A Fast and Efficient Cache Persistence Analysis

Christian Ferdinand

Universität des Saarlandes / Fachbereich Informatik
Postfach 15 11 50 / D-66041 Saarbrücken / Germany
Phone: +49 681 302 5573 Fax: +49 681 302 3065
ferdi@cs.uni-sb.de

Abstract

In this paper, abstract interpretation is applied to the problem of predicting the cache behavior of programs. The goal of our analysis is to approximate the set of all cache states for each program point, i.e., the collecting semantics. We take a two step approach: First, we define an auxiliary semantics of machine programs called the cache semantics that determines the contents of caches. Then, from the cache semantics one simple analysis by abstract interpretation is developed that allows to determine memory blocks that will be persistent in the cache. This is in contrast to former approaches [1, 6] that required two different analyses and specific interprocedural analysis methods. The persistence analysis is designed generic with the cache logic as parameter. Experimental results are presented that demonstrate the applicability of the analysis.
Keywords: cache behavior prediction, program analysis, abstract interpretation, program analysis, cache memories, real time applications, worst case execution time prediction

1 Cache Memories and Real-Time Applications

Caches are used to improve the access times of fast microprocessors to relatively slow main memories. They can reduce the number of cycles a processor is waiting for data by providing faster access to recently referenced regions of memory. Caching is more or less used for all general purpose processors, and, with increasing application sizes it becomes more and more relevant and used for high performance microcontrollers and DSPs.

Programs with hard real-time constraints have to be subjected to a schedulability analysis. It has to be determined whether all timing constraints can be satisfied. The degree of success for such a timing validation [19] depends on sharp WCET (Worst Case Execution Time) estimations. For example for hardware with caches, the typical worst case assumption is that all accesses miss the cache. This is an overly pessimistic assumption which leads to a waste of hardware resources.

2 Overview

In the following Section we briefly sketch the underlying theory of abstract interpretation and present the program analyzer generator PAG.

Cache memories are briefly described in Section 4. In Section 5 we give a semantics for programs that reflects only memory accesses (to fixed addresses) and its effects on cache memories, and in Section 7.1 we present the *persistence analysis* that computes a set of memory blocks that will be persistent in the cache, i.e., will not be removed from the cache and describe how the results of the analysis can be interpreted.

Section 8 describes possible extensions to data and combined caches. In Section 9 we present and discuss the results of our practical experiments, and Section 10 describes related work.

3 Program Analysis by Abstract Interpretation

A program analyzer takes a program as input and computes some interesting *runtime* properties. Most of these properties are undecidable. Hence, both correctness and completeness of the computed information are not achievable together. Program analysis makes no compromise on the correctness side; the computed information is reliable as for enabling optimizing transformations [20]. It can't thus guarantee completeness. The quality of the computed information, usually called its *precision*, should be as good as possible.

There is a well developed theory of static program analysis called *abstract interpretation* [5]. With this theory, correctness of a program analysis can be easily derived. According to this theory a program analysis is determined by an *abstract semantics.*

Usually the meaning of a language is given as functions for the statements of the language computing over a concrete domain. A domain is a complete partially ordered set of values. For such a semantics, an abstract version consists of a new simpler abstract domain and simpler abstract functions which define the abstract meaning for every program statement.

For the abstract semantics used in this paper and an input program, a system of recursive equations can be constructed. The variables in this system are the values of the abstract domain for every program point. In this equation system, the value at a program point depends on the values at all program points which can directly precede the execution of this program point. The *control flow graph* of a program describes every possible flow of control and therefore all dependencies between the variables of the equation system.

Lattice theory underlying abstract interpretation states that the recursive equation system can be solved by fix-point iteration if the abstract domain has only finite ascending chains, i.e., every chain of values $v_1 \sqsubset v_2 \sqsubset \cdots$ has only finite length, and if in addition every semantic function is monotonic.

The program analyzer generator PAG [2] offers the possibility to generate a program analyzer from a description of the abstract domain and of the abstract semantic functions in two high level languages, one for the domains and the other for the semantic functions. Domains can be constructed inductively starting from simple domains using operators like constructing power sets and function domains. The semantic functions are described in a functional language which combines high expressiveness with efficient implementation. Additionally the user has to supply a join function combining two domain values into one. This function is applied whenever a point in the program has two (or more) possible execution predecessors.

4 Cache Memories

A cache can be characterized by three major parameters:

- *capacity* is the number of bytes it may contain.
- *line size* (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 $n = \textit{capacity}/\textit{line size}$ blocks.
- *associativity* is the number of cache locations where a particular block may reside. $n/\textit{associativity}$ is the number of *sets* of a cache.

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

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 a replacement strategy. Common strategies are *LRU* (Least Recently Used), *FIFO* (First In First Out), and *random.*

5 Cache Semantics

We restrict our description to the semantics of A-way set associative caches with LRU replacement strategy. The fully associative and the direct mapped caches are special cases of the A-way set associative cache where $A = n$ and $A = 1$ rsp.

In the following, we consider an A-way set associative cache as a sequence of (fully associative) sets $F = \langle f_1, \ldots, f_{n/A} \rangle$ where $n = \textit{capacity}/\textit{line size}$, a set f_i as a sequence of set lines $L = \langle l_1, \ldots, l_A \rangle$, and the store as a set of memory blocks $M = \{m_1, \ldots, m_s\}$.

The function $adr : M \to \mathbb{N}_0$ gives the address of each memory block. The function $set : M \to F$ determines the set where a memory block would be stored (% denotes the modulo division):

$$set(m) = f_i; \text{ where } i = adr(m)\%(n/A) + 1$$

To indicate the absence of any memory block in a set line, we introduce a new element I; $M' = M \cup \{I\}$.

Our cache semantics separates two key aspects:

- The set where a memory block is stored: This can be statically determined as it depends only on the address of the memory block. The dynamic allocation of memory blocks to sets is modeled by the *cache states.*
- The aspect of associativity and the replacement strategy within one set of the cache: Here, the history of memory reference executions is relevant. This is modeled by the *set states.*

Definition 5.1 (concrete set state) A *(concrete) set state* is a function $s : L \to M'$. S denotes the set of all concrete set states.

Definition 5.2 (concrete cache state) A *(concrete) cache state* is a function $c : F \rightarrow S$. C denotes the set of all concrete cache states.

If $s(l_x) = m$ for a concrete set state s, then x describes the relative age of the memory block according to the LRU replacement strategy and not the physical position in the cache hardware.

The *update* function describes the side effects on the set (cache) of referencing the memory:

- The set where a memory block may reside in the cache is uniquely determined by the address of the memory block, i.e., the behavior of the sets is independent of each other.
- The LRU replacement strategy is modeled by using the positions of memory blocks within a set to indicate their relative age. The order of the memory blocks reflects the "history" of memory references[1].

  The most recently referenced memory block is put in the first position l_1 of the set. If the referenced memory block m is in the set already, then all memory blocks in the set that have been more recently used than m are shifted by one position to the next set line, i.e., they increase their relative age by one. If the memory block m is not yet in the set, then all memory blocks in the cache are shifted and the 'oldest', i.e., least recently used memory block is removed from the set.

Definition 5.3 (set update) A set update function $\mathcal{U}_S : S \times M \rightarrow S$ describes the new set state for a given set state and a referenced memory block.

Definition 5.4 (cache update) A cache update function $\mathcal{U}_C : C \times M \rightarrow C$ describes the new cache state for a given cache state and a referenced memory block.

Updates of fully associative sets with LRU replacement strategy are modeled in the following way:

$$\mathcal{U}_S(s,m) = \begin{cases} [l_1 \mapsto m, \\ \ l_i \mapsto s(l_{i-1}) \mid i = 2 \ldots h, \\ \ l_i \mapsto s(l_i) \mid i = h+1 \ldots A]; & \text{if } \exists l_h : s(l_h) = m \\ [l_1 \mapsto m, \\ \ l_i \mapsto s(l_{i-1}) \text{ for } i = 2 \ldots A]; & \text{otherwise} \end{cases}$$

Notation: $[y \mapsto z]$ denotes a function that maps y to z. $f[y \mapsto z]$ denotes a function that maps y to z and all $x \neq y$ to $f(x)$.

Updates of A-way set associative caches are modeled in the following way:

$$\mathcal{U}_C(c,m) = c[set(m) \mapsto \mathcal{U}_S(c(set(m)),m)]$$

[1] Our semantics describes the *observable* behavior of cache memories. We do not intend to mimic the physical cache implementation.

6 Control Flow Representation

We represent programs by control flow graphs consisting of nodes and typed edges. The nodes represent *basic blocks*[2]. For each basic block, the sequence of references to memory is known[3], i.e., there exists a mapping from control flow nodes to sequences of memory blocks: $\mathcal{L} : V \rightarrow M^*$.

We can describe the working of a cache with the help of the update function $\mathcal{U}_C$. Therefore, we extend $\mathcal{U}_C$ to sequences of memory references:

$$\mathcal{U}_C(c, \langle m_1, \ldots, m_y \rangle) = \mathcal{U}_C(\ldots \mathcal{U}_C(c, m_1) \ldots, m_y)$$

The cache state for a path $(k_1, \ldots, k_p)$ in the control flow graph is given by applying $\mathcal{U}_C$ to the initial cache state c_I that maps all set lines in all sets to I and the concatenation of all sequences of memory references along the path: $\mathcal{U}_C(c_I, \mathcal{L}(k_1). \ \ldots \ .\mathcal{L}(k_p))$.

7 Abstract Semantics

In order to generate an analyzer, the program analyzer generator PAG requires the specification of a domain (this will be the abstract cache states), a transfer function (this will be the abstract cache update function), and a join function that is used to combine two elements of the domain from two different paths in the control flow graph.

The domain for our abstract interpretation consists of *abstract cache states* that are constructed from *abstract set states*. In order to keep track of memory blocks that have already been replaced in the cache, i.e., are relatively older than all memory blocks in the corresponding set, we introduce additional set lines $l_\top$ in which we will collect the possibly replaced memory blocks, $L' = L \cup l_\top$.

Definition 7.1 (abstract set state) An *abstract set state* $\hat{s} : L' \rightarrow 2^{M'}$ maps set lines to sets of memory blocks. $\hat{S}$ denotes the set of all abstract set states.

Definition 7.2 (abstract cache state) An *abstract cache state* $\hat{c} : F \rightarrow \hat{S}$ maps sets to abstract set states. $\hat{C}$ denotes the set of all abstract cache states.

The abstract semantic functions describe the effect of a memory reference on an element of the abstract domain. The **abstract set (cache) update** function $\hat{\mathcal{U}}$ for abstract set (cache) states is an extension of the set (cache) update function $\mathcal{U}$ to abstract set (cache) states.

On control flow nodes with at least two[4] predecessors, *join*-functions are used to combine the abstract cache states.

Definition 7.3 (join function) A *join function* $\hat{\mathcal{J}} : \hat{C} \times \hat{C} \mapsto \hat{C}$ combines two abstract cache states.

[2] A basic block is a sequence (of fragments) of instructions in which control flow enters at the beginning and leaves at the end without halt or possibility of branching except at the end.

[3] This is appropriate for instruction caches and can be too restrictive for data caches and combined caches. See Chapter 8 for weaker restrictions.

[4] Our join functions are associative. On nodes with more than two predecessors, the join function is used iteratively.

Example 7.1 ($\hat{\mathcal{U}}_{\hat{S}}$)

	l_1	l_2	l_3	l_4	$l_\top$
$\hat{s}$	$\{m_a\}$	$\{\}$	$\{m_b, m_c\}$	$\{m_d\}$	$\{m_e\}$
$\hat{\mathcal{U}}_{\hat{S}}(\hat{s}, m_f)$	$\{m_f\}$	$\{m_a\}$	$\{\}$	$\{m_b, m_c\}$	$\{m_d, m_e\}$

The address of a memory block determines the set in which it is stored. This is reflected in the abstract cache update function in the following way:

$$\hat{\mathcal{U}}_{\hat{C}}(\hat{c}, m) = \hat{c}[set(m) \mapsto \hat{\mathcal{U}}_{\hat{S}}(\hat{c}(set(m)), m)]$$

The join function is similar to set *union*, except that if a memory block s has two different ages in two abstract cache states then the join function takes the oldest age.

$$\hat{\mathcal{J}}_{\hat{S}}(\hat{s}_1, \hat{s}_2) = \hat{s}, \text{ where:}$$

$$\begin{aligned}
\hat{s}(l_x) &= \{m \mid \exists l_a, l_b \text{ with } m \in \hat{s}_1(l_a), m \in \hat{s}_2(l_b) \text{ and } x = max(a,b)\} \\
&\cup \{m \mid m \in \hat{s}_1(l_x) \text{ and } \nexists l_a \text{ with } m \in \hat{s}_2(l_a)\} \\
&\cup \{m \mid m \in \hat{s}_2(l_x) \text{ and } \nexists l_a \text{ with } m \in \hat{s}_1(l_a)\}
\end{aligned}$$

This includes $\hat{s}(l_\top) = \hat{s}_1(l_\top) \cup \hat{s}_2(l_\top)$.

Example 7.2 ($\hat{\mathcal{J}}_{\hat{S}}$)

	l_1	l_2	l_3	l_4	$l_\top$
$\hat{s}_1$	$\{m_e\}$	$\{m_b\}$	$\{m_c\}$	$\{m_d\}$	$\{m_a\}$
$\hat{s}_2$	$\{m_c\}$	$\{m_e, m_f\}$	$\{m_a\}$	$\{m_d\}$	$\{m_b\}$
$\hat{\mathcal{J}}_{\hat{S}}(\hat{s}_1, \hat{s}_2)$	$\{\}$	$\{m_e, m_f\}$	$\{m_c\}$	$\{m_d\}$	$\{m_a, m_b\}$

The join function for abstract cache states applies the join function for abstract set states to all its abstract set states:

$$\hat{\mathcal{J}}_{\hat{C}}(\hat{c}_1, \hat{c}_2) = [f_i \mapsto \hat{\mathcal{J}}_{\hat{S}}(\hat{c}_1(f_i), \hat{c}_2(f_i)) \mid \text{ for all } 1 \leq i \leq n/A]$$

An abstract cache state $\hat{c}$ at a control flow node k that references a memory block m is interpreted in the following way: Let $\hat{s} = \hat{c}(set(m))$. If $m \in \hat{s}(l_y)$ for $y \in \{1, ..., A\}$, then the maximal relative age of m is less or equal to y provided that m is in the cache. From this follows that m cannot be replaced in the cache after an initial load. m is *persistent*. This means that all (repeated) executions of the reference to m can produce *at most one* cache miss.

7.2 Termination of the Analysis

There are only a finite number of cache lines and for each program a finite number of memory blocks. This means, the domain of abstract cache states $\hat{c} : L' \rightarrow 2^S$ is finite. Hence, every ascending chain is finite. Additionally, the abstract cache update functions $\hat{\mathcal{U}}$ and the join functions $\hat{\mathcal{J}}$ are monotonic. This guarantees that our analysis will terminate.

In the next section we will present the *persistence analysis*. It determines memory blocks that are *persistent* in the cache, i.e., stay in the cache after they have been initially been loaded into the cache. A memory reference to a memory block that can be determined to be *persistent* can produce at most one cache miss for repeated executions. This information can be used to predict the cache behavior of programs.

7.1 Persistence Analysis

Our goal is to determine the *persistence* of a memory block, i.e., the *absence* of the possibility that a memory block m is removed from the cache. If there is no possibility to remove m from the cache, then the first reference to m may result in a cache miss, but *all* further references to m cannot result in cache misses.

This is achieved by the persistence analysis as follows: It computes the *maximal* position (relative age) for all memory blocks that *may* be in the cache. This means, the position (the relative *age*) of a memory block in the abstract set state $\hat{s}$ is an upper bound of the positions (the relative *ages*) of the memory block in the concrete set states that $\hat{s}$ represents. In order to keep track of memory blocks that have already been replaced in the cache, i.e., are relatively older than all memory blocks in the corresponding set, we introduce additional set lines $l_\top$ in which we collect the possibly replaced memory blocks with the update and join function.

Let $m_a \in \hat{s}(l_x)$. The position (relative age) x of a memory block m_a in a set can only be changed by references to memory blocks m_b with $set(m_a) = set(m_b)$, i.e., by memory references that go into the same set. Other memory references do not change the position of m_a. The position is also not changed by references to memory blocks $m_b \in \hat{s}(l_y)$ where $y \leq x$, i.e., memory blocks that are already in the cache and are "younger" or the same age as m_a. m_a can stay in the cache at least for the next $A - x$ references that go to the same set and are not yet in the cache or are *older* than m_a.

The meaning of an abstract cache state is given by a *concretization function* $conc_{\hat{C}} : \hat{C} \to 2^C$. For the memory blocks that are collected in the additional set lines $l_\top$ during the analysis, our analysis cannot determine the persistence. This does not mean that these memory blocks are never in the cache during program execution. Accordingly, the concretization function makes no assertion with respect to these memory blocks. The concretization function for the persistence analysis $conc_{\hat{C}}$ is given by:

$$conc_{\hat{C}}(\hat{c}) = \{c \mid \forall 1 \leq i \leq n/A : c(f_i) \in conc_{\hat{S}}(\hat{c}(f_i))\}$$
$$conc_{\hat{S}}(\hat{s}) = \{s \mid \forall 1 \leq a \leq A : \exists b \in \{1, ..., \top\} : s(l_a) \in \hat{s}(l_b) \text{ and } a \leq b\}$$

We use the following abstract set update function:

$$\hat{U}_{\hat{S}}(\hat{s}, m) = \begin{cases} [l_1 \mapsto \{m\}, & \\ \ l_i \mapsto \hat{s}(l_{i-1}) \mid i = 2 \ldots h-1, & \\ \ l_h \mapsto \hat{s}(l_{h-1}) \cup (\hat{s}(l_h) - \{m\}), & \\ \ l_i \mapsto \hat{s}(l_i) \mid i = h+1 \ldots A, & \\ \ l_\top \mapsto \hat{s}(l_\top)]; & \text{if } \exists h \in \{1, ..., A\} : m \in \hat{s}(l_h) \\ & \\ [l_1 \mapsto \{m\}, & \\ \ l_i \mapsto \hat{s}(l_{i-1}) \text{ for } i = 2 \ldots A, & \\ \ l_\top \mapsto (\hat{s}(l_\top) - \{m\}) \cup \hat{s}(l_A) \ ]; & \text{otherwise} \end{cases}$$

8 Data Caches and Combined Caches

In [1] methods are described to statically determine the addresses of memory references to procedure parameters or local variables by a static stack level simulation [20]. This allows to use our analysis to predict the behavior of data caches or combined instruction/data caches for programs that use only scalar variables. [1] also describes methods to handle writes to caches for common cache organizations (*write through* and *write back* with *write allocate* or *no write allocate*) as well as write buffers.

9 Practical Experiments

For reasons of simplicity, we restrict our practical experiments to the analysis of instruction caches.

In order to reach acceptable analysis results for programs with procedures, we have to use interprocedural analysis methods. The interprocedural analysis (e.g., [18, 6]) methods differ in which execution contexts are distinguished for a memory reference within a procedure. A context represents the execution stack, i.e., the function calls along the corresponding path in the control flow graph to the instruction. It is represented as a sequence of function calls (*call_f*) for the functions of a program. For our experiments we simply distinguish all execution contexts that corresponds to a sequence of nonrecursive procedures. This guarantees that there are only finitely many execution contexts. It corresponds to a *virtual inlining* of the nonrecursive procedures.

The cache analysis techniques are implemented in a PAG generated analyzer that gets as input the control flow graph of a program and an instruction cache description and produces a categorization *cat* of the instruction/context pairs of the input program.

INST is the set of all instructions *inst* in a program. *CONTEXT* is the set of all distinguished execution contexts *context* of a program. *IC* is the set of all instruction/context pairs *ic*.

$$\begin{aligned} CONTEXT &= \{call\_f\}^* \\ IC &= INST \times CONTEXT \\ cat &: IC \rightarrow \{\textsf{persistent, unknown}\} \end{aligned}$$

The frontend to the analyzer reads a Sun SPARC executable in a.out format. The Sun SPARC is a RISC architecture with pipelined instruction execution. It has a uniform instruction size of four bytes. Our implementation is based on the EEL library of the Wisconsin Architectural Research Tool Set (WARTS).

The objective of our work is to improve the WCET estimation of programs on computer systems with caches. Besides the architecture, the execution time of a program depends on the program path, i.e., the sequence of instructions that are executed. But the program path is usually dependent on the program input and cannot generally be determined in advance. Therefore, a *program path analysis* is part of a WCET analysis [17, 8, 11]. For example, with the help of user annotations, like maximal iteration counts of loops, an architecture dependent worst case execution profile can be determined that gives a conservative approximation to the worst case execution path.

The program path analysis can be very accurate. Yau-Tsun Steven Li and Sharad Malik report that their estimated bounds are within two percent of the (calculated) worst case bounds for their set of benchmark examples [11]. The worst case execution profile allows to compute how often

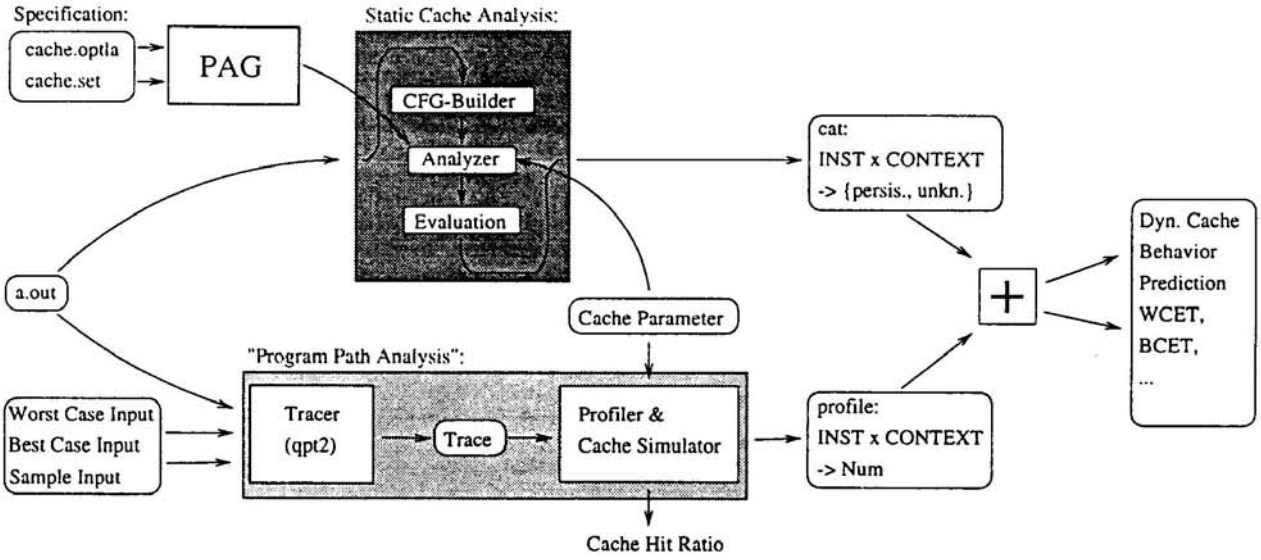


Figure 1: The structure of the analysis.

each instruction/context pair is maximally encountered. Combined with the categorizations of our cache analysis, the overall number of cache hits and cache misses can be estimated (see Figure 1).

In our experiments, we have circumvented the program path analysis problem and combine the categorizations *cat* with "exact" execution profiles instead of worst case execution profiles (see Figure 1). This allows us to assess the effectiveness of our analysis without the influence of possibly pessimistic path analyses. The profilers that produce the profiles are produced with the help of qpt2 (Quick program Profiler and Tracer) [4, 7] that is part of the WARTS distribution. A profiler for a program computes an execution profile *profile*, i.e., the execution counts for the instruction/context pairs.

$$profile \quad : \quad IC \rightarrow \mathbb{N}_0$$

For the experiments we use parts of the program suites of Frank Müller [3, 14], the `djpeg` program of Yau-Tsun Steven Li [10], and some additional programs (see Table 1). For some programs, there exists a worst case input, so that our execution profiles are worst case execution profiles. The programs are compiled with the GNU C compiler version 2.7.2 under SunOS 4.1.4 with -O2, and (if applicable) the FDLIBM (Freely Distributable LIBM) library of SunPro version 5.2.

The programs `fft`, and `stats` use arithmetic library functions. These functions are more or less structured into treatment of special cases, normalization, computation, and final rounding. Not all parts are necessarily executed when the function is called. This uncertain execution path typically leads to less precise predictions.

The AVL tree as implemented in `avl2` is a height balanced binary tree. Every insert or delete operation may lead to a series of recursive calls for re-balancing. The code of the insert and delete operations consists of many cases for the different re-balancing operations called rotations. Such a program structure seems to be rather typical for the handling of many dynamic data structures. The prediction of programs with such a structure is also difficult.

Name	Description	Inst.	*ICs*	Analysis Time	Persistent
`matmult`	50x50 matrix multiplication	154	202	0.02 sec.	87.1%
`ndes`[1]	data encryption	471	615	0.11 sec.	95.9%
`matsum`[1]	100x100 matrix summation	135	142	0.03 sec.	89.4%
`dhry`	Dhrystone integer benchmark	447	493	0.14 sec.	71.8%
`stats`	two arrays sum, mean, variance, standard deviation, and linear correlation	456	1053	0.13 sec.	78.4%
`fft`	fast Fourier transformation	1810	6044	4.84 sec.	70.1%
`djpeg`[2]	JPEG decompression	1760	3838	3.54 sec.	71.4%
`avl2`	inserts and deletes 1000 elements in an AVL tree	614	3378	1.09 sec.	93.6%

[1] Worst case input data

[2] Random input data

Table 1: Test set of C programs with number of instructions, number of distinguished instruction/context pairs, the analysis time on a SUN SPARCstation 20, and percentage of instruction/context pairs that can be determined to be persistent for a 1KB 4-way set associative instruction cache with 16 byte linesize.

The fourth column of Table 1 shows the number of distinguished instruction/context pairs of our analysis. It is a measure for the complexity of the analysis. In our current implementation, the persistence analysis for a given cache configuration can be computed within seconds on a SUN SPARCstation 20 for our test programs (see column 5 of Table 1). In our implementation, there is still room for improvements, though.

The sixth column of Table 1 shows the percentage of the instruction/context pairs that were determined to be persistent by our analysis. To asses the quality of our prediction we use the analysis results to compute guaranteed lower bounds of the cache hit ratios and compare them with measured cache hit ratios.

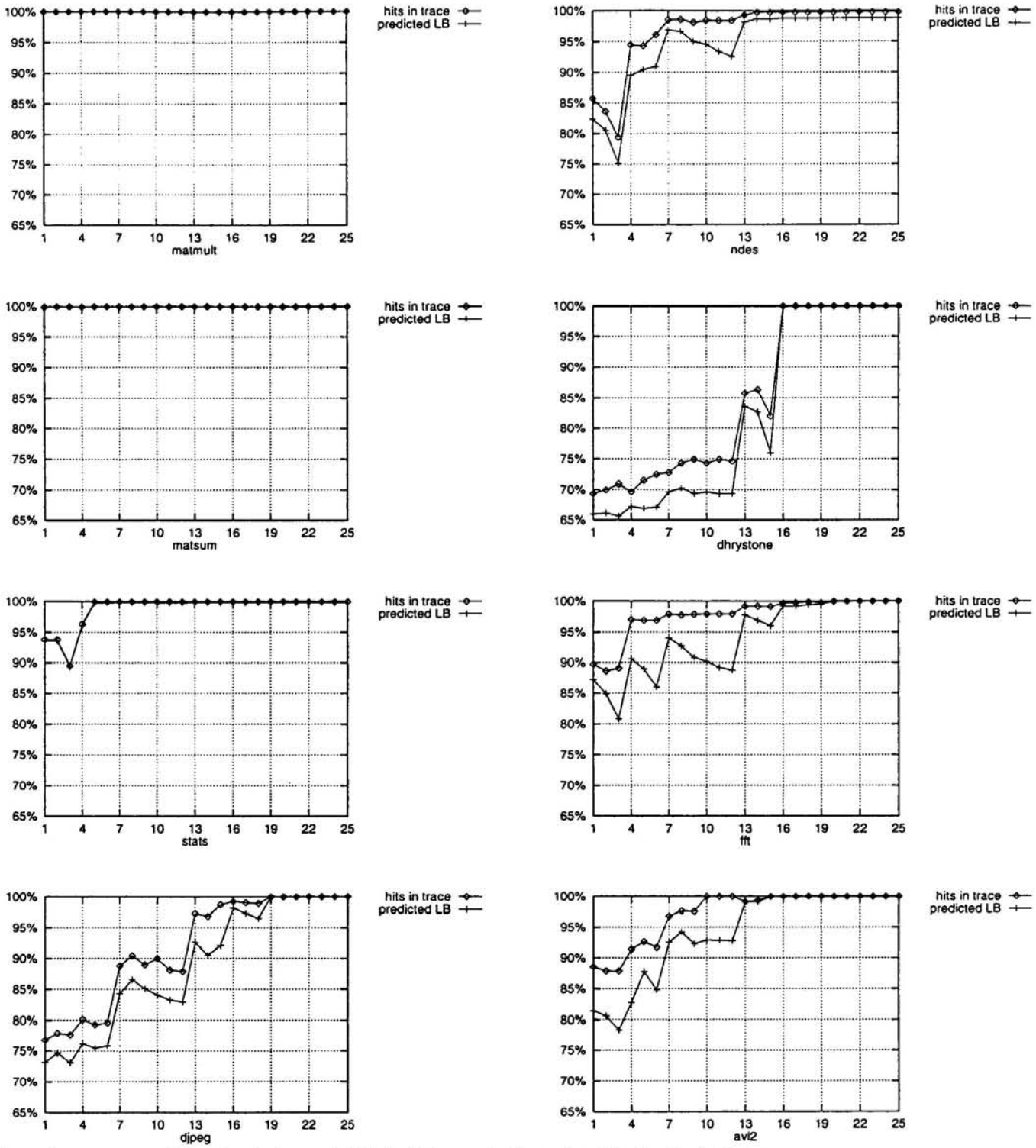
The cache hit ratio of the test programs for different cache configurations is measured by simulating the cache for the program trace. The cache simulation is always started with the empty cache, and we assume uninterrupted execution. For technical reasons, instructions in functions from dynamic link libraries[5] are not traced and their effects on the cache are therefore ignored.

With our analysis results a lower bound of cache hit ratio can be computed by combining the profiles with the results of our analysis. A memory reference (instruction) that is executed n times ($n \geq 1$) and has been categorized as *persistent* counts as *one cache miss* and $n - 1$ *cache hits*. A memory reference (instruction) that is executed n times and could not been categorized as *persistent* counts as and n *cache misses*. The predicted lower bound of the cache hit ratio is the quotient of such cache hits and the number of all executed memory references (instructions).

The predicted cache hit ratios and the measured cache hit ratios of the test programs for various cache configurations are shown in Figure 2 in percent.

Figure 2 can be interpreted as follows: Conditionally executed code, e.g. as found in the arithmetic library functions or in `avl2`, can lead to less precise predictions. There can be a wide variation of

[5] In our case, these are the calls to IO routines and timers.



The cache parameters (size - level of associativity) of the x axis tic marks. The linesize is 16 bytes.

1=128B-1	2=128B-2	3=128B-4	4=256B-1	5=256B-2	6=256B-4	7=512B-1	8=512B-2
9=512B-4	10=512B-8	11=512B-16	12=512B-32	13=1kB-1	14=1kB-2	15=1kB-4	16=2kB-1
17=2kB-2	18=2kB-4	19=4kB-1	20=4kB-2	21=4kB-4	22=8kB-1	23=8kB-2	24=8kB-4
25=20kB-5							

Figure 2: Predicted lower bounds of the cache hit ratios and the measured cache hit ratios for different cache parameters in percent.

the quality of the prediction depending on the cache configuration. For most test programs and cache configurations our method allows to predict the cache behavior quite precisely.

10 Comparison with Related Work

The possibilities to use optimizing compilers to improve cache performance of programs has extensively been studied [12, 13, 16, 21]. But all the proposed program transformations and code reorganizations do not necessarily help in computing deterministic lower bounds of the cache hit ratios of programs.

The work of Arnold, Müller, Whalley, and Harmon has been one of the starting points of our work. [14] describes a data flow analysis for the prediction of instruction cache behavior of programs for direct mapped caches. The extension to set associative instruction caches has later been given in [15]. In contrast to our method that derives semantics based categorizations of memory references only from the results of our analyses, an additional complex bottom-up algorithm over the control flow graph is used to compute a classification of the instructions for each loop level.

In [9, 10] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe describe an integrated method to determine the worst case execution path of a program and to model architecture features like instruction caches and/or pipelines. The problem of finding an accurate worst case execution time bound is formulated as an integer linear program that must be solved, which is a NP-hard problem. This approach has been implemented in the `cinderella` tool[6]. Unlike the method described in [14] or our method that rely only on the control flow graph to determine the cache behavior of a memory reference, user provided *functionality constraints* can be used to describe the control flow more precisely. For direct mapped instruction caches and programs whose execution path is well defined and not very input dependent the predictions can be computed fast and are very accurate [10]. Increasing levels of associativity where the cache behavior of one memory reference depends on more other references and less defined execution paths lead to prohibitively high analysis times.

In [8], Lim et al. describe a general framework for the computation of WCETs of programs in the presence of pipelines and cache memories. Two kinds of pipeline and cache state information are associated with every program construct for which timing equations can be formulated. One describes the pipeline and cache state when the program construct is finished. The other can be combined with the state information from the previous construct to refine the WCET computation for that program construct. Unlike our method that is based on well explored theories and tools for abstract interpretation, the set of timing equations must be explicitly solved. An approximation to the solution for the set of timing equations has been proposed. The usage of an input and output state provides a way for a modularization for the timing analysis. Experimental results are reported for three small programs, but they cannot be easily compared with our experiments.

11 Conclusion

We have described a semantics based analysis method by abstract interpretation that allows to predict the intrinsic cache behavior of programs for various types of one level caches. The theory of abstract interpretation supports the correctness proofs for the analysis and provides efficient

[6] See http://www.ee.princeton.edu/~yauli/cinderella-3.0/

implementation methods.

The analyzers are generated by the program analyzer generator PAG from very concise specifications. The implementation of the analysis is fast and does not necessitate program transformations nor additional analysis phases. No special input of a skilled user is required to tune for acceptable results. This makes it feasible to use our analyses as part of the compilation process to support the automatic schedulability analysis by the compiler.

The applicability of our methods has been shown with the results of our practical experiments.

We directly analyze executables and there are no special compilers or linkers required. Our current implementation supports the SPARC architecture. Other architectures can be supported by supplying additional frontends to our analyzers. The analyses are extensible to accommodate further cache designs like multilevel caches or wrap around line fill.

Acknowledgements

We like to thank Mark D. Hill, James R. Larus, Alvin R. Lebeck, Madhusudhan Talluri, and David A. Wood for making available the Wisconsin architectural research tool set (WARTS), Thomas Ramrath for the implementation of the PAG frontend for SPARC executables, Yau-Tsun Steven Li and Frank Müller for providing their benchmark programs, and Martin Alt and Florian Martin the developers of PAG and Reinhard Wilhelm who invented the name for the analysis for very fruitful discussions.

References

[1] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache Behavior Prediction by Abstract Interpretation. In *SAS'96, Static Analysis Symposium*, LNCS 1145, pages 52–66. Springer, Sept. 1996. Long version accepted for SAS'96 special issue of Science of Computer Programming.

[2] M. Alt and F. Martin. Generation of Efficient Interprocedural Analyzers with PAG. In *SAS'95, Static Analysis Symposium*, LNCS 983, pages 33–50. Springer, Sept. 1995.

[3] R. Arnold, F. Mueller, D. B. Whalley, and M. Harmon. Bounding Worst-Case Instruction Cache Performance. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 172–181, Dec. 1994.

[4] T. Ball and J. Larus. Optimally Profiling and Tracing Programs. In *Conference Record of POPL '92: 19nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 59–70, Jan. 1992.

[5] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Jan. 1977.

[6] C. Ferdinand, F. Martin, and R. Wilhelm. Applying Compiler Techniques to Cache Behavior Prediction. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, June 1997.

[7] J. Larus. Abstract Execution: A Technique for Efficiently Tracing Programs. *Software - Practice and Experience*, 20(12):1241–1258, Dec. 1990.

[8] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, S.-M. Moon, and C. S. Kim. An Accurate Worst Case Timing Analysis for RISC Processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, July 1995.

[9] Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 298–307, Dec. 1995.

[10] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches. In *Proceedings of the IEEE Real-Time Systems Symposium*, Dec. 1996.

[11] Y.-T. S. Li and S. Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, pages 456–461, June 1995.

[12] S. McFarling. Program Optimization for Instruction Caches. In *Architectural Support for Programming Languages and Operating Systems*, pages 183–191, Boston, Massachusetts, Apr. 1989. Association for Computing Machinery ACM.

[13] A. Mendlson, S. S. Pinter, and R. Shtokhamer. Compile Time Instruction Cache Optimizations. *Computer Architecture News*, 22(1):44–51, Mar. 1994.

[14] F. Mueller, D. B. Whalley, and M. Harmon. Predicting Instruction Cache Behavior. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, 1994.

[15] F. Mueller. Generalizing Timing Predictions to Set-Associative Caches. Technical Report TR 96-66, Institut für Informatik, Humboldt-University, July 1996.

[16] K. Pettis and R. C. Hansen. Profile Guided Code Positioning. In *ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 16–27, White Plains, New York, June 1990.

[17] P. Puschner and C. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Real-Time Systems*, 1:159–176, 1989.

[18] M. Sharir and A. Pnueli. Two Approaches to Interprocedural Data Flow Analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–233. Prentice-Hall, 1981.

[19] J. A. Stankovic. *Real-Time and Embedded Systems.* ACM 50th Anniversary Report on Real-Time Computing Research, 1996.

[20] R. Wilhelm and D. Maurer. *Compiler Design.* International Computer Science Series. Addison-Wesley, 1995. Second Printing.

[21] M. E. Wolf and M. S. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.