Component-wise Instruction-Cache Behavior Prediction
– Extended Abstract –

Oleg Parshin* Abdur Rakib† Stephan Thesing* Reinhard Wilhelm*

Abstract

The precise determination of worst-case execution times (WCETs) for programs is mostly being performed on linked executables, since all needed information and all machine parameters influencing cache performance are available to the analysis. This paper describes how to perform a component-wise prediction of the instruction-cache behavior guaranteeing conservative results compared to an analysis of a linked executable. This proves the correctness of the method based on a previous proof of correctness of the analysis of linked executables. The analysis is described for a general A-way set associative cache. The assumptions are that the replacement strategy is LRU and inter-module call relationship is acyclic.

1. Introduction

So far, WCET-determination methods mostly work on fully linked executables, since in this case all needed machine-level information about code allocation is fixed and available. This paper presents a method for component-wise analysis of the instruction-cache behavior, thus supporting incremental program development. This method uses the notion of *cache-equivalence* of memory allocations to express that one allocation of a module in the memory will display exactly the same cache behavior as the equivalent one. This equivalence is exploited to influence the linker, which can choose between several equivalent allocations when placing a module into the executable. The overall picture is the following:

1. A set of modules making up the real-time program is given. Cyclic calling dependencies are assumed to exist only inside modules, i.e., the inter-module call relationship graph is acyclic.
2. A bottom-up module-wise analysis computes a sound approximation to the cache contents at all program points of all modules taking into account safe upper approximations of the cache damages due to external function calls. The results of the module-wise analysis are combined conservatively with respect to an (in general more precise) analysis of a linked executable.

2. Cache memory architectures

A cache can be characterized by three major parameters:

cache size s is the total size of the cache, i.e. the number of bytes it may contain.

line size l **(also called block size)** is the number of contiguous bytes that are transferred from memory on a cache miss. The cache can hold at most $n = s/l$ blocks.

associativity A is the number of cache locations where a particular memory block can reside. The cache contains $\eta = n/A$ *sets*.

If a block can reside in exactly A locations, then the cache is called *A-way associative*. If a block can reside in any cache location ($A = n$), then the cache is called *fully associative*. If a block can reside in exactly one location ($A = 1$), then it is called *direct mapped*. Thus, fully associative and direct mapped cahes are special cases of the A-way cache.

In the case of an associative cache, a cache line has to be selected for replacement when the cache is full and the processor requests further data. This is done according to the *replacement strategy*. Common strategies are *LRU* (Least Recently Used), *FIFO* (First In First Out), and *random*.

In this paper we consider A-way set associative cache with LRU replacement strategy. Detailed formal description of the cache semantics can be found in [1].

[*] Rearch reported here was supported by the transregional research center AVACS (Automatic Verification and Analysis of Complex Systems) of the DFG (Deutsche Forschungsgemeinschaft). These authors are with FR Informatik, Universität des Saarlandes, Postfach 15 11 50, 66041 Saarbrücken, Germany, {oleg, thesing, wilhelm}@cs.uni-sb.de

[†] Supported by the IMPRS (International Max-Planck Research School for Computer Science). This author is with Max-Planck Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany, hossain@mpi-sb.mpg.de

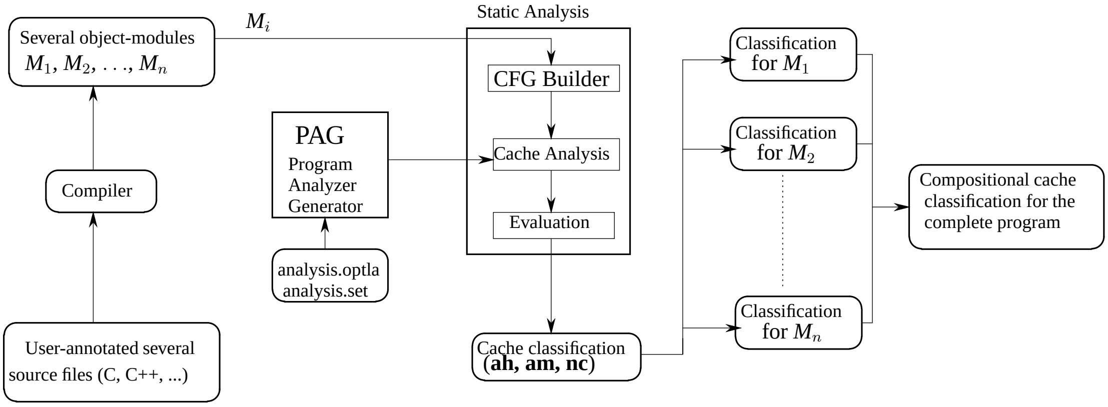

Figure 1. The structure of analysis framework.

3. The analysis framework

As input to our analysis framework (c.f. Figure 1) we have a set of object modules that are yet to be linked to form an executable. We also assume, that the user provides additional information, such as number of loop iterations, upper bound for recursions etc. A parser reads the object modules and reconstructs the control flow graph for each module [9]. Nodes of the control flow graph represent *basic blocks*. For each basic block it is known which memory blocks it references.

Analysis results are computed using Abstract Interpretation [2]. The *collecting semantics* of a module is safely approximated using *abstract cache states* [11].

We distinguish two kinds of analyses. The *must analysis* determines a set of memory blocks that are definitely in the cache at a given program point whenever execution reaches this point. The *may analysis* determines set of memory blocks that may be in the cache at given program point. The complement of may analysis is used to determine which blocks are definitely not in the cache.

The analyses are used to classify the memory references into *always hit*, *always miss,* or *non-classified* [1].

Termination of analyses is guaranteed [3].

4. Notion of equivalence

Our aim is to ensure that the results of cache behavior analysis obtained at module level can be combined in a conservative way with respect to the results of the analysis of a linked executable. The relative and absolute address spaces of a module before and after linking, respectively, will in general be different. Consequently, the cache behavior of a module before and after linking may be different.

We perform cache analysis for each module using available relative address information and a fixed mapping of relative addresses of an object module to cache sets.

This section is concerned with the conditions under which the different allocations of modules will display equivalent cache behavior (in the sense of number of cache hits/misses, not exact content of the cache).

4.1. Equivalent memory allocation with respect to the fixed set mapping

Suppose a set of modules $M_1, M_2, \ldots, M_p$ forms a program. Each module consists of the set $\{m^i_0, \ldots, m^i_{k_i}\}$ of memory blocks (each memory block has the size of a cache line). A linker combines these modules in the sequence $M_1, M_2, \ldots, M_p$. We assume that all object modules are created with base address 0. According to this assumption, in each module M_i the block m^i_0 is mapped to the first set of the cache.

A question arises when we consider a linked executable, whether the absolute address in the executable, which corresponds to a relative address inside the module, will be mapped to the same set. Since linkers only shuffle segments of object modules but do not rearrange their internals, all the internal memory addresses become offsets from the new base address of the modules.

Since the number of the set to which block m_i is mapped is determined as $(i \mod \eta)$ [3], two memory blocks are mapped to the same set, if the difference q between their addresses is a multiple of $(\eta \cdot l)$, because each block has the size of l bytes.

In order to preserve a fixed mapping of addresses for module M_1, the executable has to be created with such a base address q. The base addresses of the modules $M_2, \ldots, M_p$ depend on the sum of the sizes of previous modules. To preserve the fixed mapping for these modules, some *wasted space* between them has to be added, such that the base address of each module will be a multiple of $(\eta \cdot l)$.

4.2. Conservative cache-behavior analysis

In order to combine the results of module-wise analysis conservatively with respect to the analysis of a linked executable, we choose a placement of modules according to Section 4.1, i.e., the absolute base address of each module should be a multiple of $(\eta \cdot l)$.

5. Proposed analysis method

As input for the analysis we have a directed acyclic intermodule dependency graph where vertices represent modules and edges represent call relations between modules. Our analysis is based on a bottom-up approach, starting from modules which are not dependent on any other module in this graph (i.e., with $outdegree = 0$). At each stage (for each module) the produced results are kept in a special data structure so that analysis results of a module will be available later to the modules, which have calls to this module.

5.1. Module-wise cache analysis

We analyze cache behavior separately for each procedure using the framework from Section 3 in the two following call contexts: (i) *local call* – a call between two procedures in the same module; and (ii) *external call* – a call between two procedures in the different modules. In the last case we combine at the return point the analysis result of the caller with the analysis result of the callee.

During the analysis of a procedure we assume that cache is initially empty for the must analysis and everything may be in the cache with age 0 for the may analysis.

The analysis for modules, which have calls to other modules (i.e, with $outdegree \neq 0$) uses the precomputed information of its called modules, whenever needed at the calling points. Since the cache contents of the calling module will be changed according to the called module's cache information, we have to consider the cache damages due to calls to the external procedures. In the following subsection we describe how to handle such cache effects during calls between modules.

5.2. Cache damage analysis

The aim of the cache damage analysis is to provide the correct information about the bounds of replacements in a particular set, i.e., to determine an upper-bound of the number of replacements occurring in a particular set for the must analysis, and a lower-bound for the may analysis.

Let us consider must analysis. The elements of each set of the caller's cache are age by the upper bound of the number of replacements in the same set of the callee's cache, and the elements of this set in the callee's cache retain their age during the cache damage update (cf. formal cache semantics in [1]).

If before the call some memory block m is in the caller's cache set f with age x, and the upper bound of replacements in this set due to the call is a, then this block will survive in the cache after the call if $x + a \leqslant A - 1$.

A procedure may be also called from inside a loop of another procedure. If some block m is in the callee's cache at the return point with the age x, then it will be in the caller's cache after return. Since the procedure is called inside a loop, this block may survive in the cache during all following iterations, and be in the caller's cache with the age y before the call. If $y + a \leqslant A - 1$ then this block will survive in the caller's cache during the call with the age $z = y + a$. Hence, there may exist multiple copies of the block m in the same set with different ages. In order to avoid such a situation we flush all callee's memory blocks which are in the caller's cache before the call.

5.3. Properties of the method

The following steps are followed during the analysis of a module: (i) construct the control flow graph for each procedure, (ii) identify the local and the external calls, (iii) analyze all possible paths considering local and external calls of a procedure, (iv) update the cache information according to the call contexts using cache-damage analysis result, and (v) store the cache analysis information for each procedure in the corresponding data structure.

The analysis result of the complete program is the composition of the analysis results of all the modules. We have the following properties of the method.

Termination of the analysis. Termination of the may and must analysis is guaranteed. Cache damage analysis terminates, since the domain is finite, update functions are monotone, and the join functions are monotone, associative and commutative (cf. full version of this paper [4]).

The results of the module-wise analysis are conservative. Our analysis is based on a bottom-up approach and during the analysis of each procedure we take safe initial approximations according to Section 5.1. Therefore, we can conclude the following theorem:

Theorem 1 *The results of component-wise cache behavior prediction are conservative to an analysis results of a linked executable, assuming the equivalent module placement according to Section 4.1.*

For the sake of space, we omit the proof of the theorem. The proof can be found in the full version of the paper [4] .

Maximum wasted memory space. As we have seen in Section 4.1, some memory space is wasted in order to preserve the equivalent memory allocation w.r.t. the fixed set mapping. The wasted memory space in the worst case is $(\eta \cdot l - 1) \cdot (p - 1)$.

6. Related work

Most of the research on precise cache-behavior prediction is being performed on fully linked executables.

The work [5] propose a compositional instruction-cache behavior prediction. Their goal is to decrease the analysis effort by splitting the analysis into several phases, a module-level analysis, preprocessing calls, and a compositional analysis using this information. The motivation comes from the claim that only small programs can be analyzed by the traditional methods. However, as shown in [10] these methods realized in commercially available tools are in routine use in the aeronautics and also in the automotive industry. Their method needs the availability of all modules, while ours analyzes modules as they are compiled and combines the analysis results in a conservative way. A research group at the Laboratory of Embedded Systems Innovation and Technology (LIT) described in [6] a framework, PERF, which works with the object code generated by the integrated tools in order to determine execution-time limit estimations for functions that compose a real-time system. Their cache behavior prediction method is based on the extended timing schemata proposed by [7, 8].

7. Conclusions and future work

We have presented a technique for predicting the cache behavior for A-way set associative instruction caches component-wise. Given a set of object code-modules, a parser reads the object code-modules and reconstructs the control flow. The cache analysis technique works in a bottom-up way starting from minimal modules of the module dependency graph. The analysis computes a sound approximation to the cache contents at all program points of all modules taking safe upper approximations of the cache damages of called external functions into account. The analysis results can be combined in a conservative way with respect to an analysis of a fully linked executable.

Our current research direction includes component-wise data cache behavior prediction. Data cache analysis is more difficult than instruction cache analysis, because the effective data address may change when an instruction referencing data is executed repeatedly. We will implement a tool to estimate the worst-case execution time of a real-time system, where the system is given as a set of object code modules.

References

[1] Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache Behavior Prediction by Abstract Interpretation. In *Proceedings of the Third International Symposium on Static Analysis*, pages 52–66. Springer-Verlag, 1996.

[2] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

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

[4] Oleg Parshin, Abdur Rakib, Stephan Thesing, and Reinhard Wilhelm. Component-wise Instruction-Cache Behavior Prediction. Accepted for 2nd International Symposium on Automated Technology for Verification and Analysis, 2004.

[5] Kaustubh S. Patil. Compositional Static Cache Analysis Using Module-Level Abstraction. Master's thesis, North Carolina State University, 2003.

[6] Douglas Renaux, João Góes, and Robson Linhares. WCET Estimation from Object Code Implemented in the PERF Environment. In *2^{nd} International Workshop on Worst-Case Execution Time Analysis (Satellite Event to ECRTS'02)*, pages 28–35, Technical University of Vienna, Austria, June 18, 2002.

[7] Lim S-S et al. An Accurate Worst-Case Timing Analysis for RISC Processors. *In IEEE Real-Time Systems Symposium*, pages 97–108, December 1994.

[8] Min S.L et al. An Accurate Instruction Cache Analysis Technique for Real-time Systems. *In Proceedings of the Workshop on Architectures for Real-time Applications*, April 1994.

[9] Henrik Theiling. Extracting Safe and Precise Control Flow from Binaries. In *Proceedings of the 7th Conference on Real-Time Computing Systems and Applications*, Cheju Island, South Korea, 2000.

[10] Stephan Thesing et. al. An Abstract Interpretation-Based Timing Validation of Hard Real-Time Avionics Software Systems. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN 2003)*, pages 625–632. IEEE Computer Society, June 2003.

[11] Reinhard Wilhelm. Why AI + ILP is good for WCET, but MC is not, nor ILP alone. In *Proceedings of the Fifth International Conference on Verification, Model Checking and Abstract Interpretation*, Venice, Italy, January 2004.