Fast and Efficient Cache Behavior Prediction

Christian Ferdinand and Reinhard Wilhelm
Fachbereich Informatik / Universität des Saarlandes
Postfach 15 11 50 / D-66041 Saarbrücken
Fon: 0681 302 3434 Fax: 0681 302 3065
{ferdi,wilhelm}@cs.uni-sb.de
http://www.cs.uni-sb.de/~{ferdi,wilhelm}

1 Real-Time Applications and Modern Hardware Architectures

A *real-time system* is one in which the correctness of the system depends not only on the logical results, but also on the time at which the results are produced. Hard real-time systems are those where it is absolutely imperative that responses occur within the specified deadlines. Examples of areas where real-time systems are used include [10]: process control, nuclear power plants, agile manufacturing, intelligent vehicle highway systems, avionics, air traffic control, telecommunications (the information super highway), multimedia, real-time simulation, virtual reality, medical applications (e.g., telemedicine and intensive care monitoring), and defense applications (e.g., command, control and communications). The market for real-time systems and real-time software is huge, and real-time technology is becoming more and more pervasive, e.g., in a 1997 Opel Omega, 12 microprocessors are used for various functions and many of them have real-time tasks like anti locking system, air bag, and motor control; in a Mercedes-Benz-S-Klasse there are up to 48 microprocessors.

Among the typical application areas are many safety critical, where a failure of the system may lead to severe damage and/or loss of life. Such real-time systems necessitate a timing validation, usually referred to as *schedulability analysis* or *scheduling analysis*. A system is said to be *schedulable* if it can be shown that all timing requirements will be met. A real-time system is often structured as a set of processes with deadlines whereby execution can be distributed over multiple processors. There exist many results and analysis methods for real-time scheduling, but these analysis methods require that the WCET (Worst Case Execution Time) of each task (subtask, critical section, ...) is known.

However, the achievements of modern computer architectures [8] like cache memories and processor pipelines that made possible the tremendous performance increase in recent years complicate the prediction of sharp WCETs. The state of a cache depends on the execution history. This means that the cache behavior of the execution of a reference to an instruction or data could be influenced by instructions that are very far away in the program text, in other modules, in libraries, or even in other programs including the operating system.

In the presence of caches, methods to predict the WCET from execution time measurements of programs or tasks like software monitoring, the dual loop benchmark approach, direct execution time measurement with a logic analyzer, or hardware simulation are not generally applicable as additional instructions to measure the execution time may change the cache behavior and a worst case input that takes the cache behavior into account is usually not known.

Analysis methods that do not consider the cache are not able to provide tight WCET estimations for cached systems. The typical worst case assumption is that all accesses miss the cache[1]. This is an overly pessimistic assumption which leads to a waste of hardware resources in order to guarantee the meeting of all deadlines. This is especially undesirable for mass products like embedded systems in automobiles and mobile phones or systems that require very high computing performance where slight additional computing performance requirements can lead to immense increases in costs.

In this paper, we present an analysis method that is based on the theory of abstract interpretation and is capable of predicting tight bounds on the cache behavior of typical programs.

[1] Hennessy and Patterson [8] describe typical values for first level caches in 1995 workstations: Hit time 1–2 clock cycles (normally 1); Miss penalty 8–66 clock cycles. In more modern CPU designs the miss penalty can be even higher.

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 their effects on cache memories, and we present the *must analysis* that computes a set of memory blocks that are always in the cache and the *may analysis* that computes a set of memory blocks that may be in the cache and describe how the results of the analyses can be interpreted.

The behavior of memory references within loops and recursive procedures can be analyzed with interprocedural analysis methods. In Section 6 existing approaches are discussed and a new approach is presented. An example is given in Section 7. Section 8 introduces an additional improvement and Section 9 describes extensions to data and combined caches.

In Section 10 we present and discuss the results of our practical experiments.

3 Program Analysis by Abstract Interpretation

Program analysis is a widely used technique to determine runtime properties of a given program without actually executing it. Such information is used for example in optimizing compilers [11] to enable code improving transformations.

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

There is a well developed theory of static program analysis called *abstract interpretation* [3]. With this theory, correctness of a program analysis can be systematically 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. 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.

The program analyzer generator PAG [2] offers the possibility to generate a program analyzer from a description of the abstract domain and 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 $n = \mathit{capacity}/\mathit{line\ size}$ blocks.
- *associativity* is the number of cache locations where a particular block may reside.
  $n/\mathit{associativity}$ 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*.

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 behavior of an A-way set associative cache is completely described by the behavior of its n/A fully associative sets[2].

[2]This holds also for direct mapped caches where $A = 1$.

For the sake of space, we restrict our description to the semantics of fully associative caches with LRU replacement strategy. More complete descriptions that explicitly describe direct mapped and A-way set associative caches can be found in [7, 1].

5 Cache Semantics

In the following, we consider a (fully associative) cache as a set of cache lines $L = \{l_1, \ldots, l_n\}$, and the store as a set of memory blocks $S = \{s_1, \ldots, s_m\}$.
To indicate the absence of any memory block in a cache line, we introduce a new element I; $S' = S \cup \{I\}$.

Definition 1 (concrete cache state)
A *(concrete) cache state* is a function $c : L \to S'$.
C_c denotes the set of all concrete cache states.

If $c(l_x) = s_y$ for a concrete cache state c, 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 effect on the cache of referencing the memory. The LRU replacement strategy is modeled by putting the most recently referenced memory block in the first position l_1. If the referenced memory block s_x is in the cache already, then all memory blocks in the cache that have been more recently used than s_x increase their relative age by one, i.e., they are shifted by one position to the next cache line. If the memory block s_x is not in the cache already, then all memory blocks in the cache are shifted and the 'oldest', i.e., least recently used memory block is removed from the cache.

Definition 2 (cache update) A cache update function $\mathcal{U} : C_c \times S \to C_c$ describes the new cache state for a given cache state and a referenced memory block.

Updates of fully associative caches with LRU replacement strategy are modeled as in Figure 1.

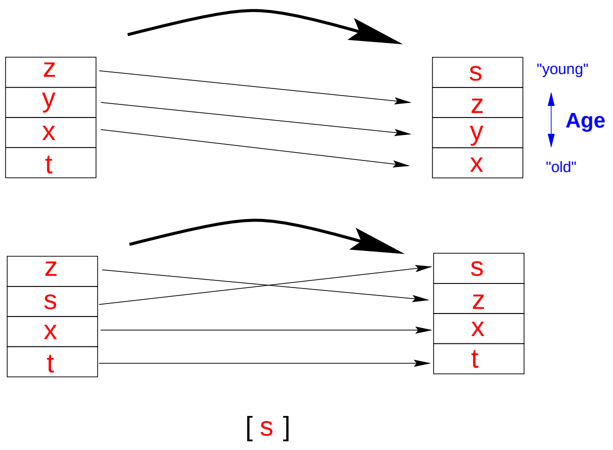


Figure 1: Update of a concrete fully associative (sub-) cache.

5.1 Control Flow Representation

We represent programs by control flow graphs consisting of nodes and typed edges. The nodes represent *basic blocks*[3]. We assume that for each basic block, the sequence of references to memory is known[4], i.e., there exists a mapping from control flow nodes to sequences of memory blocks: $\mathcal{L} : V \to S^*$.

We can describe the working of a cache with the help of the update function $\mathcal{U}$. Therefore, we extend $\mathcal{U}$ to sequences of memory references: $\mathcal{U}(c, \langle s_{x_1}, \ldots, s_{x_y} \rangle) = \mathcal{U}(\ldots(\mathcal{U}(c, s_{x_1}))\ldots, s_{x_y})$.

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

5.2 Abstract Semantics

The domain for our abstract interpretation consists of *abstract cache states*:

Definition 3 (abstract cache state) An *abstract cache state* $\hat{c} : L \to 2^S$ maps cache lines to sets of memory blocks. $\hat{C}$ denotes the set of all abstract cache states.

We will present two analyses. The **must analysis** determines a set of memory blocks that are in the cache at a given program point upon any execution. The **may analysis** determines all memory blocks that may be in the cache at a given program point. The latter analysis is used to guarantee the absence of a memory block in the cache.

The analyses are used to compute a categorization for each memory reference that describes its cache behavior. The categories are described in Table 1.

[3]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. For our cache analysis, it is most convenient to have one memory reference per control flow node. Therefore, our nodes may represent the different fragments of machine instructions that access memory. For non-precisely determined addresses of data references, one can use a set of possibly referenced memory blocks.

[4]This is appropriate for instruction caches and can be too restricted for data caches and combined caches. See [7, 1] for weaker restrictions.

Category	Abb.	Meaning
always hit	ah	The memory reference will always result in a cache hit.
always miss	am	The memory reference will always result in a cache miss.
not classified	nc	The memory reference could neither be classified as ah nor am.

Table 1: Categorizations of memory references.

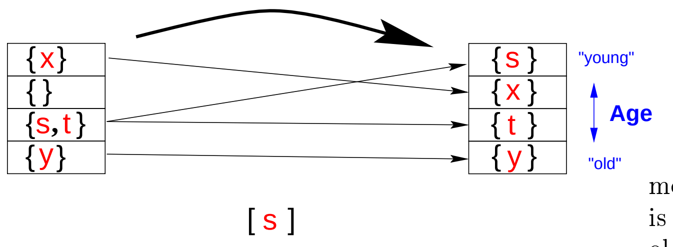


Figure 2: Update of an abstract fully associative (sub-) cache.

The abstract semantic functions describe the effects of a control flow node on an element of the abstract domain. The **abstract cache update** function $\hat{\mathcal{U}}$ for abstract cache states is a canonical extension of the cache update function $\mathcal{U}$ to abstract cache states.

To combine the information from different paths through the control flow graph to a node, *join*-functions are used. They combine the abstract cache states on all control flow nodes with at least two[5] predecessors.

Definition 4 (join function) A *join function* $\hat{\mathcal{J}} : \hat{C} \times \hat{C} \mapsto \hat{C}$ combines two abstract cache states.

5.3 Must Analysis

To determine if a memory block is definitely in the cache we use abstract cache states where the positions of the memory blocks in the abstract cache state are upper bounds of the *ages* of the memory blocks. $\hat{c}(l_x) = \{s_y, \ldots, s_z\}$ means the memory blocks $s_y, \ldots, s_z$ are in the cache. $s_y, \ldots s_z$ will stay in the cache at least for the next $n-x$ references to memory blocks that are not in the cache or are *older* than $s_y, \ldots, s_z$, whereby s_a is older than s_b means: $\exists l_x, l_y : s_a \in \hat{c}(l_x), s_b \in \hat{c}(l_y), x > y$.
We use the abstract cache update function depicted in Figure 2.

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

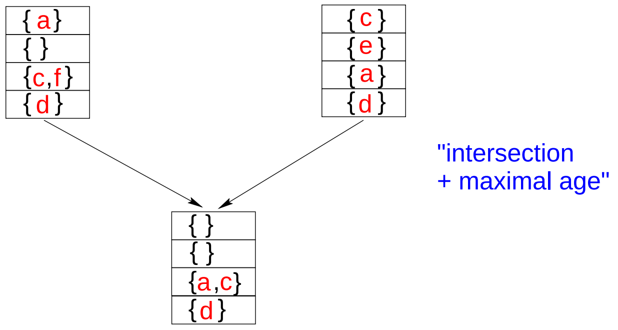


Figure 3: Join for the must analysis.

The join function is similar to set intersection. A memory block only stays in the abstract cache, if it is in both operand abstract caches states. It gets the oldest age, if it has two different ages (see Figure 3).

The solution of the must analysis computed by the PAG generated analyzers is interpreted as follows: Let $\hat{c}$ be an abstract cache state at a control flow node k that references a memory block s_x. If $s_x \in \hat{c}(l_y)$ for a cache line l_y then s_x is definitely in the cache. A reference to s_x is categorized as *always hit* (ah).

5.4 May Analysis

To determine, if a memory block s_x is never in the cache, we compute the set of all memory blocks that *may* be in the cache. We use abstract cache states where the positions of the memory blocks in the abstract cache state are lower bounds of the *ages* of the memory blocks. $\hat{c}(l_x) = \{s_y, \ldots, s_z\}$ means the memory blocks $s_y, \ldots, s_z$ may be in the cache. A memory block $s_w \in \{s_y, \ldots, s_z\}$ will be removed from the cache after at most $n - x + 1$ references to memory blocks that are not in the cache or are *older or the same age* than s_w, if there are no memory references to s_w. s_a is older or same age than s_b means: $\exists l_x, l_y : s_a \in \hat{c}(l_x), s_b \in \hat{c}(l_y), x \geq y$.

We use the following join function: The join function is similar to set union. If a memory block s has two different ages in the two abstract cache states then the join function takes the youngest age (see Figure 4).

The solution of the may analysis computed by the PAG generated analyzers is interpreted as follows: Let $\hat{c}$ be an abstract cache state at a control flow node k that references a memory block s_x. If s_x is not in $\hat{c}(l_y)$ for an arbitrary l_y then it is definitely not in the cache. A reference to s_x is categorized as *always miss* (am).

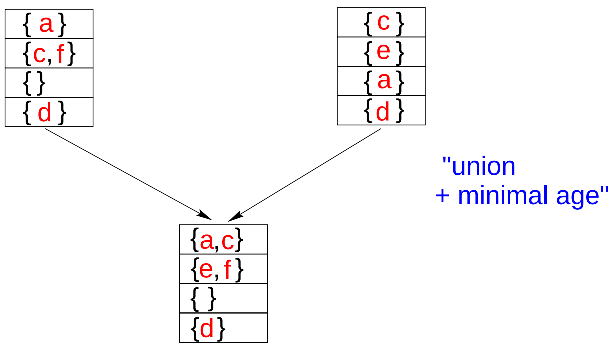


Figure 4: Join for the may analysis.

5.5 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 monotone. This guarantees that our analysis will terminate.

6 Analysis of Loops and Recursive Procedures

Loops and recursive procedures are of special interest, since programs spend most of their runtime there.

A loop often iterates more than once. Since the execution of the loop body usually changes the cache contents, it is useful to distinguish the first iteration from others.

For our analysis of cache behavior we treat loops as procedures to be able to use existing methods for interprocedural analysis (see Figure 5).

		proc $\texttt{loop}_L$();
⋮		
while *P* do		if *P* then
BODY	$\Longrightarrow$	*BODY*
		$\texttt{loop}_L$(); (2)
end;		end
⋮		⋮
		$\texttt{loop}_L$(); (1)
		⋮

Figure 5: Loop transformation.

In the presence of (recursive) procedures, a memory reference can be executed in different execution contexts. An execution context corresponds to a path in the call graph of the program.

The interprocedural analysis methods differ in which execution contexts are distinguished for a memory reference within a procedure. Widely used is the *callstring approach* [9].

The applicability of this approach to cache behavior prediction is limited [7].

To get more precise results for the cache behavior prediction, we have developed the **VIVU approach** which has been implemented with the mapping mechanism of PAG as described in [2]. Paths through the call graph that only differ in the number of repeated passes through a cycle are not distinguished. It can be compared with a combination of *virtual inlining* of all non recursive procedures and *virtual unrolling* of the first iterations of all recursive procedures including loops.

The results of the callstring(0), callstring(1), and the VIVU approach are compared in Section 10.

7 Example

We consider must and may analyses for a fully associative data cache of 4 lines for the following program fragment of a loop, where *..x..* stands for a construct that references variable x:

while *..e..* do *..b..*; *..c..*; *..a..*; *..d..*; *..c..* end

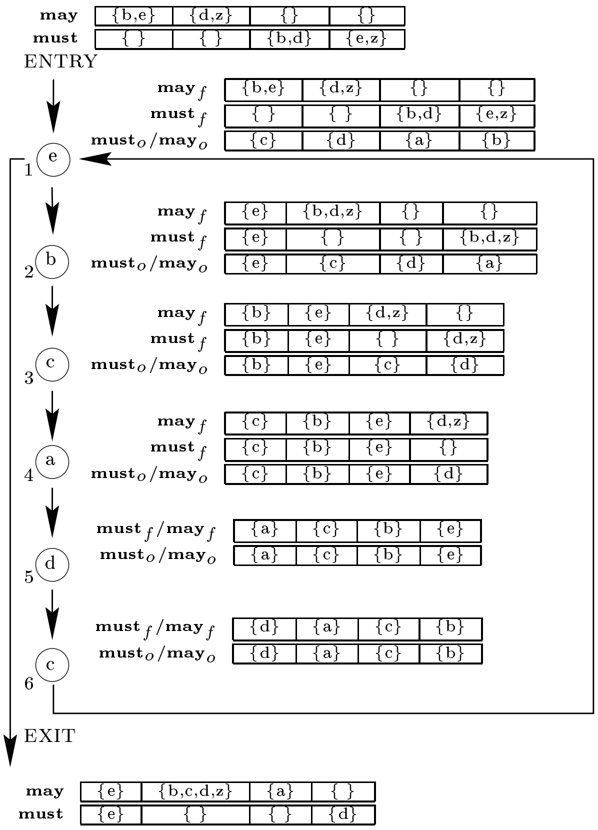
The control flow graph and the result of the analyses with VIVU[6] are shown in Figure 6. We assume that each variable fits exactly into one cache line. The nodes of the control flow graph are numbered 1 to 6, and each node is marked with the variable it accesses. For the analysis, we assume the loop has been implicitly transformed into a procedure according to Figure 5.

Each node is marked with the abstract cache states computed by the PAG–generated analyzer immediately before the abstract cache states are updated according to the memory references. The loop entry edge is marked with the incoming abstract cache states. The loop exit edge is marked with the outgoing abstract cache states.

8 Heuristics to Bound the Number of Cache Misses

For memory references that can neither be classified as always hit nor as always miss one can use a simple heuristics to determine a safe upper bound on the number of cache misses.

[6]Here, the analyses with callstring(1) yield the same results.



(Node,Variable)	first iteration	other iterations
(1,e), (2,b)	always hit	always miss
(3,c)	always miss	always hit
(4,a), (5,d)	always miss	always miss
(6,c)	always hit	always hit

Figure 6: Must and may analysis for a fully associative data cache with VIVU. **must** and **may** are the abstract cache states for the must and the may analysis. $\mathbf{must}_f$ and $\mathbf{may}_f$ are the abstract cache states for the first loop iteration. $\mathbf{must}_o$ and $\mathbf{may}_o$ are the abstract cache states for all other iterations. The interpretation of the abstract cache states is given in the table above.

For each memory reference classified as nc we compute the set of *competing* memory blocks, i.e., the memory blocks that are in the same fully associative set in the abstract cache state of the may analysis. For instance, if the competing memory blocks reside in less than A (= level of associativity) memory blocks, then all references to the memory block in the given context will result in at most one cache miss. Generally, an upper bound for cache misses of the references to the memory block is given by one plus the maximal number of possible sequences of length A of references to pairwise disjoint competing memory blocks. To determine this bound is a nontrivial problem. We use simple heuristics (described in [7]) to compute a safe approximation to the upper bound.

9 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 [11]. 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. [7] 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.

[6] presents an analysis that allows to determine the persistence of memory blocks in the cache. [7] generalizes the persistence analysis to sets of possibly referenced memory locations, e.g., arrays. This generalization determines memory locations that survive in the cache thus providing effective and efficient means to compute an upper bound of the number of possible cache misses. Furthermore, it is examined how data dependence analysis and program restructuring methods to increase data locality can be used to determine worst case bounds on the number of cache misses.

10 Practical Experiments

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

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. A context represents the execution stack, i.e., the function calls and loops along the corresponding path in the control flow graph to the instruction. It is represented as a sequence[7] of first and recursive function calls ($call\_f_f, call\_f_r$) and first and other executions of loops ($loop\_l_f, loop\_l_o$) for the functions f and (virtually) transformed loops l of a program. $INST$ is the set of all instructions $inst$ in a program. $CONTEXT$ is the set of all execution contexts $context$ of a program. IC is the set of all instruction/context pairs ic.

$$\begin{aligned} CONTEXT &= \{call\_f_f, call\_f_r, loop\_l_f, loop\_l_o\}^* \\ IC &= INST \times CONTEXT \\ cat &: IC \rightarrow \{\mathsf{ah}, \mathsf{am}, \mathsf{nc}\} \end{aligned}$$

[7]For callstring(1) the sequence has a maximal length of one. For callstring(0) the sequence is empty.

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. 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 worst case execution profile allows to compute how often 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 7).

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 7). 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) 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.

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

Name	Description	Inst.
`matmult`	50x50 matrix multiplication	154
`ndes`[1]	data encryption	471
`matsum`[1]	100x100 matrix summation	135
`dhry`	Dhrystone integer benchmark	447
`stats`	two arrays sum, mean, variance, standard deviation, and linear correlation	456
`fft`	fast Fourier transformation	1810
`djpeg`	JPEG decompression	1760
`lloops`	Livermore loops in C	5677
`avl2`	inserts and deletes 1000 elements in an AVL tree	614

[1]Worst case input data

Table 2: Test set of C programs with number of instructions.

For the experiments we use parts of the program suites of Frank Müller, the `djpeg` program of Yau-Tsun Steven Li, and some additional programs (see Table 2). For some programs, there exist worst case inputs, 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`, `stats` and `lloops` 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 relatively many occurrences of `nc` in our categorizations.

The executable of `lloops` consists of more than 100 loops that are often deeply nested. This program structure leads to a very high number of distinguished execution contexts with the VIVU approach.

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.

	callstring(0)			callstring(1)			VIVU		
Name	ah	am	nc	ah	am	nc	ah	am	nc
`matmult`	113	15	26	168	25	21	406	40	0
`ndes`	339	14	118	734	36	131	1407	123	39
`matsum`	99	18	18	139	25	13	212	35	0
`dhry`	297	30	120	427	39	140	798	145	136
`stats`	311	16	129	612	26	213	1109	126	197
`fft`	1233	145	432	2212	239	629	19261	1206	5536
`djpeg`	1225	39	496	2297	188	497	65190	6421	5596
`lloops`	3928	22	1727	26750	7099	3470	585994	54221	48156
`avl2`	377	39	198	1112	123	400	2949	287	1290

Table 3: The numbers of occurrences of `ah`, `am`, and `nc` in the categorizations for a 1KB 4-way set associative instruction cache with 16 byte linesize.

Table 3 shows the distribution of `ah`, `am`, and `nc` in the categorizations for the test programs for callstring(0), callstring(1), and VIVU for one selected cache configuration. The sum of `ah`, `am`, and `nc` in the categorizations is the number of distinguished instruction/context pairs. It is a measure for the complexity of the analysis. In our current implementation, the categorization for a given cache configuration can be computed within seconds on a SUN SPARCstation 20 for most of our test programs, but the computation for `lloops` with VIVU requires about 7 minutes. In our implementation, there is room for improvements, though.

To give a more expressive presentation of the results

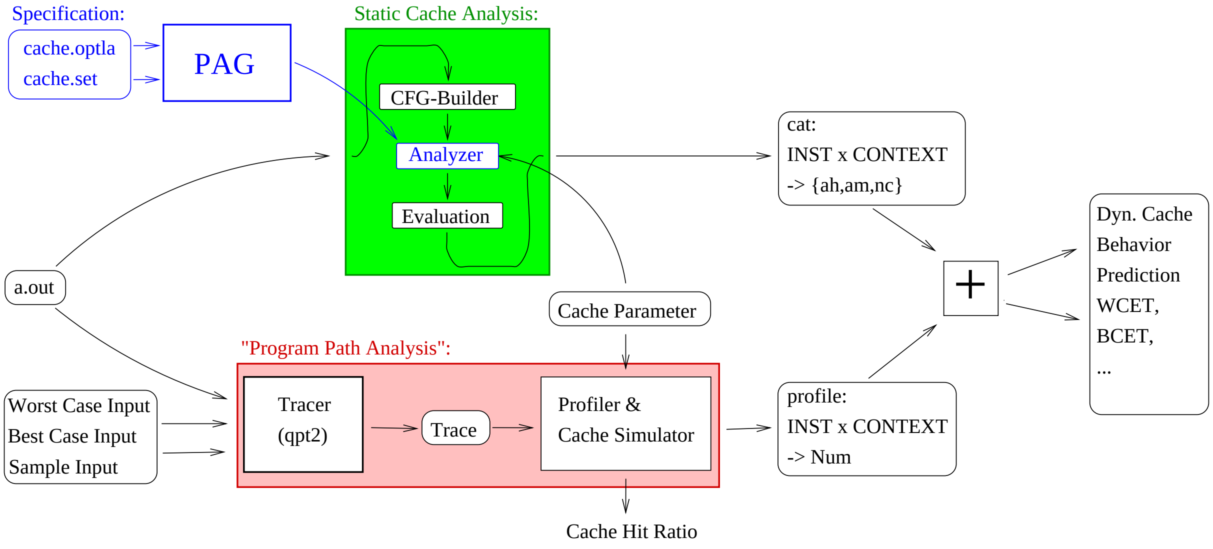


Figure 7: The structure of the analysis.

of our experiments than bounds on cache hit ratios, we assume an idealized virtual hardware that executes all instructions that result in an instruction cache hit in one cycle and all instructions that result in an instruction cache miss in 10.

The cache behavior of the test programs for different cache configurations is computed 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[8] are not traced and their effects on the cache are therefore ignored. From the number of hits and misses in the trace we compute the execution time *ET* of our idealized virtual hardware.

With our categorization and heuristics an upper and a lower bound of the execution time can be computed by combining the profiles with the results of our analyses. An upper bound of the execution time is given if we count all instructions in the profile as misses that cannot be determined from the categorization as cache hits. A lower bound of the execution time is given if we count all instructions in the profile as hits that cannot be determined from the categorization as cache misses. The upper and lower bounds of the test programs for various cache configurations are shown in Figure 8 in percent of the execution time *ET*.
Figure 8 can be interpreted as follows:

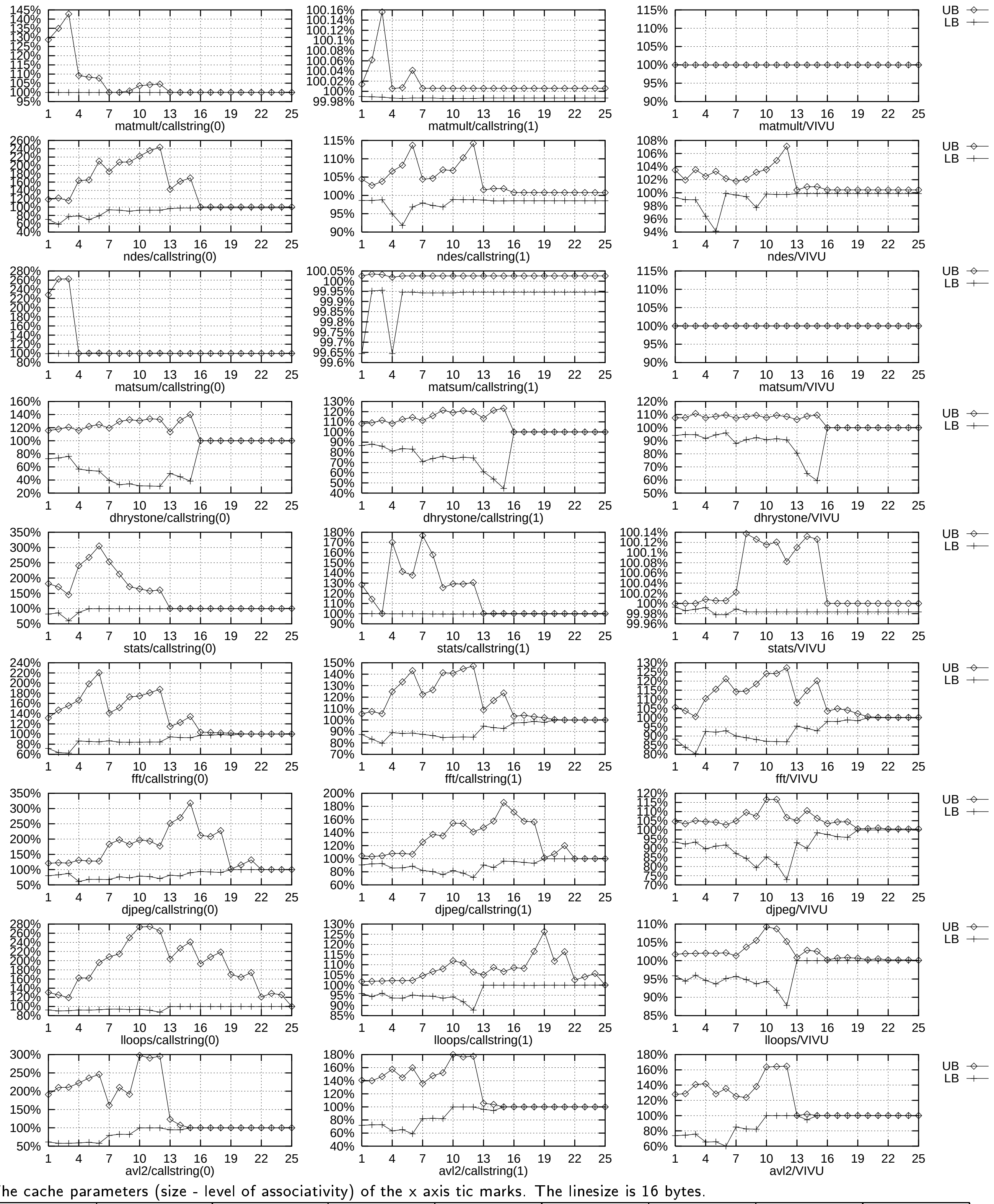
- The VIVU approach generally leads to the most precise predictions.
- Conditionally executed code, e.g. as found in the arithmetic library functions or in `avl2`, can lead to less precise predictions which result from many `nc` in the categorizations.
- There can be a wide variation of the quality of the prediction depending on the cache configuration.
- For all test programs our method (especially with VIVU) gives much better results than naive methods which count all memory references as misses for a WCET estimation, and as hits for a BCET estimation.

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

11 Conclusion

We have described semantics based analysis methods by abstract interpretation that allow to predict the intrinsic behavior of programs for various types of one level caches and processor pipelines. The cache analyzers have been implemented. The applicability of our methods has been shown with the results of our practical experiments. Our approach has many advantages:

- The theory of abstract interpretation supports correctness proofs for the analysis and provides efficient implementation methods.
- The cache analyzers are generated by the program analyzer generator `PAG` from very concise specifications.



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 8: Upper (UB) and lower bounds (LB) for the execution time for different cache parameters in % of execution time for callstring(0), callstring(1), and VIVU.

- It is possible to trade time for precision, but even with the VIVU approach our implementation of the analyses is quite fast.
- The newly developed VIVU approach makes it possible to predict the cache behavior within tight bounds for many programs and cache configurations.
- 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 front ends to our analyzers.
- No special input of a skilled user is required to tune for acceptable results. This makes it feasible to use our analyses in an automatic schedulability analysis.
- The cache and the pipeline analysis can be naturally integrated [7].
- The analyses are extensible to accommodate further cache designs like multilevel caches, wrap around line fill, or pseudo associative caches.

References

[1] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache Behavior Prediction by Abstract Interpretation. In *Proceedings of SAS'96, Static Analysis Symposium*, LNCS 1145, pages 52–66. Springer, Sept. 1996.

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

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

[4] 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*, pages 37–46, June 1997.

[5] C. Ferdinand, F. Martin, and R. Wilhelm. Cache Behavior Prediction by Abstract Interpretation. *Science of Computer Programming*, 1998. Selected for SAS'96 special issue.

[6] C. Ferdinand. A Fast and Efficient Cache Persistence Analysis. Technical Report 10/97, Universität des Saarlandes, Sonderforschungsbereich 124, Aug. 1997.

[7] C. Ferdinand. Cache Behavior Prediction for Real-Time Systems. Dissertation, Universität des Saarlandes, Sept. 1997.

[8] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1996. Second Edition.

[9] 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.

[10] J. A. Stankovic. *Real-Time and Embedded Systems*. ACM 50th Anniversary Report on Real-Time Computing Research, 1996. http://www-ccs.cs.umass.edu/sdcr/rt.ps.

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