Modeling the Function Cache for Worst-Case Execution Time Analysis*

Raimund Kirner
Real-Time Systems Group
Institut fuer Technische Informatik
Technische Universitaet Wien, Austria
raimund@vmars.tuwien.ac.at

Martin Schoeberl
System-on-a-Chip Group
Institut fuer Technische Informatik
Technische Universitaet Wien, Austria
mschoebe@mail.tuwien.ac.at

ABSTRACT

Static worst-case execution time (WCET) analysis is done by modeling the hardware behavior. In this paper we describe a WCET analysis technique to analyze systems with *function caches*, a special kind of instruction cache that caches whole functions only. This cache was designed with the aim to be more predictable for the worst-case than existing instruction caches. Within this paper we developed a cache analysis technique for the function cache. One of the new concepts of this analysis technique is the *local persistence* analysis, which allows to precisely model the function cache.

Categories and Subject Descriptors

B.4 [**Memory Structures**]: Performance Analysis and Design Aids; B.8 [**Performance and Reliability**]: Performance Analysis and Design Aids

General Terms

Performance,Verification

Keywords

worst-case execution time, WCET, cache analysis, function cache

1. INTRODUCTION

The analysis of the worst-case execution time (WCET) is mandatory to reason about the timing behavior of safety-critical real-time systems.

In this paper we describe the WCET analysis for systems with a *function cache* [8], a special type of instruction cache that stores complete functions. This cache organization is motivated by the fact that, for example, Java programs usually consists of short methods and this organization is a better predictable choice for real-time systems.

The idea of a *function* cache is to load complete functions into the cache instead of filling the cache with individual blocks on a miss during program execution. The cache is filled on function calls and returns. This means that all cache misses are lumped together with a known miss penalty. All other instructions inside a function are guaranteed hits. This property also simplifies, and speeds up, the instruction fetch path in the processor. It is not necessary to access the tag memory for each instruction. The tag memory is usually the critical path in a cache access.

The miss penalty depends on the size of the function to be loaded. For the call instruction the size of the called function and for the return instruction the size of the caller determine the cache load time.

The motivation for the function cache is, besides the simpler hardware, to provide a predictable instruction cache solution for real-time systems. In [11] a simple version of the function cache has been included into a WCET analysis tool. The miss penalty on a call or return is added to the control flow graph (CFG) as additional vertices before the call instruction and after the call instruction. The algorithm considered a *function cache that can hold only two functions*. In that case only the leaves of the call tree can result in a cache hit. This reduction allowed to perform the cache analysis function local in the CFG.

In this paper we extend WCET analysis to a function cache that is called *variable block cache* in [8]. The cache consists of several blocks similar to cache lines in a traditional cache. Still whole functions are loaded into the cache. A function in the cache has to be loaded into consecutive blocks. A function loaded over the cache end to the cache start is considered continuous as the cache is addressed with a modulo counting program counter.

The replacement policy on a cache miss is as follows: A *next* pointer indicates the start block for a new function. After loading the new function the pointer is incremented by the number of blocks the new loaded functions consumes. Once a function is partially overwritten, all remaining cache blocks of this function are marked as empty. This replacement strategy provides an age ordering with respect to the load time of a function.

The implemented cache can be configured with respect to the cache size and the block size. A smaller block size results in a better hit to miss relation, but consumes either more hardware or more time for the hit detection. In our

*This work has been partially supported by the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) under contract P18925-N13.

DAC 2007, June 4–8, 2007, San Diego, California, USA.

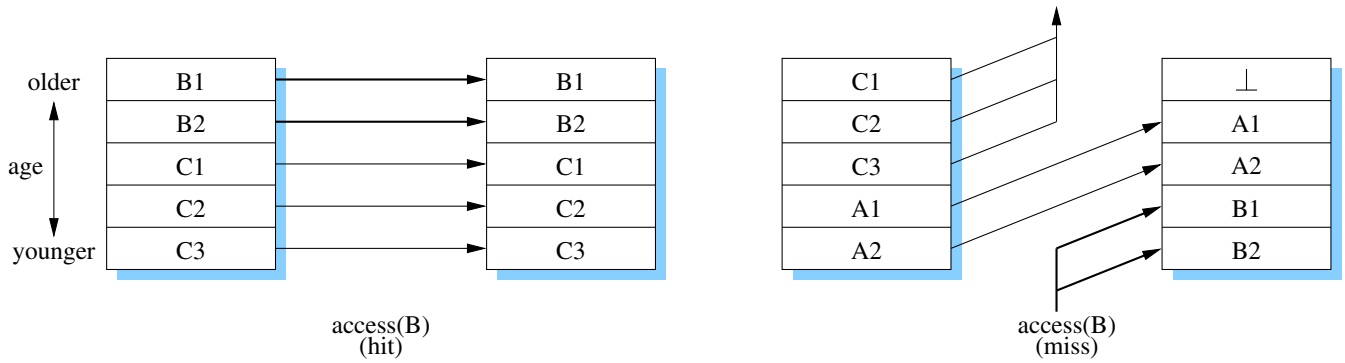

Figure 1: Update of the function cache. The left scenario shows the cache state update in case that the access to function B is a hit (no change). The right scenario shows the cache state update in case that the access to function B is a miss (function C is removed).

example we use a 4 KB function cache with 16 blocks.

The proposed cache was first introduced by Schöberl and already implemented in hardware for the Java processor JOP [9] and named *method cache* [8]. However, the concept can also be applied to a standard RISC processor programmed in C/C++. The only restrictions are that no jumps into a function and out of a function are allowed. The cache handling is part of the call and return instruction. Furthermore, the length of the function needs to be stored in the executable. A convenient place to store this information is one word before the function.

2. MODELLING THE FUNCTION CACHE

In this section we describe how we model the function cache for WCET analysis. Lets assume that the function cache has n cache lines; L is the set of cache lines. F is the set of functions of the program to be analyzed. Each function $f \in F$ is divided into a set C_f of code fractions by the size of a cache line: $|C_f| = \lceil \frac{size(f)}{size(\text{cache line})} \rceil$. The set of all code fractions of the program is denoted as C: $C = \bigcup_{f \in F} C_f$.

A function reference is a function that maps a function identifier to its memory blocks: $FR : ID \mapsto F$.

2.1 Concrete Cache Semantics

To compare the cache model with the informal description given in Section 1, we first discuss the concrete cache semantics more formally. To enable a cache analysis algorithm that is similar to others described in the literature, we represent the cache in a different, but logically equivalent, form: instead of using a *next* pointer to determine the position where replacement is done, we arrange the cache blocks according to their age of loading. Thus, when a replacement is done, the new blocks are added at the place of youngest age. All already loaded blocks are shifted towards the oldest age.

To model the cache state we define C' as the set of code fractions C extended with the element $\perp$ to indicate a cache line with no valid content: $C' = C \cup \perp$. Now we define the concrete state of the cache as $S_c : L \mapsto C' \mid \forall \ell \in L, \forall f \in F.\ S_c(\ell) \in C_f \rightarrow (\forall m \in C_f, \exists \ell' \in L.\ m = S_c(\ell'))$. The condition of the concrete cache state expresses the fact that a function is either loaded completely into the cache or, in the other case, none of its blocks are loaded.

The state of the function cache is updated each time a function reference FR occurs (call or return instruction). The update is described by the *update function* $\mathcal{U} : S_c \times FR \mapsto S_c$, which uses *least recently loaded* (LRL) as replacement policy. The LRL replacement of also called FIFO (first in, first out). An intuitive understanding of how $\mathcal{U}$ is defined is intended by the examples of cache updates given in Figure 1.

2.2 Abstract Cache Semantics

To analyze the cache behavior we use an abstraction of the concrete cache semantics. In the abstract cache state we are able to express that multiple blocks may be potentially loaded at a certain cache line. The *abstract* cache state $\widetilde{S}_c$ is defined as: $\widetilde{S}_c : L \mapsto 2^C \mid \forall \ell \in L, \forall f \in F, \forall m \in \widetilde{S}_c(\ell).\ m \in C_f \rightarrow (\forall m' \in C_f, \exists \ell' \in L.\ m' \in \widetilde{S}_c(\ell'))$. As with the concrete cache state, a function may be only loaded completely into the cache or, in the other case, none of its blocks are loaded.

Depending on the function reference FR, the *abstract update function* $\widetilde{\mathcal{U}}$ transforms an abstract cache state into another one: $\widetilde{\mathcal{U}} : \widetilde{S}_c \times FR \mapsto \widetilde{S}_c$.

To get a more compact representation of the cache state, we represent all memory blocks of a function by a single block, extended with a length attribute. Figure 2 shows an example of the update of an abstract cache state. As the function D has a length of 2, the previous cache content is shifted two steps towards the older age of the cache. As function A does not fit completely into the cache anymore, it gets removed from $\widetilde{S}_c$ when loading function D.

2.3 Categorization of Cache Behavior

Before doing the path analysis to calculate the WCET we analyze the cache behavior to classify it into five categories. These categories are described in Table 1. We do the classification of the cache behavior by data-flow analysis [1]. The classification we use is similar to the four categories of [13], but our cache analysis is different in the sense that only few instructions update the cache state (function call/return). The cache analysis in [13] is called *abstract interpretation*, but it is also a data-flow analysis.

Category	*Name*	*Meaning*
ah	always hit	Each access to the cache is a hit.
am	always miss	Each access to the cache is a miss.
gp	globally persistent	For the whole program execution, the first access is not classified, but the second all further accesses are a hit.
lp(ϑ)	locally persistent	For each entering of a context ϑ, the first access is not classified, but the second all further accesses are a hit.
nc	not classified	The access is not classified as one of the categories above.

Table 1: Categorizations of Function Cache behavior

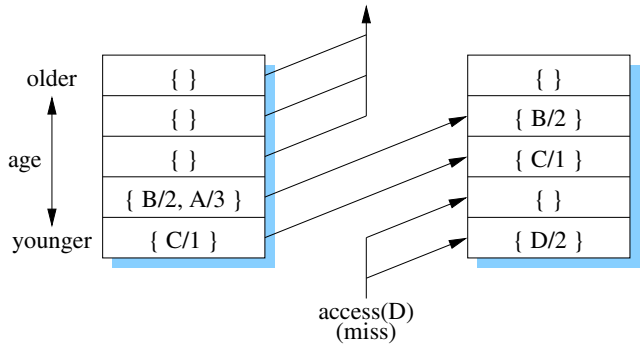

Figure 2: Update of abstract function cache.

The data domain of our data-flow analysis is a lattice of finite height (derived from the abstract cache state $\widetilde{S}_c$). The abstract update function as well as the join function, which is described below, are monotonic. Thus, it is guaranteed that this data-flow analysis will always terminate.

The program contexts for which we do data-flow analysis are described in Section 3. Data-flow analysis uses at each node four instances of the data domain: *IN*, *OUT*, *GEN*, and *KILL*. The content of *GEN* and *KILL* is defined by the abstract cache semantics, *OUT* is initialized as $\perp$ (empty cache). The global fixpoint solution of the data-flow analysis is obtained by solving the equations of Equation 1 and Equation 2 iteratively:

$$OUT_i = (IN_i \;/\; KILL_i) \cup GEN_i \tag{1}$$

Note that for all nodes i that do not contain a function reference (all nodes, except function calls and its return locations) we get $KILL_i{=}GEN_i{=}\emptyset$ and thus $OUT_i{=}IN_i$.

Besides the update function $\widetilde{\mathcal{U}}$ we have to define a join function $\widetilde{\mathcal{J}}$ to merge two or more abstract cache states: $\widetilde{\mathcal{J}} : \widetilde{S}_c{\times}\widetilde{S}_c \mapsto \widetilde{S}_c$. The extension to join more than two states is defined as follows: $\widetilde{\mathcal{J}}(s_1,\ldots,s_n) = \widetilde{\mathcal{J}}(\widetilde{\mathcal{J}}(s_1,\ldots),s_n)$. Assuming a node i has n predecessors $1\ldots n$, the following equation is used:

$$IN_i = \widetilde{\mathcal{J}}(OUT_1,\ldots,OUT_n) \tag{2}$$

As in [13], to obtain the categorization of Table 1 we use four data-flow analyzes (with different definitions of $\widetilde{\mathcal{J}}$). To obtain the cached functions of the combined abstract state, $\widetilde{\mathcal{J}}$ uses either *intersection* or *union* operation. The new age of each cached function is either the *maximum* or the *minimum* age of the function. The four analyzes are defined as follows:

Must Analysis: calculates the function references that are guaranteed to be a cache hit. $\widetilde{\mathcal{J}}_{must}$ uses *intersection* and *maximum*. Each $f \in FR$ is categorized as *always hit* (ah), if $\exists \ell{\in}L.\ (\widetilde{S}_c(\ell) \cap C_f) \neq \emptyset$, where $\widetilde{S}_c$ is the set IN_i at the concrete program context i of the function reference.

May Analysis: calculates the function references that may be a cache hit. $\widetilde{\mathcal{J}}_{may}$ uses *union* and *minimum*. Each $f \in FR$ is categorized as *always miss* (am), if $\forall \ell{\in}L.\ (\widetilde{S}_c(\ell) \cap C_f) = \emptyset$.

Global Persistence Analysis: calculates the function references where the first access of the whole program execution cannot be classified as hit or miss, but each subsequent access is guaranteed to be a hit. $\widetilde{\mathcal{J}}_{gp}$ uses *union* and *maximum*. Global persistence analysis uses an additional virtual cache line $\ell_\perp$, which holds those memory blocks of functions that could have been removed from the cache [13]. Each $f \in FR$ is categorized as *globally persistent* (gp), if

$$\Big(\exists \ell \neq \ell_\perp.\ (\widetilde{S}_c(\ell) \cap C_f) \neq \emptyset\Big) \;\wedge\; \Big((\widetilde{S}_c(\ell_\perp) \cap C_f) = \emptyset\Big)$$

Local Persistence Analysis: The local persistence analysis is a generalization of the global persistence analysis. Local persistence analysis calculates the function references where the first access for each entering of a context ϑ cannot be classified as hit or miss, but each subsequent access within the same context ϑ or a nested context $\vartheta{\circ}\vartheta'$ is guaranteed to be a hit, as long as ϑ is not left. Thus, global persistence analysis is the special case where the only context ϑ considered by the analysis is the whole program. There is no special constraint of what a program context should be, as long as the contexts can be hierarchically nested. For example, a context $\vartheta{\circ}\vartheta'$ is a nested context of ϑ. Defining $\mathcal{T}$ as the set of contexts and τ as the set of context specifiers, a context $\vartheta \in \mathcal{T}$ is thus a chain $(\tau_1,\tau_2,\tau_3,\ldots)$ with $\tau_i \in \tau$. The length of a context ϑ is denoted as $|\vartheta|$, two contexts ϑ_1 and ϑ_2 are chained together by $\vartheta_1{\circ}\vartheta_2$. If a chain of context specifiers starts with the program entry, it is called an *absolute context*; otherwise it is called a *relative context*.

$\widetilde{\mathcal{J}}_{lp}$, the join operation for local persistence analysis, also uses *union* and *maximum*.

Similar to global persistence analysis, local persistence analysis uses a set of additional cache lines $\{\ell_\perp^\vartheta \mid \vartheta \in \mathcal{T}_{lp}\}$, where $\mathcal{T}_{lp}$ is the context categorization for finding local persistency. Note that $\mathcal{T}_{lp}$ may be much coarser than the contexts of the cache analysis itself (described in Section 3), especially in the case of having many contexts for the cache analysis. For example, the program representation for data-flow analysis may have

100.000 contexts, but one models $\mathcal{T}_{lp}$ with only 500 different context categories.

An example of an abstract cache state for local persistence analysis is shown in Figure 3. The bars in the area labelled "*removal contexts*" represent the cache lines $\ell_{\perp}^{\vartheta}$. For example, $\ell_{\perp}^{1.1.1}$ shows that the function D has been removed from the cache in context *1.1.1*.

Each $f \in FR$ at context ϑ' is categorized as *locally persistent within context* ϑ (denoted as $\mathrm{lp}(\vartheta)$), if

$$\forall \vartheta_1, \vartheta_2 \in \mathcal{T}_{lp}.\ ((|\vartheta_2| \geq 0) \wedge (\vartheta_1 = \vartheta' \circ \vartheta_2)) \rightarrow (C_f \cap \ell_{\perp}^{\vartheta_1} = \emptyset)$$

Above definition says that a memory reference f is only locally persistent relative to context ϑ if there is no inner scope of ϑ where the content of this memory reference could be removed from the cache.

Furthermore, we require a local persistency to be optimal, i.e., there is no outer scope of ϑ that also classifies for local persistence:

$$\forall \vartheta_1, \vartheta_2 \in \mathcal{T}_{lp}.\ ((|\vartheta_2| \geq 1) \wedge (\vartheta = \vartheta_1 \circ \vartheta_2)) \rightarrow (\exists \vartheta_3 \in \mathcal{T}_{lp}.\ (|\vartheta_3| \geq 0) \wedge (C_f \cap \ell_{\perp}^{\vartheta_1} \neq \emptyset))$$

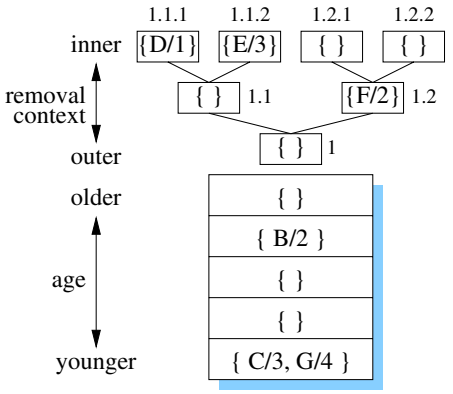

Figure 3: Abstract cache state for local persistence analysis.

3. CONTEXTS OF THE CACHE ANALYSIS

In Section 2.3 we have discussed how to classify the cache behavior for different program contexts. What these program contexts are is discussed in this section.

The behavior of the function cache depends on the control flow of the program and may also differ with the call context of a function. Thus a *super graph* [6], which is the combination of the control flow graph (CFG) and the call graph (CG) of a program, is an interesting start for our purpose of cache analysis. However, to distinguish different call contexts, *inlining* is a common technique. To distinguish different instances of function calls within a loop, *unrolling* is used. With inlining and unrolling the result is a so-called *extended super graph* (ESG).

In practice, the unrolling and inlining is not done by simply duplicating the nodes of the super graph, as this explodes the program size dramatically. Instead, the super graph is virtually extended by copying only those data elements used for data-flow analysis. Further, in cache analysis it is not effective to use techniques like the call-string approach [12] to distinguish different call contexts [5]. Instead, the typical technique for cache analysis is to distinguish between the first access and all remaining accesses [13, 2]. This technique can be applied for both, separation of loop contexts and separation of call contexts (in case of direct recursion) [13].

For a (direct) recursive function call c at context ϑ we represent the context of the first call as $\vartheta \circ \mathtt{C}[c]$ and all of its recursive calls as $\vartheta \circ \mathtt{R}[c]$. Similarly, we denote the context of the first iteration of a loop in context ϑ as $\vartheta \circ \mathtt{F}[\ell]$ and for all other iterations as $\vartheta \circ \mathtt{O}[\ell]$. Thus each context ϑ of a node is a sequence of local contexts $\vartheta_1, \ldots, \vartheta_n \mid \vartheta_i \in \{\ \mathtt{C}[c], \mathtt{R}[c], \mathtt{F}[\ell], \mathtt{O}[\ell]\ \}$. The only reason of using different letters to specify the contexts of loops and the contexts of function calls is to improve readability.

4. IPET-BASED PATH ANALYSIS

After we have categorized the function cache behavior, we perform the path analysis by translating the *extended super graph* (ESG) of the program into a set of integer linear programming (ILP) constraints. This technique is called *implicit path enumeration technique* (IPET) and is already described in Puschner et al. for CFGs having control-flow edges of constant execution time [7]. To solve the constraints and calculate the WCET bound we use the tool *lp_solve*[1]. The unfolded ESG is basically a graph $\langle E, N, s \rangle$ where E is the set of control-flow edges, N is the set of nodes (e.g., basic blocks), and s is the distinguished start node. The goal function to be maximized is of the form $\sum_{e \in E} t(e) \cdot c(e)$, where $t(e)$ is a constant that represents the execution time of edge e and $c(e)$ is one of the flow variables that have to be calculated from the flow constraints. A simple kind of flow constraints is of the form

$$\sum_{e \in IN(n)} c(e) = \sum_{e \in OUT(n)} c(e) \qquad (3)$$

where $IN(n)$, $OUT(n)$ are the incoming and outgoing control-flow edges of a node $n \in N$.

4.1 Modeling the Function Cache

For modeling the function cache we have to treat a subset $EFR \subseteq E$ separately, which represents the call edges and return edges of a function call. For each edge $e \in EFR$ we actually use two edges in the goal function, e_h and e_m, where e_m represents a function reference in case of a cache miss and e_h in case of a cache hit. The additional flow constraint $c(e_m) + c(e_h) = c(e)$ links the cache hits and misses to the control flow.

Depending on the function cache categorization we have to generate extra cache constraints. If the edge e of a function reference is categorized as *am* respectively *ah*, we generate the extra constraint '$c(e_h) = 0$' respectively '$c(e_m) = 0$'. If e is categorized as *gp*, we generate the additional constraint '$c(e_h) \geq c(e) - 1$', which is in fact a simplified coding of '$c(e_h) = c(e) = 0 \vee (c(e) > 0 \wedge c(e_h) = c(e) - 1)$', when searching for the maximum of the goal function (WCET bound). In case of a local persistency *lp(ϑ)* of an edge e we generate the additional constraint '$c(e_h) \geq c(e) - (\sum_{e' \in E_{IN}(\vartheta)} c(e'))$',

[1] freely available at `http://groups.yahoo.com/group/lp_solve/files/`

Result	*Measured*	$Calc_{NC}$	$Calc_{DB}$	$Calc_{VB}$
Cycles	12340	16238	12988	12468
Rel.	+0 %	+31.6 %	+5.3 %	+1.0 %

Table 2: Results of WCET analysis

where $E_{IN}(\vartheta)$ describes the set of CFG edges leading from an outer context into the context ϑ. If e is categorized as *nc*, no additional cache constraint is generated.

Maximizing the goal function given in Equation 3 together with the flow constraints described above, delivers an upper bound of the WCET.

5. CASE STUDY

We evaluate our function cache analysis on the Java processor JOP. JOP is a processor designed to be predictable with respect to WCET. In [10] it was shown that the WCET analysis at the microcode level and showed that the execution time of the bytecode, the instruction set of a Java virtual machine, can be predicted cycle accurate. There are no shared processor resources over bytecode boundaries. Therefore, the processor is free of timing anomalies as found in current RISC processors [4].

We evaluate our analysis on a small, synthetic benchmark, consisting of a few functions, two loops and a *if* statement. This case study is small enough to precisely measure the WCET. However, it is still complex enough to show the benefit of the proposed function cache analysis.

Table 2 compares the results of three different WCET analyzes absolutely and relatively with the measured WCET value:

- Column $Calc_{NC}$ shows the WCET value if we assume that the cache behavior is always a miss (a *function cache* that can cache only a single function would behave like this).
- Column $Calc_{DB}$ shows the result using a simple modeling of a *function cache* consisting of only two blocks. This simple analysis by Schöberl and Perdersen has been described in [11].
- The result of the WCET analysis using the cache modeling described in this paper is shown in column $Calc_{VB}$.

Even this is a simple case study, allowing us to generate the ILP constraints manually, it already shows that the *function cache* modeling of this paper is superior than the previous one. The local persistence analysis of this paper will provide further improvements with case studies of larger size. However, to analyze the performance of the function cache in general, and the performance of the local persistence analysis, additional experiments have to be done.

6. RELATED WORK

There exist several approaches of modeling direct-mapped or set-associative instruction caches for WCET analysis.

Müller et al. used data-flow analysis to mark each instruction with an instruction cache category. The analysis was first done for direct-mapped caches [2]. The authors introduced the cache categories *first miss*, *first hit*, *always miss*, *first hit*. Caches with unknown behavior where also classified as *always miss*. In subsequent research the authors extended their analysis to set-associative caches.

The implicit path enumeration technique (IPET) based on integer-linear programming (ILP) was introduced by Puschner et al. to analyze programs of arbitrary shape [7].

Li et al. combined the path analysis based on the IPET approach directly with additional ILP constraints describing the cache behavior. The result was an elegant analysis of systems with direct-mapped or set-associative instruction caches [3]. Unfortunately, this combination resulted in quite high analysis times for real-size caches and programs.

Ferdinand et al. describe an instruction cache analysis, whose results are used for path analysis based on ILP [13]. The approach of the authors is quite similar to our approach, as we use a similar cache categorization and a graph unfolding that is comparable to their VIVU (virtual inlining, virtual unrolling) approach. The main new concept of our cache analsis is the local persistence analsis, which has been developed especially for the function cache.

7. SUMMARY AND CONCLUSION

In this paper we have described an analysis technique to statically calculate the WCET of systems with a *function cache*, a special type of instruction cache that stores whole functions only. Compared to other methods of instruction cache modeling we included a special cache categorization, called *local persistency*, denoted as $\mathrm{lp}(\vartheta)$. The *local persistency* cache categorization has been developed especially to model the function cache, however, it could be also applied to the analysis of commonly used cache architectures like set-associative caches. A first evaluation with a synthetic case study was done to check that the function cache analysis method works. The WCET analyzability shown in this paper provides evidence that the novel *function cache* is a promising concept of instruction caching.

Future work is to finish the implementation of the described analysis method and then compare it on a collection of benchmarks with existing methods of modeling set-associative instruction caches.

8. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, June 1997. ISBN 0-201-10088-6.

[2] R. D. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding Worst-Case Instruction Cache Performance. In *Proc. 15th Real-Time Systems Symposium (RTSS)*, pages 172–181, Brookline, Massachusetts, Dec. 1994.

[3] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *Proc. 17th Real-Time Systems Symposium*, pages 254–263. IEEE, Dec. 1996.

[4] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *RTSS '99: Proceedings of the 20th IEEE Real-Time Systems Symposium*, page 12, Washington, DC, USA, 1999. IEEE Computer Society.

[5] F. Martin, M. Alt, R. Wilhelm, and C. Ferdinand. Experimental comparison of *call string* and *functional* approaches to interprocedural analysis. In *Proc. 7th*

International Conference on Compiler Construction, LNCS 1383. Springer, 1998.

[6] E. E. Myers. A precise inter-procedural data flow algorithm. In *Proc. 8th ACM Symposium on Principles of Programming Languages*, pages 219–230, 1981.

[7] P. Puschner and A. V. Schedl. Computing maximum task execution times – a graph-based approach. *Journal of Real-Time Systems*, 13:67–91, 1997.

[8] M. 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.

[9] M. Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.

[10] M. Schoeberl. A time predictable Java processor. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE 2006)*, pages 800–805, Munich, Germany, March 2006.

[11] M. Schoeberl and R. Pedersen. WCET analysis for a Java processor. In *Proceedings of the Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2006)*, Paris, France, October 2006.

[12] M. Sharir and A. Pnueli. *Program Flow Analysis: Theory and Application*, chapter 7, Two approaches to interprocedural data flow analysis, pages 189–233. Prentice Hall, 1981.

[13] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separate cache and path analyses. *Real-Time Systems*, 18(2/3), 2000.