Object Identifier** 10.4230/OASIcs.WCET.2014.42

## 1 Introduction

In this paper, we are concerned with instruction cache architectures for real-time systems, and in particular with their analysis. For time-predictable architectures, we expect that it is possible to compute precise worst-case execution time (WCET) bounds for a sequence of instructions by only considering the number of cache misses, instead of the actual cache state at each instruction. Time-compositional architectures, such as Patmos [17], that enjoy this property are not just simpler to analyze, they also facilitate a larger class of timing analysis techniques and reduce the problem of interfering components, such as data and instruction caches. For non time-compositional architectures, global analyses like ours are still applicable, but need to be combined with local classification using prediction graphs [1].

The method cache was devised as an alternative to set-associative instruction caches for time-predictable architectures [16]. In contrast to traditional instruction caches, it does not manage individual cache lines, but holds entire blocks of code of variable size, e.g., an entire function. Conceptually, the advantage of the method cache is that only few, compiler-controlled instructions may cause a cache miss. This can be exploited to better

* This work was partially funded under the European Union's 7th Framework Programme under grant agreement no. 288008: Time-predictable Multi-Core Architecture for Embedded Systems (T-CREST).


14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014).
Editor: Heiko Falk; pp. 42–51

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

control instruction-memory related latencies, to avoid interferences between instruction and data cache competing for main memory access, and to optimize instruction cache usage. The performance of the method cache and the quality of worst-case guarantees for this cache, however, strongly depend on the compiler and the availability of a precise cache analysis.

In this article, we show that with a global view on the cache behavior, it is possible to effectively analyze instruction caches using the first-in-first-out (FIFO) replacement policy. We therefore invalidate previous assumptions that the method cache, which uses a FIFO replacement strategy, is intrinsically hard to analyze.

The key contribution of this article is a novel cache analysis that is applicable to various instruction cache architectures. To the best of our knowledge, this is the first scalable and precise analysis technique for method caches with variable code block size and FIFO replacement. Our cache analysis is also applicable to standard set-associative caches using either FIFO or the least-recently-used (LRU) replacement strategy, and is thus applicable to a wide range of target platforms.

The paper is organized as follows: Section 2 provides background information on the method cache and its variants. Section 3 presents our scope-based cache analysis, which is evaluated in Section 4. Section 5 discusses related work, Section 6 concludes the paper.

## 2 Method Cache

In contrast to set-associative caches, the method cache stores whole blocks of code of variable size [16, 2], Cache entries are allocated and evicted only at certain instructions such as call and return instructions, and are in general considerably larger than a cache line of a set-associative cache.

This design brings several advantages. It requires less memory for cache tags than a set-associative cache, there are no interferences with the data cache to be considered, and the cache may use more efficient burst transfers to load the program code into the cache. Furthermore, since cache lookup and replacement only have to be performed at call and return instructions, those cache management operations are not on the critical path of the processor pipeline and could be multi-cycle operations. Finally, the cache analysis only needs to consider a few instructions such as call and return, as all other instructions are guaranteed hits and do not change the cache state.

However, the application code must be partitioned into code blocks at compile time. If the code consists only of small blocks, the method cache may evict entries due to its limited associativity. Large code blocks containing control flow or call sites might be evicted before all instructions in that block have been executed. The compiler must therefore partition the control flow graph into code blocks so that the cache performance is not degraded either by the cache's associativity or by loading unused code into the cache.

A method cache can be implemented in a multitude of ways: *Fixed block method cache*: The cache is organized into blocks of fixed size. Each function is allocated in exactly one cache block. While this organization has the highest fragmentation, it allows for a LRU replacement policy. *Variable block method cache*: Like the fixed-block method cache, the cache is organized in blocks of fixed size. However, a function can be allocated to multiple blocks. This leads to a lower fragmentation, but a LRU policy is complex to implement in hardware. Thus, a FIFO replacement policy is usually employed. *Variable sized method cache*: This variant abandons the internal organization in blocks, thus eliminating internal fragmentation.

```
void m() {
  access(m) /* miss if m is not cached */
  access(a);
  access(m) /* miss if a evicted m */
  access(b);
  access(m) /* miss if b evicted m */
  access(c);
  access(m) /* miss if c evicted m */
}
```

**Figure 1** Limits of CHMC: FIFO cache analysis (4-way)

## 3 Scope-Based Cache Analysis

The two cache analysis techniques that have been extensively studied and implemented in industrial WCET analysis tools are cache hit-miss classification (CHMC) and persistence analysis. The former uses abstract interpretation to compute the set of possible cache states at each instruction in a virtually unrolled and inlined program model [18]. Subsequently, each access to memory is either classified as cache hit, as cache miss, or unknown. This information is finally taken into account when analyzing the timing behavior of the pipeline.

As discussed in [1], if the access to a memory block depends on an unpredictable condition, the CHMC technique fails to classify corresponding cache accesses. CHMC analysis is also less effective for FIFO caches, even though a precise cache state abstraction has been developed [3, 4]. In the example in Figure 1, assuming a cache with associativity four, it is not possible to classify any of the individual accesses to `m` as cache hit. This is because any of the accesses to `m` might be a cache miss, depending on the initial state, although there will be at most one cache miss for `m` in total.

Persistence analysis [1, 9] was conceived to improve the precision of the analysis of LRU caches, and overcome the limitation of CHMC concerning conditional cache accesses. The idea is to classify whether an instruction might suffer a cache miss the first time it is executed, but will hit the cache on subsequent accesses. If this is the case, the cache access is said to be (globally) persistent. Because a cache access might be persistent for the execution of one function, but not the whole program or task, a more useful notion is that of local persistence. An access is said to be persistent with respect to a scope (e.g., the execution of a function), if every time the scope is executed, the access will be a cache miss the first time only. The fact that a persistent cache block will be loaded only on the first access during each execution of the scope is subsequently taken into account during WCET calculation.

Persistence analysis as defined in the literature improves the analysis of LRU caches, but is not applicable to caches using the FIFO replacement policy. Consider again the example in Figure 1: although at most one of the accesses to $m$ is a cache miss, it need not be the first one. Therefore it is necessary to generalize the concept of persistence analysis (*first-miss*) to an analysis that accounts for misses that may occur *at most once*, although not necessarily at the first access. Moreover, in order to obtain a precise analysis, we found that it is necessary to review and refine the concept of persistence scopes. Support for scopes that include some but not all accesses to a memory block (e.g. because of shared library routines) and for scopes that do not correspond to functions or loops (e.g. single-entry regions) are crucial for efficient scope-based analysis of FIFO caches, but have not been considered in previous work.

## 3.1 Scope-based Cache Analysis Framework

The following discussion considers the general form of the scope-based cache analysis that is applicable for standard set-associative instruction caches and also for different variants of the method cache. We will use the term *memory block* to either denote a block of memory associated with a cache line, or a code block in context of the method cache.

The goal of the scope-based cache analysis is to compute a set of scopes $\mathcal{S}(B)$ for each memory block $B$, such that (1) every access to the memory block is in at least one scope $S \in \mathcal{S}(B)$ (2) during the execution of any scope $S \in \mathcal{S}(B)$, at most one access to $B$ will be a cache miss. If those two conditions are met, the sum of the execution frequencies of all scopes in $\mathcal{S}(B)$ is an upper bound for the number of cache misses for the memory block $B$.

In summary, these are the constituents of our analysis:

Scope Graph: The scope graph used by our analysis is an acyclic, hierarchical control-flow representation. Each node in the scope graph represents either a function, a loop or a callsite. The scope graph data structure is essential to ensure the performance of our cache analysis.

Conflict Detection: The analysis builds on a decision procedure that determines whether a memory block needs to be loaded at most once during the execution of a scope. In this case, we say the block is conflict-free with respect to the scope.

Computation of Conflict-Free Scopes: For each memory block, the analysis computes a set of conflict-free scopes. We not only consider functions and loops, but dynamically compute single-entry regions that are conflict-free scopes. The analysis traverses the acyclic scope graph bottom-up, and reuses results from nested scopes to ensure good performance.

IPET model: In the IPET model, we introduce *cache miss* variables, that represent the frequency of cache misses at a certain instruction. Their sum corresponds to the total number of cache misses for one memory block. The total number of cache misses of a memory block is in turn bounded by the frequency of the conflict-free scopes associated with the block.

## 3.2 Scope Graph

A scope graph is a hierarchical representation of the program's control-flow. Each node represents a set of instruction sequences that correspond to one execution of the program fragment represented by the scope. The root node represents all executions of the program to be analyzed, and each of the instruction sequences represented by a child node is included in one of instruction sequences represented by the parent node. The kind of scope graph used in our analysis is *acyclic*; this is essential to permit a bottom-up analysis. The nodes in a scope graph either correspond to functions, loops or callsites. Scopes that correspond to single-entry regions are not represented in the scope graph, but formed dynamically during analysis (see Section 3.3). Nodes in the scope graph are in turn associated with access graphs that model cache accesses within the scope.

Access graphs are acyclic flow graphs and consist of five different kinds of nodes: The *entry node* and *exit node* of an access graph represent the start and end of any cache access sequence associated with the scope, and *control-flow nodes* represent the beginning of a basic block. Cache accesses that are local to the scope are represented by *memory access nodes*. *Subscope nodes* correspond to the execution of a subscope that is a child in the scope graph. Finally, *back-edge nodes* model transfer to the scopes's entry node, for example to model the back edge of a loop.

**Algorithm 1** Scope Set Computation

```
 1: procedure CacheScopeAnalysis(ScopeGraph G)
 2:     for all memory blocks B do
 3:         S[B] ← {}
 4:     end for
 5:     for all nodes N ∈ bottom-up–traversal(G) do
 6:         for all memory blocks B accessed in N do
 7:             if IsConflictingScope(B, N) or
                   N is root node then
 8:                 S[B] ← S[B] ∪ CollectScopes(B, N)
 9:             end if
10:         end for
11:     end for
12:     return S
13: end procedure
```

The construction of the scope graph starts at the scope node that represents the program or task to be analyzed, and processes every function exactly once. First, we compute the strongly connected components (SCC) of the function's CFG. Each trivial SCC corresponds to a basic block, and is replaced by a control-flow node and a sequence of zero or more access graph nodes representing memory accesses and function calls. Non-trivial SCCs are replaced by a subscope node representing the corresponding loop, which is processed in turn.

The access graph of a loop is constructed in a similar way. First, we replace back edges by edges to back-edge nodes that represent transfer of control to the scope entry. This way, every access graph is acyclic, which simplifies the implementation of conflict-detection algorithms. Next, the SCCs of the loop's CFG are computed, and the same procedure as for functions is applied to construct the access graph. In this case, non-trivial SCCs correspond to nested loops. The access graph for a callsite is a simple flow graph that executes one of the subscopes corresponding to functions possibly called at the callsite.

## 3.3 Computation of Conflict-Free Scopes

Algorithm 1 determines a set of scopes for each memory block $B$, such that the total number of cache misses of $B$ is bounded by the sum of the frequencies of these scopes. It traverses the scope graph bottom-up, starting at the leaves, and determines for every memory block, whether it is conflict-free with respect to the current scope. If the memory block is not conflict-free, the routine CollectScopes computes conflict-free subscopes.

The goal of CollectScopes is to determine conflict-free single-entry regions in the CFG of the conflicting scope (Algorithm 2). The algorithm traverses the access graph of the scope in topological order, visiting every node exactly once. The procedure assumes the existence of functions NewRegion(v) to create a new single-entry region with header $v$ and AddToRegion(R,v) to add $v$ to region $R$. Back-edge nodes always form a singleton region, as they correspond to edges back to the entry of the access graph (Line 5). In order to expand a region, all predecessors of a node have to be in the same region (Line 6). Moreover, a node can only be in the same region as its predecessors, if the resulting scope is conflict-free with respect to the memory block. Accesses to memory blocks in conflicting subscopes need not be collected, as they have already been handled while processing the subscope (Line 19).

Note that if the call to AddToRegion(R,v) on Line 11 is replaced by NewRegion(v), one obtains a simpler variant of the algorithm that only considers static persistence scopes.

**Algorithm 2** Scope Collection

```
1: procedure CollectScopes(Block B, Scope N)
2:     G ← access graph of N
3:     for all nodes v ∈ topological-traversal(G) do
4:         if ∃u ∈ preds(v) s.t. u is back-edge node then
5:             NewRegion(v)
6:         else if ∃u ∈ preds(v) s.t. ∀w ∈ preds(v)
           GetRegion(u) = GetRegion(w) then
7:             R ← GetRegion(u)
8:             if IsConflictingRegion(B, R ∪ {v}) then
9:                 NewRegion(v)
10:            else
11:                AddToRegion(R, v)
12:            end if
13:        else
14:            NewRegion(v)
15:        end if
16:    end for
17:    S ← { }
18:    for all regions R do
19:        if B is accessed in R and
           R is not a conflicting subscope then
20:            S ← S ∪ R
21:        end if
22:    end for
23:    return S
24: end procedure
```

## 3.4 Conflict Detection

So far, the cache analysis framework did not distinguish between different replacement strategies, but relied on the existence of the decision procedures IsConflictingScope and IsConflictingRegion that we describe next.

A memory block is *conflict-free* with respect to a scope, if it will be loaded from memory at most once during the execution of this scope. For set-associative caches using either the LRU or FIFO replacement strategy, this is the case if the cardinality of the set of all distinct cache lines that (1) map to the same cache set and (2) are possibly accessed during the execution of the scope, is less than or equal to the associativity of the cache. For all variations of the method cache, a memory block is conflict-free if (1) the number of distinct code blocks is less than or equal to the associativity of the cache and (2) the total size of all the distinct accessed code blocks, is less than or equal to the size of the method cache.

IsConflictingScope$(B, N)$ and IsConflictingRegion$(B, R)$ are true if the memory block $B$ is *not* conflict-free with respect to the scope associated with $N$ and $R$, respectively.

The conflict detection strategies described above are sound for both LRU and FIFO caches. The LRU replacement strategy also permits a potentially more precise conflict detection routine, however. In an LRU cache, a memory block is conflict-free with respect to the scope if it is locally persistent with respect to the scope. This in turn is the case if any sequence of accesses between two accesses to the memory block is conflict free.

## 3.5 IPET Modeling

The integration of the cache miss constraints into the IPET model is shown in Algorithm 3. In the algorithm, we write $f(x)$ to denote a linear expression that corresponds to the execution frequency of $x$. For all memory access nodes, we introduce a *cache miss variable* that models the frequency of cache misses at this point in the program. The cost of this variable is the cost

**Algorithm 3** Extend IPET

```
1: procedure ExtendIPET(ScopeGraph G)
2:     Scopes ← CacheScopeAnalysis(G)
3:     for all access nodes N do
4:         add cache miss instruction variable N_m
5:         assert N_m ≤ f(N)
6:         cost(N_m) = miss penalty for N
7:     end for
8:     for all memory blocks B do
9:         let A = access nodes for B
10:        assert Σ_{N∈A} f(N_m) ≤ Σ_{S∈Scopes[B]} f(S)
11:    end for
12: end procedure
```

of a cache miss of the corresponding memory block, and is reflected in the objective function of the ILP. Furthermore, each of these variables is obviously bounded by the frequency of the corresponding memory access (Line 5). For all memory blocks potentially accessed, we assert that the sum of all corresponding cache miss variables is bounded by the sum of the frequencies of scopes in the conflict-free scope set for that block (Line 10).

## 4 Evaluation

The target platform for our evaluation is the Patmos architecture [17]. Patmos uses a five-stage in-order dual-issue RISC pipeline; floating-point operations are performed in software. The `patmos-clang` compiler builds on the LLVM compiler framework, and provides support for WCET analysis, as well as a backend for the Patmos architecture [14]. The benchmarks are compiled using the default optimization settings (-O2), which enables most LLVM optimizations. Measurements were carried out using `pasim`, a cycle-accurate simulation of the Patmos architecture. We do not use a data cache and assume data resides in a scratch-pad memory. In our setting, the relative influence of the instruction cache performance on the performance of the processor is thus more pronounced. Since Patmos is a time-composable architecture, this decision does not have any effect on the timing of the instruction cache.

For the memory timing, we use the actual timings of an SDRAM controller developed for Patmos [10]. This memory controller is able to hide latencies for longer burst lengths. In the evaluation we use burst sizes of 8 words that take 11 cycles for a read.

The WCET analysis is carried out using `platin`, a tool that is bundled with our compiler and is primarily intended to bridge the gap between compilers and WCET analysis tools. For evaluation purposes, `platin` supports the extraction of flow fact hypothesis from test runs. This is useful for early-stage development estimates, and provides an excellent way to compare cache analysis results with measurements [7]. The cache analysis itself is only provided with information on infeasible paths, whereas the WCET calculation uses all flow facts that were extracted from test the runs, including the observed frequency of basic blocks.

For our evaluation, we considered benchmarks from two well-established benchmark suites for WCET analysis; the MRTC benchmark suite [6] and PapaBench [13].In this section, we compare the analysis results of a set-associative instruction cache and three variations of the method cache: (1) a set-associative cache with LRU replacement, (2) a fixed block method cache, using LRU replacement, (3) a variable block method cache with FIFO replacement, and (4) a variable size method cache with FIFO replacement.

Table 1 shows the evaluation results for 1 KB caches, using our scope-based cache analysis technique. We chose rather small 1 KB caches to stress test the analysis, as using larger

| | benchmark | I$-8 | FB-4 | VB-8 | VS-8 | VS-8-NS |
|---|---|---|---|---|---|---|
| 1 | ndes | 1.00 | 1.07 | 1.07 | *1.05* | 1.24 |
| 2 | jfdctint | 1.00 | 1.11 | 1.11 | 1.11 | *1.00* |
| 3 | cnt | 1.00 | 1.08 | 1.08 | 1.08 | *1.00* |
| 4 | fdct | 1.00 | 1.09 | 1.09 | 1.09 | *1.00* |
| 5 | adpcm | 1.00 | 1.01 | 1.01 | 1.01 | 1.01 |
| 6 | edn | 1.00 | 1.06 | 1.06 | *1.06* | 1.13 |
| 7 | select | 1.00 | 1.09 | 1.09 | *1.11* | 1.35 |
| 8 | fbw/send_to_pilot | 1.00 | 1.18 | 1.13 | *1.09* | 1.59 |
| 9 | qsort | 1.00 | 1.75 | 1.67 | *1.12* | 5.55 |
| 10 | fbw/check_failsafe | 1.00 | 1.60 | 1.60 | *1.31* | 1.81 |
| 11 | fbw/check_values | 1.00 | 1.60 | 1.60 | *1.31* | 1.82 |
| 12 | ud | 1.00 | 1.08 | 1.07 | *1.02* | 1.12 |
| 13 | fbw/ppm_task | 1.00 | 1.44 | *1.39* | 1.40 | 1.85 |
| 14 | nsichneu | 1.00 | 1.18 | 1.18 | *1.18* | 1.29 |

**Table 1** Comparison of an instruction cache and three different method cache variants

caches results in many of the benchmarks being trivial to analyze. For all method cache configurations but the last one, we used a reasonable default for the function splitter, which prefers block of roughly 256 bytes if possible. The baseline is the performance of a standard 8-way set-associative cache (`I$-8`). The next column provides the relative WCET performance of a fixed-block method cache (`FB-4`). We chose a 4-way cache here to have reasonable large block sizes (256 bytes) that functions need to be split to. The fourth column (`VB-8`) shows the performance of variable-block method cache with a block size of 128 bytes.

The column `VB-8` shows the performance of variable-block method cache with associativity 8. The third method-cache variant is a variable-sized cache, where code blocks do not need to be multiples of some block size. The result for this cache using the usual function splitter setting is shown in column `VS-8`. Additionally, we evaluated the performance of the variable size method cache when the function splitter only splits functions down to fit in the cache (`VS-8-NS`).

It can be seen that the method cache performs equal or poorer than the instruction cache in all explored configurations. However, for most benchmarks the increase of the WCET bound is in an acceptable range for the variable-sized block method cache configuration. What is surprising is that the very simplistic fixed-block cache performs quite well due to the good compiler support through function inlining and splitting. This might be an indication that there is more headroom to adapt the function splitter to provide better results for the variable-sized method cache.

## 5 Related Work

The analysis presented in this paper builds on our earlier work on method cache analysis for JOP [8]. The previous analysis was tied to the concept of a Java method, did not use the scope-graph representation, and only considered methods as scopes. Metzlaff and Ungerer also compared the WCET bounds for different instruction cache architectures [12]. For the method cache, their CHMC analysis represents possible cache states as the powerset of all possible cache configurations. As this is prohibitively expensive if the number of possible

cache configurations grows, the cache has to be reset at the analysis start, and might not be applicable to larger applications [11].

For set-associative caches, the CHMC analysis of Grund and Reineke [4] exploits the fact that when memory blocks are accessed repeatedly, it becomes possible to classify accesses in a FIFO cache. Their work was the first to demonstrate that CHMC analysis is feasible for FIFO caches. An innovative approach by the same authors is to use results from cache competitive analysis [15]. The idea of this analysis is to determine the maximum number of misses for a smaller LRU cache that is known to perform no better than the FIFO cache. However, this analysis effectively reduces the size of the cache visible to the analysis by a factor of two.

Whereas early persistence analyses were unsound, recent articles published corrected dataflow algorithms that promise improvements compared to CHMC analyses [9, 1]. The implementation in [1] combines persistence analysis and CHMC, which appears to be an advisable strategy in practice, especially when dealing with complex architectures. Neither article, however, considers the FIFO replacement strategy, fine-grained scopes or memory blocks that belong to more than one scope.

Guan et. al. recently presented an approach to FIFO cache analysis that, similar to ours, derives cache miss constraints [5]. Instead of precise scopes, they attempt to improve precision by deriving *cache miss ratios* relative to the scope entry. Their evaluation demonstrates significant improvements compared to a CHMC-based FIFO analysis, and could possibly be integrated into our framework.

## 6 Conclusion

In this paper, we presented a new cache analysis framework that generalizes work on cache persistence analysis and it is applicable to a wide range of instruction caches using either FIFO or LRU replacement strategies.

We used this cache analysis and our LLVM-based compiler to explore method caches in a realistic setting. We found that even in combination with a memory controller that is limited to bursts of fixed length, our analysis is able to attest the method cache a WCET performance that is competitive to an 8-way set-associative instruction cache. Furthermore, we showed that adding explicit support for the method cache in the compiler has a high impact on the performance of the method cache.

In future work, we plan to enhance the function splitter algorithm and integrate it with the cache analysis so that the cache block region formation can profit from the knowledge of the analysis. Finally, we also plan to apply the scope-based cache analysis to data caches, where persistence analysis has been proven to be useful before.

## References

1 Christoph Cullmann. Cache persistence analysis: Theory and practice. *ACM Transactions on Embedded Computer Systems*, 12(1s):40, 2013.

2 Philipp Degasperi, Stefan Hepp, Wolfgang Puffitsch, and Martin Schoeberl. A method cache for Patmos. In *to appear: Proceedings of the 17th IEEE Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC 2014)*, Reno, Nevada, USA, June 2014. IEEE.

3 Daniel Grund and Jan Reineke. Abstract interpretation of FIFO replacement. In *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09, pages 120–136, Berlin, Heidelberg, 2009. Springer-Verlag.

**4** Daniel Grund and Jan Reineke. Precise and efficient FIFO-replacement analysis based on static phase detection. In *Proceedings of the 2010 22nd Euromicro Conference on Real-Time Systems*, ECRTS '10, pages 155–164, 2010.

**5** Nan Guan, Xinping Yang, Mingsong Lv, and Wang Yi. Fifo cache analysis for wcet estimation: A quantitative approach. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 296–301, San Jose, CA, USA, 2013. EDA Consortium.

**6** Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET benchmarks - past, present and future. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, July 2010.

**7** Benedikt Huber, Wolfgang Puffitsch, and Peter Puschner. Towards an open timing analysis platform. In *11th International Workshop on Worst-Case Execution Time Analysis*, July 2011.

**8** Benedikt Huber and Martin Schoeberl. Comparison of implicit path enumeration and model checking based WCET analysis. In *Proceedings of the 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 23–34, Dublin, Ireland, July 2009. OCG.

**9** Bach Khoa Huynh, Lei Ju, and Abhik Roychoudhury. Scope-aware data cache analysis for WCET estimation. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 203–212, 2011.

**10** Edgar Lakis and Martin Schoeberl. An SDRAM controller for real-time systems. In *Proceedings of the 9th Workshop on Software Technologies for Embedded and Ubiquitous Systems*, 2013.

**11** Stefan Metzlaff and Theo Ungerer. Impact of instruction cache and different instruction scratchpads on the wcet estimate. In *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 1442 –1449, june 2012.

**12** Stefan Metzlaff and Theo Ungerer. A comparison of instruction memories from the WCET perspective. *Journal of Systems Architecture*, (0):–, 2013.

**13** Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean Paul Bahsoun, and Marianne De Michiel. PapaBench: a free real-time benchmark. In *Proceedings of 6th International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2006.

**14** Peter Puschner, Daniel Prokesch, Benedikt Huber, Jens Knoop, Stefan Hepp, and Gernot Gebhard. The T-CREST approach of compiler and WCET-analysis integration. In *9th Workshop on Software Technologies for Future Embedded and Ubiquitious Systems (SEUS 2013)*, pages 33–40, 2013.

**15** Jan Reineke and Daniel Grund. Relative competitive analysis of cache replacement policies. In *Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '08, pages 51–60, 2008.

**16** Martin Schoeberl. A time predictable instruction cache for a Java processor. In *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)*, volume 3292 of *LNCS*, pages 371–382, Agia Napa, Cyprus, October 2004. Springer.

**17** Martin Schoeberl, Pascal Schleuniger, Wolfgang Puffitsch, Florian Brandner, Christian W. Probst, Sven Karlsson, and Tommy Thorn. Towards a time-predictable dual-issue microprocessor: The Patmos approach. In *First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011)*, pages 11–20, Grenoble, France, March 2011.

**18** Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2/3):157–179, 2000.