Cache Behavior Prediction by Abstract Interpretation

Christian Ferdinand Florian Martin Reinhard Wilhelm
Martin Alt

Universität des Saarlandes, Fachbereich Informatik, Postfach 15 11 50, D-66041 Saarbrücken, Germany, {ferdi,florian,wilhelm,alt}@cs.uni-sb.de

Abstract

Abstract interpretation is a technique for the static detection of dynamic properties of programs. It is semantics based, that is, it computes approximative properties of the semantics of programs. On this basis, it allows for correctness proofs of analyses. It replaces commonly used ad hoc techniques by systematic, provable ones, and it allows the automatic generation of analyzers from specifications as in the Program Analyzer Generator, `PAG`.

In this paper, abstract interpretation is applied to the problem of predicting the cache behavior of programs. Abstract semantics of machine programs are defined which determine the contents of caches. For interprocedural analysis, existing methods are examined and a new approach that is especially tailored for the cache analysis is presented. This allows for a static classification of the cache behavior of memory references of programs. The calculated information can be used to sharpen worst case execution time estimations. It is possible to analyze instruction, data, and combined instruction/data caches for common (re)placement and write strategies. Experimental results are presented that demonstrate the applicability of the analysis.

Keywords: 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 [1]. Caching is more or less used for all general purpose processors,

[1] Hennessy and Patterson [10] describe typical values for caches in 1990 workstations and minicomputers: Hit time 1–4 clock cycles (normally 1); Miss penalty 8–32 clock cycles.

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, e.g. by the compiler [32,8]. This should determine whether all timing constraints can be satisfied. WCET (Worst Case Execution Time) extimations for processes have to be used for this. The degree of success for such a timing validation [31] depends on sharp WCET estimations. There are two components to the prediction of WCETS:

(i) architecture modeling, the determination of how much time it will take to execute an execution path on the target system, and
(ii) program path analysis, the determination of a worst case execution path.

Here, we focus on the first point.

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 we present the *must analysis* that computes for all program points a set of memory blocks that must be in the cache whenever control reaches this point and the *may analysis* that computes a set of memory blocks that may be in the cache. 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 describes extensions to data and combined caches. In Section 10 we present and discuss the results of practical experiments from an implementation of the analyses, and Section 11 describes related work.

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 [33] to enable code improving transformations. A program analyzer takes a program as input and computes some interesting 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. 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* [5–7]. 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 an abstract semantics and an input program, a system of recursive equations can be constructed. The variables in this system stand for the values of the abstract domain at 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. For example, the value after the exit of a loop depends on the value at the end of the loop body and on the value before the loop because it is possible that the loop is never executed. 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 fixpoint 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` [1,2] 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 $n = capacity/line\ size$ blocks.
- *associativity* is the number of cache locations where a particular block may reside. $n/associativity$ is the number of *sets* of a cache. A *set* can be considered as a fully associative subcache.

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* [30].

In the case of an associative cache, a memory block 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*.

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.

5 Cache Semantics

In the following, we consider an A-way set associative cache as a sequence of (fully associative) sets $F = \langle f_1, \ldots, f_{n/A} \rangle$, 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 $set : M \to F$ gives 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 statically be determined as it depends only on the address of the memory block. The dynamic distribution of memory blocks into sets is modeled with 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 with the *set states*.

Definition 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 2 (concrete cache state) A *(concrete) cache state* is a function $c : F \to S$. 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.

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

Definition 4 (cache update) A cache update function $\mathcal{U}_C : C \times M \to C$ 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:

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

5.1 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: $\mathcal{L} : V \to M^*$.

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: $\mathcal{U}_C(c_I, \mathcal{L}(k_1). \ldots .\mathcal{L}(k_p))$.

[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. 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.

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

The domain for our abstract interpretation consists of *abstract cache states* that are constructed from *abstract set states*:

Definition 5 (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 6 (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.

We will present two analyses. The **must analysis** determines a set of memory blocks that are definitely in the cache whenever control reaches a given program point. 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.

Table 1
Categorizations of memory references.

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.

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

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

An abstract cache state $\hat{c}$ describes a set of concrete cache states c, and an abstract set state $\hat{s}$ describes a set of concrete set states s.

To determine if a memory block is definitely in the cache we use abstract set states where 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.

$m_a \in \hat{s}(l_x)$ means that the memory block m_a is in the cache. The position (relative age) 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 will 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* $conc_{\hat{C}} : \hat{C} \rightarrow 2^C$. The concretization function for the must analysis $conc^{\cap}_{\hat{C}}$ is given by:

$$conc^{\cap}_{\hat{C}}(\hat{c}) = \{c \mid \forall 1 \leq i \leq n/A : c(f_i) \in conc^{\cap}_{\hat{S}}(\hat{c}(f_i))\}$$
$$conc^{\cap}_{\hat{S}}(\hat{s}) = \{s \mid \forall 1 \leq a \leq A : \forall m \in \hat{s}(l_a) : \exists b : s(l_b) = m \text{ and } b \leq a\}$$

We use the following abstract set update function:

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

Example 1 ($\hat{\mathcal{U}}^{\cap}_{\hat{S}}$)

	l_1	l_2	l_3	l_4
$\hat{s}$	$\{m_a\}$	$\{\}$	$\{m_b, m_c\}$	$\{m_d\}$
$\hat{\mathcal{U}}^{\cap}_{\hat{S}}(\hat{s}, m_c)$	$\{m_c\}$	$\{m_a\}$	$\{m_b\}$	$\{m_d\}$

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:

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

The join function for abstract set states is similar to set intersection. A memory block only stays in the abstract set state, if it is in both operand abstract set states. It gets the oldest age, if it has two different ages.

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

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

Example 2 ($\hat{\mathcal{J}}^{\cap}_{\hat{S}}$)

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

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

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

An abstract cache state $\hat{c}$ at a control flow node k is interpreted in the following way: Let m a memory block and $\hat{s} = \hat{c}(set(m))$. If $m \in \hat{s}(l_y)$ for a set line l_y then m is definitely in the cache every time control reaches k. Therefore, a reference to m is categorized as *always hit* (ah).

5.4 May Analysis

To determine, if a memory block is never in the cache, we compute the set of all memory blocks that *may* be in the cache. We use abstract set states $\hat{s}$ where the position (the relative *age*) of a memory block in the abstract set

state is a lower bound of the positions (the relative *ages*) of the memory blocks in the concrete set states that $\hat{s}$ represents.

$m_a \in \hat{s}(l_x)$ means the memory blocks m_a may be in the cache. The position (relative age) 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 $y < x$, i.e., memory blocks that are already in the cache and are "younger" as m_a.

If there are no memory references to m_a, then m_a will be removed from the cache after at most $A - x + 1$ references to memory blocks that go into the same set and are not yet in the cache or are *older or the same age* than m_a.

The concretization function for the may analysis $conc^{\cup}_{\hat{C}}$ is given by:

$$conc^{\cup}_{\hat{C}}(\hat{c}) = \{c \mid \forall 1 \leq i \leq n/A : c(f_i) \in conc^{\cup}_{\hat{S}}(\hat{c}(f_i))\}$$
$$conc^{\cup}_{\hat{S}}(\hat{s}) = \{s \mid \forall 1 \leq a \leq A : \forall m \in s(l_a) : \exists b : \hat{s}(l_b) = m \text{ and } b \leq a\}$$

We use the following abstract set update function:

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

Example 3 ($\hat{\mathcal{U}}^{\cup}_{\hat{S}}$)

	l_1	l_2	l_3	l_4
$\hat{s}$	$\{m_a\}$	$\{m_b, m_c\}$	$\{\}$	$\{m_d\}$
$\hat{\mathcal{U}}^{\cup}_{\hat{S}}(\hat{s}, m_c)$	$\{m_c\}$	$\{m_a\}$	$\{m_b\}$	$\{m_d\}$

The abstract cache update function for the may analysis has the same structure as the one for the must analysis:

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

The join function is similar to set union. If a memory block s has two different ages in two abstract cache states then the join function takes the youngest age.

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

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

Example 4 ($\hat{\mathcal{J}}^{\cup}_{\hat{S}}$)

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

The join function for abstract cache states for the may analysis has the same structure as for the the must analysis:

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

An abstract cache state $\hat{c}$ at a control flow node k is interpreted in the following way: Let m be a memory block and $\hat{s} = \hat{c}(set(m))$. If m is not in $\hat{s}(l_y)$ for an arbitrary l_y then it is definitely not in the cache whenever control reaches k. Therefore, a reference to m is categorized as *always miss* (am).

5.5 Termination of the Analysis

There are only a finite number of sets and set lines and for each program a finite number of memory blocks. This means the domain of abstract cache states $\hat{c} : F \to (L \to 2^{M'})$ is finite. Hence, every ascending[5] 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.

[5] The order is given by set inclusion and the concretization functions.

6 Analysis of Loops and Recursive Procedures

Loops and recursive procedures are of special interest, since programs spend most of their time there. In a control flow graph, a loop is represented as a cycle. The *start node* of a loop [6] has two incoming edges. One represents the entry into the loop, the other represents the control flow from the end of the loop to the beginning of the loop. The latter is called *loop edge* (see Figure 1).

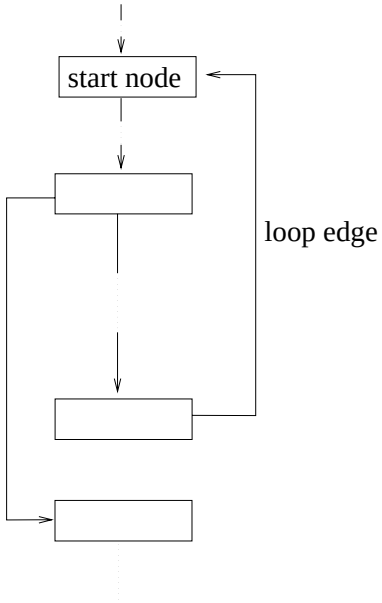

Fig. 1. Control flow graph of a loop.

There are loops that can iterate 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. This could be achieved by conceptually unrolling each loop once.

Example 5 Let us consider a sufficiently large fully associative data cache with LRU replacement strategy and the following program fragment:

```
...
/* Variable x not in the data cache */
for i:=1 to .. do  ... y:=x  ... end
...
```

[6] We consider here loops that correspond to the loop constructs of 'higher programming languages'. Program analysis is not restricted to this, but will produce more precise results for programs with well behaved control flow.

In the first execution of the loop, the reference to `x` will be a cache miss, because `x` is not in the cache. In all further iterations the reference to `x` will be a cache hit, if the cache is sufficiently large to hold all variables referenced within the loop.

For the abstract interpretation, the join function $\hat{\mathcal{J}}^{\cap}$ combines the abstract cache states at the start node of the loop. Since the join function is 'similar' to set intersection, the combined abstract cache state will never include the variable `x`, because `x` is not in the abstract cache state before the loop is entered. For a WCET computation for a program this is a safe approximation, but nevertheless not very good.

Loop unrolling would overcome this problem. After the first unrolled iteration, `x` would be in the abstract cache state and would be classified as always hit.

For our analysis of cache behavior we treat loops as procedures to be able to use existing methods for the interprocedural analysis [7] . This is done by transforming all loops into "loop-procedures" in the control flow graph according to Figure 2. This is only done for the analyses and has no influence on the program code.

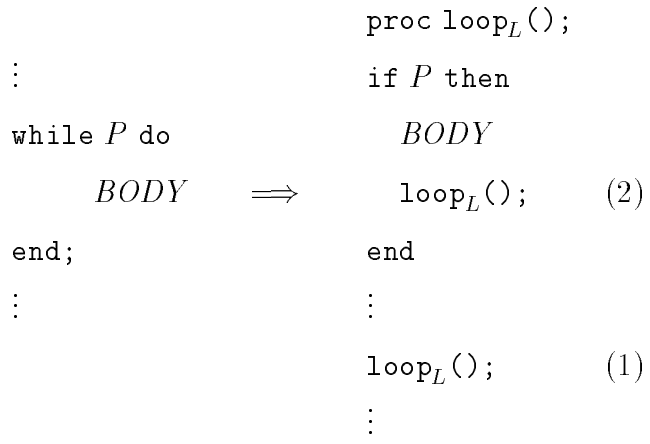

Fig. 2. Loop transformation.

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 are the *callstring approach* and the *functional approach* which have been proposed by Sharir and Pnueli [29] and are implemented in `PAG`.

[7] Ludwell Harrison `III` [9] also proposed this transformation for the analysis of loops.

The callstring approach limits the number of distinguished execution contexts statically. To do this the call graph is considered. The goal is, not to merge information that is obtained on different paths through the graph. But in presence of recursion, the graph is cyclic and therefore has an infinite number of paths. So only the information obtained on paths which differ in suffixes of a fixed length K are kept separated.

In the functional approach, the number of distinguished execution contexts is not statically limited. The PAG generated analyzer tabulates all different input values and output values of the abstract domain (here: abstract cache states) for every procedure. To guarantee termination of the analysis, the abstract domain has to be finite. The functional approach computes the most precise solution.

The applicability of these approaches to the cache behavior prediction is limited:

- **Callstring approach:** If we restrict the callstring length K to 0 (*callstring*(0)), then one categorization for each memory reference in the program is computed. This is fast, but yields not very precise information.

  Callstring(1) gives better results, as it distinguishes as many different execution contexts of a memory reference in a procedure as there are calls. For each transformed loop there is one call to the loop–procedure at the original place of the loop in the program (1) (see Figure 2) and one for the recursive call of the loop-procedure (2). The first call corresponds to the first iteration of the loop. The second call corresponds to all other iterations of the loop.

  Longer callstrings increase the analysis effort and lead to a more precise categorization. The precision gained is quite poor with respect to the enormously increasing analysis costs, as there are many execution contexts distinguished that are "non interesting" for our analysis.
- **Functional approach:** The dynamically distinguished execution contexts cannot be easily combined with the results of a program path analysis that determines a safe approximation to the worst case execution path. This makes a WCET estimation more difficult.

To overcome the deficiencies of the callstring(>1) and the functional approaches, we have developed the **VIVU approach** which has been implemented with the mapping mechanism of PAG as described in [1]. It corresponds to callstring(∞), but 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 [8] and *virtual unrolling* of the first iterations of all recursive proce-

[8] This has also been used in [22,15].

dures (and loop-procedures). The results of the VIVU approach can naturally be combined with the results of a path analysis to predict the WCET of a program.

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 [9] are shown in Figure 3. We assume that all variables are stored in pairwise different memory blocks. 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 loop-procedure according to Figure 2.

Each node is marked with the abstract cache states (in the same format as in Example 1) 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 Data Caches and Combined Caches

Our analysis can be used to predict the behavior of data caches or combined instruction/data caches, if the addresses of referenced data can be statically computed.

Addresses of references to global data can usually be easily determined. Local variables and procedure parameters that are allocated on the stack are addressed relatively to the stack pointer or frame pointer, i.e., a register that points to a known address within the procedure frame on the execution stack. If the values of the stack pointer or frame pointer are known, the absolute addresses of the variables and parameters can be determined by a data flow

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

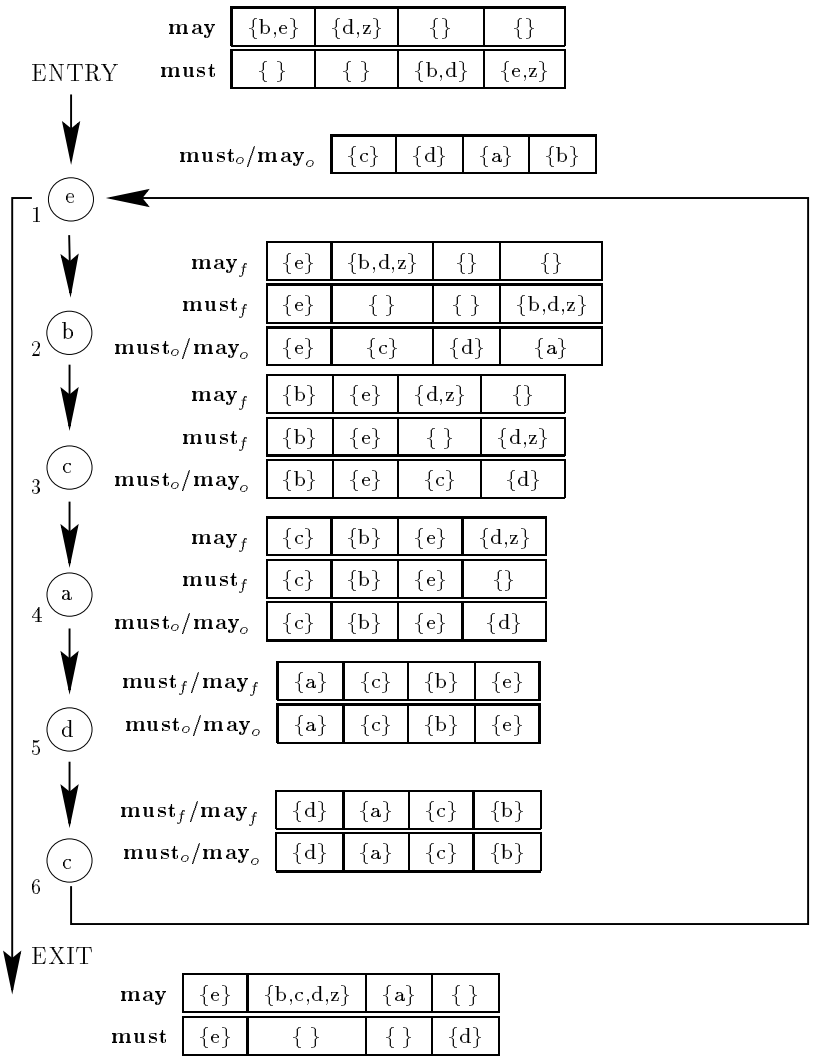

Fig. 3. Must and may analysis for a fully associative data cache with VIVU. **must** and **may** are the abstract cache states for the must and the may analysis. **must**$_f$ and **may**$_f$ are the abstract cache states for the first loop iteration. **must**$_o$ and **may**$_o$ are the abstract cache states for all other iterations. The abstract cache states can be interpreted for each variable reference as follows:

(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

analysis [12]. For programs without recursive procedures, it is possible to determine all values of the stack or frame pointers for all procedures for the distinguished execution contexts of the cache behavior analysis.

To support the analysis of programs for which not all addresses of the memory references can precisely be determined, the $\hat{\mathcal{U}}$ functions are extended to handle a set of possibly referenced memory locations [10].

Since it is not definitely known which memory block is put into the cache, the update function $\hat{\mathcal{U}}^{\cap}_{\hat{C}}$ for the must analysis applied to a set of possible memory locations $\{m_1, ..., m_x\}$ and an abstract cache state $\hat{c}$ only affects the ages of the memory blocks in $\hat{c}$ in all sets where an element of $\{m_1, ..., m_x\}$ could be stored:

$$\hat{\mathcal{U}}^{\cap}_{\hat{C}}(\hat{c}, \{m_1, ..., m_x\}) = \hat{c}[f_i \mapsto \hat{\mathcal{U}}^{\cap}_{\hat{S}}(\hat{c}(f_i), X_{f_i}) \text{ for all } f_i \in \{set(m_1), ..., set(m_x)\}]$$
$$\text{where } X_{f_i} = \{m_y \mid m_y \in \{m_1, ..., m_x\} \text{ and } set(m_y) = f_i\}$$

$$\hat{\mathcal{U}}^{\cap}_{\hat{S}}(\hat{s}, \{m_1, ..., m_x\}) = \hat{\mathcal{U}}^{\cap}_{\hat{S}}(\ldots \hat{\mathcal{U}}^{\cap}_{\hat{S}}(\hat{s}, \{m_1\}) \ldots, \{m_x\})$$

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

The update function $\hat{\mathcal{U}}^{\cup}_{\hat{C}}$ for the may analysis applied to a set $\{m_1, ..., m_x\}$ of possible memory locations and an abstract cache state $\hat{c}$ inserts all elements of $\{m_1, ..., m_x\}$ into their corresponding sets. The ages of the memory blocks that are already in $\hat{c}$ are not changed, because it is not known which set of

[10] References to an array X can be treated conservatively by using a reference to the set $\{m_1, ..., m_x\}$ of all memory blocks of X.

the concrete cache is touched:

$$\hat{\mathcal{U}}^{\cup}_{\hat{C}}(\hat{c}, \{m_1, ..., m_x\}) = \hat{c}[f_i \mapsto \hat{\mathcal{U}}^{\cup}_{\hat{S}}(\hat{c}(f_i), X_{f_i}) \text{ for all } f_i \in \{set(m_1), ..., set(m_x)\}]$$
$$\text{where } X_{f_i} = \{m_y \mid m_y \in \{m_1, ..., m_x\} \text{ and } set(m_y) = f_i\}$$

$$\hat{\mathcal{U}}^{\cup}_{\hat{S}}(\hat{s}, \{m_1, ..., m_x\}) = [l_1 \mapsto l_1 \cup \{m_1, ..., m_x\}, \; l_i \mapsto \hat{s}(l_i) - \{m_1, ..., m_x\} \mid i = 2 \ldots A];$$

9 Writes

So far, we have ignored writing to a cache and only considered reading from a cache. There are two common cache organizations with respect to writing to the cache [10]:

- *Write through:* On a cache write the data is written to both the memory block and the corresponding set line.
- *Write back:* The data is written only to the set line. The modified set line is written to main memory only when it is replaced. This is usually implemented with a bit (called *dirty bit*) for each set line that indicates whether the set line has been modified.

The execution time of a store instruction often depends on whether the memory block that is written is in the cache (*write hit*) or not (*write miss*). For the prediction of hits and misses we can use our analysis. There are two common cache organizations with respect to write misses:

- *Write allocate:* The block is loaded into the cache. This is generally used for write back caches.
- *No write allocate:* The block is not loaded into the cache. The write changes only the main memory. This is often used for write through caches.

Writes to write through/write allocate caches can be treated as reads for the cache analysis. For no write allocate caches, a write access to a block m is treated as a read access, if m is already in the concrete or abstract cache state. Otherwise, the write access is ignored.

Write back caches write a modified line to memory when the line is replaced. The timing of a load or store instruction may depend on whether a modified or

an unmodified line is replaced [11] . To keep track of modified set lines, we extend the cache states by a 'dirty' bit, i.e., we use pairs (m, b) of memory blocks and dirty bits instead of memory blocks in the set/cache states, where $b = \mathbf{d}$ means modified, $b = \mathbf{p}$ means unmodified. The update functions distinguish *reads* and *writes*. The dirty bit is set to **d** on writes, and to **p** on reads. The join function for the must analysis sets the dirty bit for a memory block to **d**, only if it is set to **d** in both arguments. The join function for the may analysis sets the dirty bit for a memory block to **d**, if it is set to **d** in at least one argument.

Let k be a control flow graph node, m be a memory reference at k, $\hat{c}_1^{\cup}$ the abstract cache state for the *may* analysis immediately before m is referenced, and $\hat{c}_2^{\cup}$ the abstract cache state immediately after m was referenced, $\hat{c}_1^{\cap}$ the abstract cache state for the *must* analysis immediately before m is referenced, and $\hat{c}_2^{\cap}$ the abstract cache state immediately after m was referenced.

If the memory reference to m cannot be classified as always hit, then all dirty memory blocks that may have been replaced by the memory reference to m are contained in:

$$Rep = \left\{ m \mid (m, \mathbf{d}) \in \bigcup_{i=1}^{n/A} \bigcup_{j=1}^{A} \hat{c}_1^{\cup}(f_i)(l_j) \right\} - \left\{ m \mid (m, b) \in \bigcup_{i=1}^{n/A} \bigcup_{j=1}^{A} \hat{c}_2^{\cap}(f_i)(l_j) \right\}$$

- If the memory reference to m has been classified as always hit or $Rep = \emptyset$, then no dirty memory block has been replaced. This reference has definitively caused **no write back**.
- If $Rep \neq \emptyset$, then we have to consider a possible write back.
- If there is a $(m, \mathbf{d})$ pair in $\hat{c}_1^{\cap}$ that is not in $\hat{c}_2^{\cup}$, then a dirty memory block has been replaced. This reference has definitively caused a **write back**.

The identified (possible) write backs can be used in another abstract interpretation similar to the cache analysis for the prediction of the write buffer behavior.

10 Practical Experiments

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

[11] Many cache designs use write buffers that hold a limited number of blocks. Write buffers may delay a cache access, when they are full or data is referenced that is still in the buffer. To analyze the behavior of the write buffers possible 'write backs' have to be determined.

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 call graph. It is represented as a sequence [12] of first and recursive function calls ($call\_f_f$, $call\_f_r$) and first and other execution of loops ($loop\_l_f$, $loop\_l_o$) for the functions f and (conceptually) 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*.

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

Additionally, we compute for every instruction/context pair *ic* with $cat(ic) =$ nc the set of *competing* instructions, i.e., the instructions that are in the same fully associative set in the abstract cache state of the may analysis. For instance, if the competing instructions reside in less than A (= level of associativity) memory blocks, then all executions of the instruction will result in at most one cache miss. Generally, an upper bound of the number of cache misses of the instruction is given by one plus the maximal number of possible sequences of length A of executions of competing instruction that are stored in pairwise disjoint memory blocks. To determine the bound is a nontrivial problem. We use simple heuristics to compute a safe approximation to the upper bound.

Our experiments have been performed for the Sun SPARC architecture. The Sun SPARC is a RISC architecture with pipelined instruction execution. It has a uniform instruction size of four bytes. The front end to the analyzer reads a Sun SPARC executable in a.out format. Our implementation is based on the EEL library [13] of the Wisconsin Architectural Research Tool Set (WARTS). EEL (Executable Editing Library) is a C++ library for building tools to analyze and modify an executable (compiled) program. It hides system-specific detail (like executable file format) and allows to edit linked executables, not just object files.

The objective of our work is to improve the WCET estimation of programs on computer systems with caches. The execution time of a program depends on the program path, i.e., the sequence of instructions that are executed and their individual execution times. But the program path is usually dependent on the program input and cannot generally be determined in advance. Therefore, a

[12] For callstring(K) the sequence has a maximal length of K.

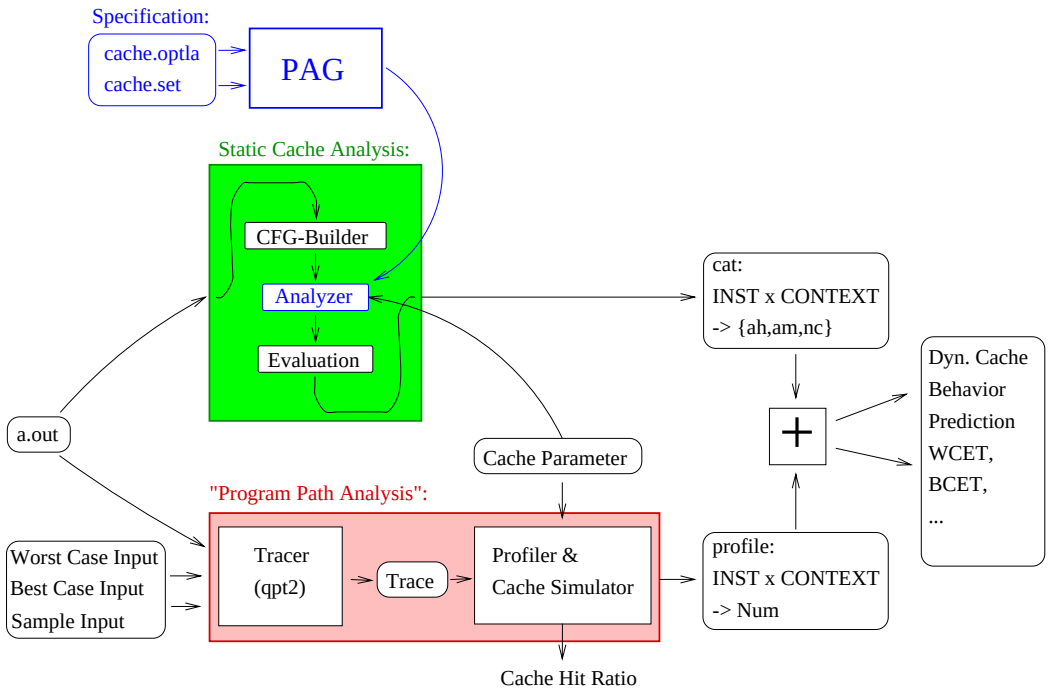

Fig. 4. The structure of the analysis.

program path analysis is part of a WCET analysis [27,17,14,15]. 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 [14]. 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 4).

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 4). 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.

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

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

Name	Description	Inst.
`matmult`	50x50 matrix multiplication	154
`ndes`[1]	data encryption	471
`matsum`[1]	100x100 matrix summation	135
`dhry`	Dhrystone integer benchmark	447
`fdct`[2]	JPEG forward discrete cosine transform	370
`stats`	two arrays sum, mean, variance, standard deviation, and linear correlation	456
`fft`	fast Fourier transformation	1810
`djpeg`[2]	JPEG decompression (128x96 color image)	1760
`lloops`	Livermore loops in C	5677
`avl2`	inserts and deletes 1000 elements in an AVL tree	614

[1] Worst case input data

[2] Random input data

For the experiments we use parts of the program suites of Frank Müller [3,22], the `djpeg` and `fdct` program of Yau-Tsun Steven Li [16], and some additional programs (see Table 2). 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`, `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 in deeply nested loop nests. 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 rebalancing. The code of the insert and delete operations consists of many cases for the different rebalancing operations called rotations. Such a program structure seems to be rather typical for the handling of many dynamic data structures.

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.

	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
fdct	277	9	84	617	93	0	617	93	0
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 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 of our experiments than bounds on cache hit ratios, we assume an idealized 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 cycles[13].

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[14] 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 hardware.

[13] These are the same parameters as used in [21].
[14] In our case, these are the calls to IO routines and timers.

With our categorization an upper and a lower bound of the execution time can be computed by combining the profiles with the results of our analysis. 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 Figures 5 and 6 in percent of the execution time *ET* (the meaning of the x axis tic marks is given in Table 4).

Table 4
The cache parameters (size - level of associativity) of the x axis tic marks of Figures 5 and 6. 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

Figures 5 and 6 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 the naive methods that counts all memory references as misses for a WCET estimation, and as hits for a BCET estimation.

11 Related Work

The computation of WCETs for real-time programs is an ongoing research activity. Park and Shaw [24] describe a method to derive WCETs from the structure of programs. In [27], Puschner and Koza propose methods to guide the computation of WCETs by user annotations like maximal loop counts. This approach seems to be commonly used in WCET analysis tools. Both approaches do not take cache behavior into account.

The possibilities to use optimizing compilers to improve cache performance of programs has extensively been studied [18,19,25,26,34]. But all the proposed

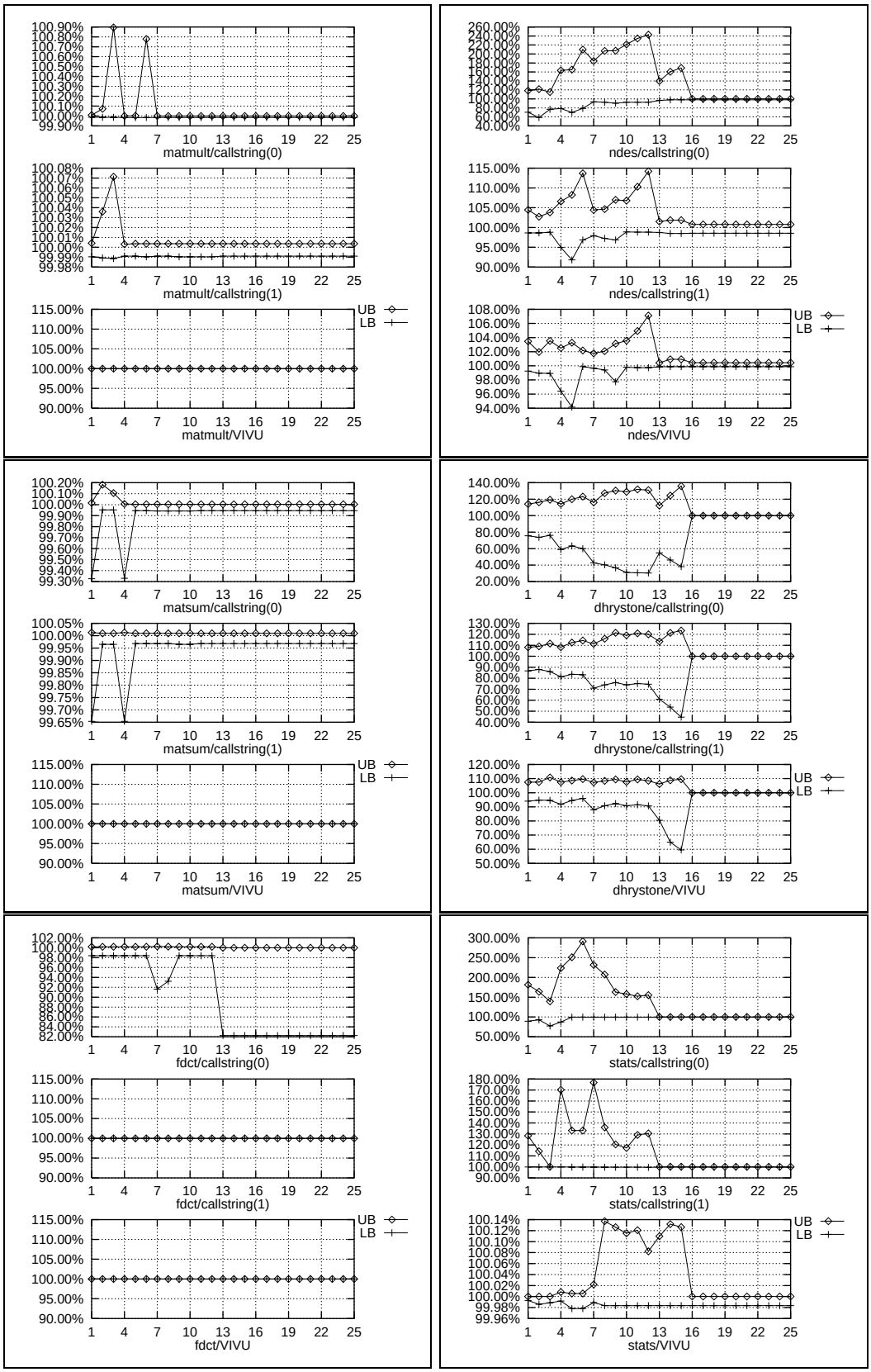

Fig. 5. Upper (UB) and lower bounds (LB) for the execution time for different cache parameters in % of execution time for callstring(0), callstring(1), and VIVU.

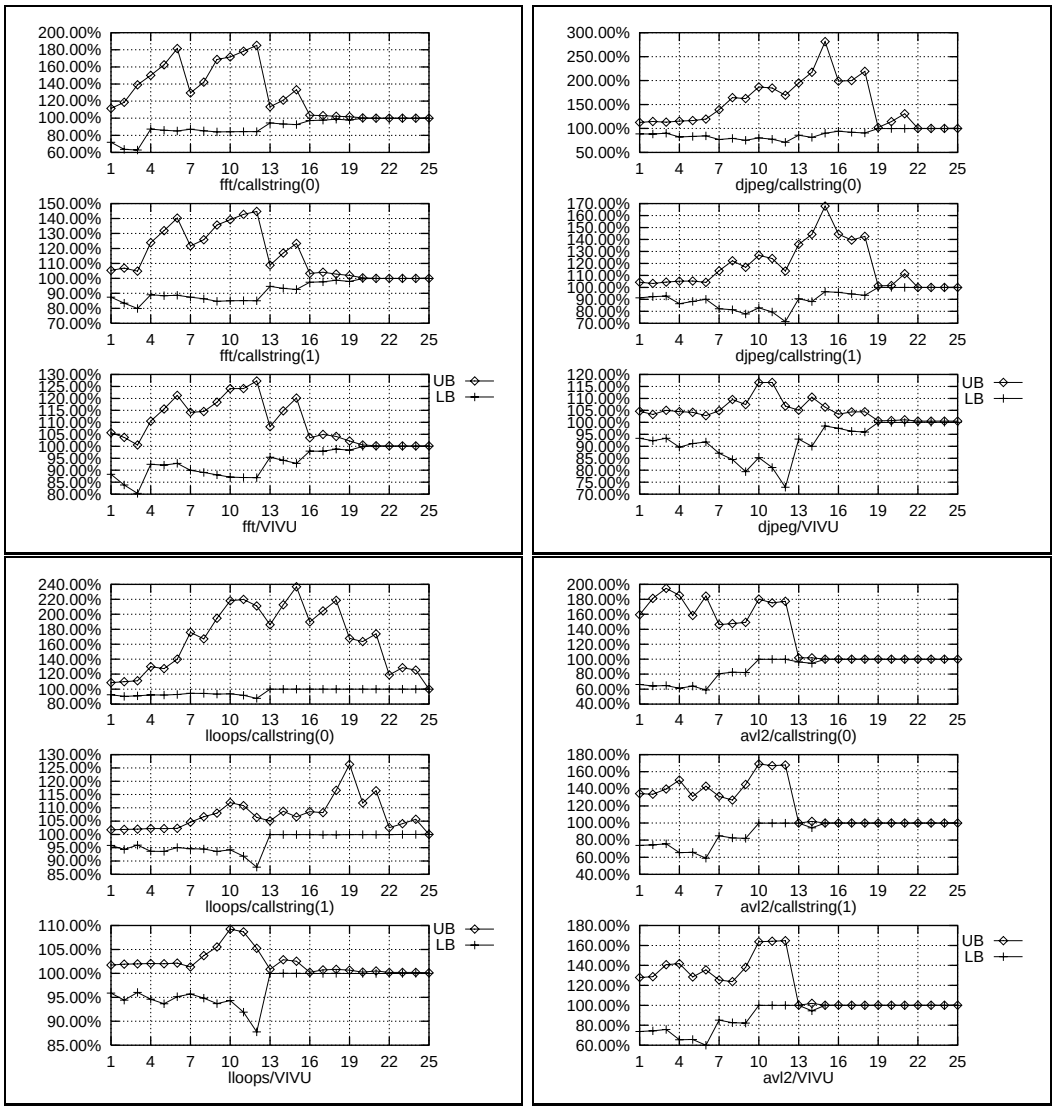

Fig. 6. Upper (UB) and lower bounds (LB) for the execution time for different cache parameters in % of execution time for callstring(0), callstring(1), and VIVU.

program transformations and code reorganizations do not necessarily help in computing the worst case execution time of a program.

An overview of 'Cache Issues in Real-Time Systems' is given in [4]. We restrict our examination here to the intrinsic cache behavior.

The work of Arnold, Müller, Whalley, and Harmon has been one of the starting points of our work. [22,20] 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 [21]. Two data flow analyses are used. The result of the first corresponds to the result of our may analysis. The second is only required for set associative caches for the categorization of instructions within loops. It corresponds to the first analysis

whereby the loop back edges are deleted in the control flow graph. 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. The distinction of a first or a further execution of a loop is not explicit but expressed by the classifications *first miss* and *first hit*. For a set of small programs the same or slightly worse upper bounds of the execution time than our results are reported in [21][15]. But the assessment is difficult as the environment for the experiments is not the same, e.g., different compilers have been used to compile the test programs.

In [15,16] 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[16]. Unlike the method described in [22] 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 [16]. 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 [17], 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.

The approach of Lim et al. has also been applied to data caches. In [11], Hur et al. treat references to unknown addresses as two cache misses. The reported

[15] For the sake of space, the results of not all programs could be reported here.
[16] See http://www.ee.princeton.edu/~yauli/cinderella-3.0/

results are worse than the ones without data cache analysis where one assumes one cache miss for every data reference. But the authors expect that the results improve with better methods to resolve addresses of data references. For loops that reference only data that fit entirely into the cache, Kim et al. [12] have improved the approach based on the *pigeonhole principle*. Applied to the cache analysis, the pigeonhole principle says: If we have n memory reference to m memory locations and $n > m$ and all referenced memory blocks fit into the cache, then there must inevitably some cache hits.

A method for the data cache analysis by graph coloring is described in [23,28]. Similar to the Chow-Hennessy register allocator, variables are allocated to cache lines. The objective of the analysis is to show that throughout the live range of a cache line, no other memory access interferes with this particular cache line. This approach has limited success even for small programs.

12 Conclusion and Future Work

We have described semantics based analysis methods 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 implementation methods.

The analyzers are generated by the program analyzer generator `PAG` from very concise specifications. It is possible to trade time for precision, but even with the VIVU approach our implementation of the analyses is quite fast. 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. 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. The analyses are extensible to accommodate further cache designs like multilevel caches or wrap around line fill.

Future work includes the integration of our tool with a program path analysis. We are working on extension to predict the pipeline behavior of processors.

The pipeline analyzers will be generated from a description similar to the specifications used for the generation of code schedulers. For the analysis of array references, there exist methods based on data dependency analysis which should be combined with our approach. Finally, we will explore methods that allow to combine the separated analyses of modules, libraries, or operating systems calls and thereby support the modularization of the analysis.

Acknowledgement

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` front end for SPARC executables, Yau-Tsun Steven Li and Frank Müller for providing their benchmark programs, and the latter for fruitful discussions.

References

[1] 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.

[2] M. Alt, F. Martin, and R. Wilhelm. Generating Dataflow Analyzers with PAG. Technical Report A10-95, Universität des Saarlandes, 1995.

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

[4] S. Basumallick and K. Nilsen. Cache Issues in Real-Time Systems. In *Proceedings of the 1994 ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, June 1994.

[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] P. Cousot and R. Cousot. Static Determination of Dynamic Properties of Generalized Type Unions. In *Proceedings of an ACM Conference on Language Design for Reliable Software*, volume 12(3), pages 77–94, Mar. 1977.

[7] P. Cousot and R. Cousot. Static Determination of Dynamic Properties of Recursive Procedures. *Formal Description of Programming Concepts*, pages 237–277, 1978.

[8] W. A. Halang and K. M. Sacha. *Real-Time Systems*. World Scientific, 1992.

[9] L. Harrison. *Personal communication on Abstract Interpretation, Dagstuhl Seminar*, 1995.

[10] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[11] Y. Hur, Y. H. Bea, S.-S. Lim, B.-D. Rhee, S. L. Min, Y. C. Park, M. Lee, H. Shin, and C. S. Kim. Worst case timing analysis of RISC processors: R3000/R3010 case study. In *IEEE Real-Time Systems Symposium*, pages 308–319, Dec. 1995.

[12] S. Kim, S. Min, and R. Ha. Efficient worst case timing analysis of data caching. In *IEEE Real-Time Technology and Applications Symposium*, June 1996.

[13] J. R. Larus. *EEL Guts: Using the EEL Executable Editing Library*. Computer Sciences Department, University of Wisconsin-Madison, 1996.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] F. Mueller. Static Cache Simulation and its Applications. Phd thesis, Florida State University, July 1994.

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

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

[23] K. D. Nilsen and B. Rygg. Worst-Case Execution Time Analysis on Modern Processors. In *Proceedings of the 1995 ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, June 1995.

[24] C. Y. Park and A. C. Shaw. Experiments with a Program Timing Tool Based on Source-Level Timing Schema. *IEEE Computer*, 25(5):48–57, May 1991.

[25] 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.

[26] A. K. Porterfield. Software Methods for Improvement of Cache Performance on Supercomputer Applications. Phd thesis, Rice University, May 1989.

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

[28] J. Rawat. Static Analysis of Cache Performance for Real-Time Programming. Masters thesis, Iowa State University, May 1993.

[29] 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.

[30] A. Smith. Cache Memories. *ACM Computing surveys*, 14(3):473–530, Sept. 1983.

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

[32] A. D. Stoyenko, V. C. Hamacher, and R. C. Holt. Analyzing Hard-Real-Time Programs For Guaranteed Schedulability. *IEEE Transactions on Software Engineering*, 17(8), Aug. 1991.

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

[34] M. E. Wolf and M. S. Lam. A Data Locality Optimizing Algorithm. *SIGPLAN Notices*, 26(6):30–44, June 1991. Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.