Applying Compiler Techniques to Cache Behavior Prediction

Christian Ferdinand, Florian Martin, and Reinhard Wilhelm
Universität des Saarlandes / Fachbereich Informatik
Postfach 15 11 50 / D-66041 Saarbrücken / Germany
{ferdi,florian,wilhelm}@cs.uni-sb.de

Abstract

In previous work [1], we have developed the theoretical basis for the prediction of the cache behavior of programs by abstract interpretation. Abstract interpretation is a technique for the static analysis 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 thus 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 semantics of machine programs are refined which determine the contents of caches. For interprocedural analysis, existing methods are examined and a new approach that is especially tailored for the analysis of hardware with states 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.

1 Cache Memories and Real-Time Applications

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

Programs with hard real-time constraints have to be subjected to a schedulability analysis. It has to be determined whether all timing constraints can be satisfied. The degree of success for such a timing validation [12] depends on sharp WCET (Worst Case Execution Time) estimations. For example for hardware with caches, the typical worst case assumption is that all accesses miss the cache. This is an overly pessimistic assumption which leads to a waste of hardware resources.

2 Overview

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

Cache memories are briefly described in Section 4. In Section 5 we give a semantics for programs that reflects only memory accesses (to fixed addresses) and its effects on cache memories, and we present the *must analysis* that computes a set of memory blocks that must be 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 describes extensions to data and combined caches.

In Section 9 we present and discuss the results of our practical experiments, and Section 10 describes related work.

3 Program Analysis by Abstract Interpretation

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 [13] 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* [4]. 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 are the values of the abstract domain for every program point. In this equation system, the value at a program point depends on the values at all program points which can directly precede the execution of this program point. 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 fix-point iteration if the abstract domain has only finite ascending chains, i.e., every chain of values $v_1 \sqsubset v_2 \sqsubset \cdots$ has only finite length, and if in addition every semantic function is monotonic.

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

4 Cache Memories

A cache can be characterized by three major parameters:

- *capacity* is the number of bytes it may contain.
- *line size* (also called block size) is the number of contiguous bytes that are transferred from memory on a cache miss. The cache can hold at most $n = capacity/line\ size$ blocks.
- *associativity* is the number of cache locations where a particular block may reside.
  n/associativity is the number of *sets* of a cache.

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[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 [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 $c : L \rightarrow S'$.
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 $\mathcal{U} : C_c \times S \rightarrow C_c$ 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 in the following way:

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

$$\mathcal{U}(c, s_x) = \begin{cases} [l_1 \mapsto s_x, \\ \ l_i \mapsto c(l_{i-1}) \mid i = 2 \ldots h, \\ \ l_i \mapsto c(l_i) \mid i = h+1 \ldots n]; \\ \qquad \qquad \text{if } \exists l_h : c(l_h) = s_x \\ [l_1 \mapsto s_x, \\ \ l_i \mapsto c(l_{i-1}) \text{ for } i = 2 \ldots n]; \\ \qquad \qquad \qquad \text{otherwise} \end{cases}$$

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 $f(x)$.

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

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

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 the 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 under all circumstances. 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.

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.

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.

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

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

5.1 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 following abstract cache update function:

$$\hat{\mathcal{U}}^{\cap}(\hat{c}, s_x) = \hat{c}'$$

$$\hat{c}' = \begin{cases} [l_1 \mapsto \{s_x\}, \\ \ l_i \mapsto \hat{c}(l_{i-1}) \mid i = 2 \ldots h-1, \\ \ l_h \mapsto \hat{c}(l_{h-1}) \cup (\hat{c}(l_h) - \{s_x\}), \\ \ l_i \mapsto \hat{c}(l_i) \mid i = h+1 \ldots n]; \\ \qquad \qquad \qquad \text{if } \exists l_h : s_x \in \hat{c}(l_h) \\ [l_1 \mapsto \{s_x\}, l_i \mapsto \hat{c}(l_{i-1}) \text{ for } i = 2 \ldots n]; \\ \qquad \qquad \qquad \qquad \text{otherwise} \end{cases}$$

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

	l_1	l_2	l_3	l_4
$\hat{c}$	$\{s_a\}$	$\{\}$	$\{s_b, s_c\}$	$\{s_d\}$
$\hat{\mathcal{U}}^{\cap}(\hat{c}, s_c)$	$\{s_c\}$	$\{s_a\}$	$\{s_b\}$	$\{s_d\}$

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.

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

[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. For non-precisely determined addresses of data references, one can use a set of possibly referenced memory blocks.

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

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

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

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

	l_1	l_2	l_3	l_4
$\hat{c}_1$	$\{s_a\}$	$\{s_b\}$	$\{s_c\}$	$\{s_d\}$
$\hat{c}_2$	$\{s_c\}$	$\{s_e\}$	$\{s_a\}$	$\{s_d\}$
$\hat{\mathcal{J}}^{\cap}(\hat{c}_1, \hat{c}_2)$	$\{\}$	$\{\}$	$\{s_c, s_a\}$	$\{s_d\}$

An abstract cache state $\hat{c}$ at a control flow node k that references a memory block s_x is interpreted in the following way: 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).

5.2 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 abstract cache update function:

$$\hat{\mathcal{U}}^{\cup}(\hat{c}, s_x) = \hat{c}'$$

$$\hat{c}' = \begin{cases} [l_1 \mapsto \{s_x\}, \\ \;\; l_i \mapsto \hat{c}(l_{i-1}) \mid i = 2 \ldots h, \\ \;\; l_{h+1} \mapsto \hat{c}(l_{h+1}) \cup (\hat{c}(l_h) - \{s_x\}), \\ \;\; l_i \mapsto \hat{c}(l_i) \mid i = h+2 \ldots n]; \\ \hfill \text{if } \exists l_h : s_x \in \hat{c}(l_h) \\ [l_1 \mapsto \{s_x\}, l_i \mapsto \hat{c}(l_{i-1}) \text{ for } i = 2 \ldots n]; \\ \hfill \text{otherwise} \end{cases}$$

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

	l_1	l_2	l_3	l_4
$\hat{c}$	$\{s_a\}$	$\{s_b, s_c\}$	$\{\}$	$\{s_d\}$
$\hat{\mathcal{U}}^{\cup}(\hat{c}, s_c)$	$\{s_c\}$	$\{s_a\}$	$\{s_b\}$	$\{s_d\}$

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.

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

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

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

	l_1	l_2	l_3	l_4
$\hat{c}_1$	$\{s_a\}$	$\{s_b\}$	$\{s_c\}$	$\{s_d\}$
$\hat{c}_2$	$\{s_c\}$	$\{s_e, s_f\}$	$\{s_a\}$	$\{s_d\}$
$\hat{\mathcal{J}}^{\cup}(\hat{c}_1, \hat{c}_2)$	$\{s_a, s_c\}$	$\{s_b, s_e, s_f\}$	$\{\}$	$\{s_d\}$

An abstract cache state $\hat{c}$ at a control flow node k that references a memory block s_x is interpreted in the following way: 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).

5.3 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 $\hat{c} : L \longrightarrow 2^S$ is finite. Hence, every ascending chain is finite. Additionally, the abstract cache update functions $\hat{\mathcal{U}}$ and the join functions $\hat{\mathcal{J}}$ are monotonic. This guarantees that our analysis will terminate.

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 the interprocedural analysis (see Figure 1).

```
                               proc loop_L();
  :
                               if P then
while P do                       BODY
     BODY           ==>          loop_L();        (2)
end;                           end
  :                              :
                               loop_L();          (1)
                                 :
```

Figure 1: 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 a 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 [11] and are implemented in PAG.

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 has therefore an infinite number of paths. So only the information obtained on paths with a different suffix of a fixed length K is 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 best possible 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 very 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 1); 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. But they do not help for a WCET estimation, as the additionally 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.

- **Functional approach:** Here the same argument as for callstrings longer than one holds.

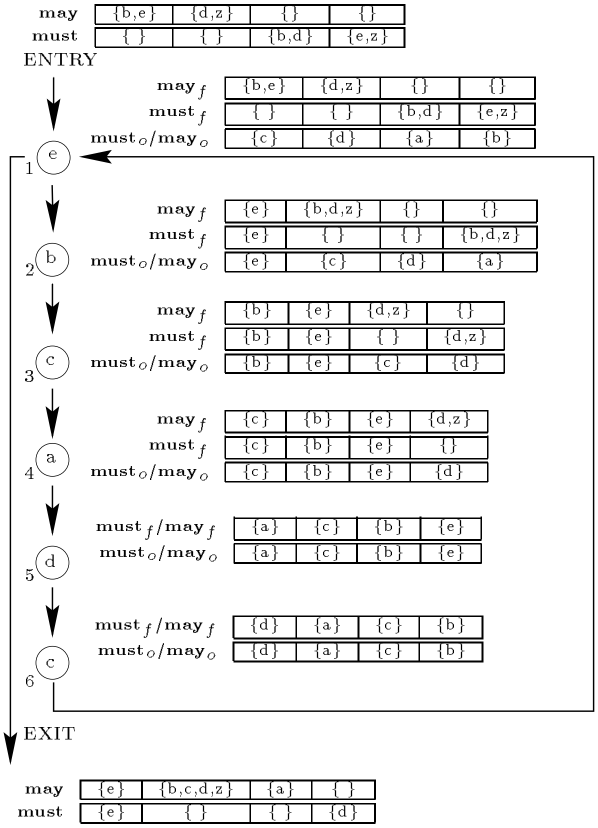
To get more precise results for the cache behavior prediction, we have developed the **VIVU approach** which has been implemented with the mapping mechanism of PAG as described in [2]. It corresponds to callstring(∞), but paths through the call graph that only differ in the number of repeated passes through a cycle are not distinguished. It can be compared with a combination of *virtual inlining* of all non recursive procedures[5] and *virtual unrolling* of the first iterations of all recursive procedures including loops. 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 9.

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


may {b,e} {d,z} {} {}
must { } { } {b,d} {e,z}
ENTRY
may_f {b,e} {d,z} {} {}
must_f { } { } {b,d} {e,z}
must_o/may_o {c} {d} {a} {b}
1 e
may_f {e} {b,d,z} {} {}
must_f {e} { } { } {b,d,z}
must_o/may_o {e} {c} {d} {a}
2 b
may_f {b} {e} {d,z} {}
must_f {b} {e} { } {d,z}
must_o/may_o {b} {e} {c} {d}
3 c
may_f {c} {b} {e} {d,z}
must_f {c} {b} {e} {}
must_o/may_o {c} {b} {e} {d}
4 a
must_f/may_f {a} {c} {b} {e}
must_o/may_o {a} {c} {b} {e}
5 d
must_f/may_f {d} {a} {c} {b}
must_o/may_o {d} {a} {c} {b}
6 c
EXIT
may {e} {b,c,d,z} {a} { }
must {e} { } { } {d}

(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 2: 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.

The control flow graph and the result of the analyses with VIVU[6] are shown in Figure 2. 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

[5] This has also been used in [9, 6].

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

analysis, we assume the loop has been implicitly transformed into a procedure according to Figure 1.

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

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 [13]. This allows to use our analysis to predict the behavior of data caches or combined instruction/data caches for programs that use only scalar variables. [1] also describes methods to handle writes to caches for common cache organizations (*write through* and *write back* with *write allocate* or *no write allocate*) as well as write buffers.

9 Practical Experiments

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

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 execution 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*.

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

Additionally, we compute for every instruction/context pair *ic* with $cat(ic) = \mathsf{nc}$ 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 in the given context will result in at most one cache miss. Generally, an upper bound of the cache misses of the instruction is given by one plus the maximal number of possible sequences of length A of executions of competing instructions that are 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.

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 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 [8]. 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 3).

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 3). 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.

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

For the experiments we use parts of the program suites of Frank Müller [3, 9], the djpeg program of Yau-Tsun Steven Li [7], 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

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

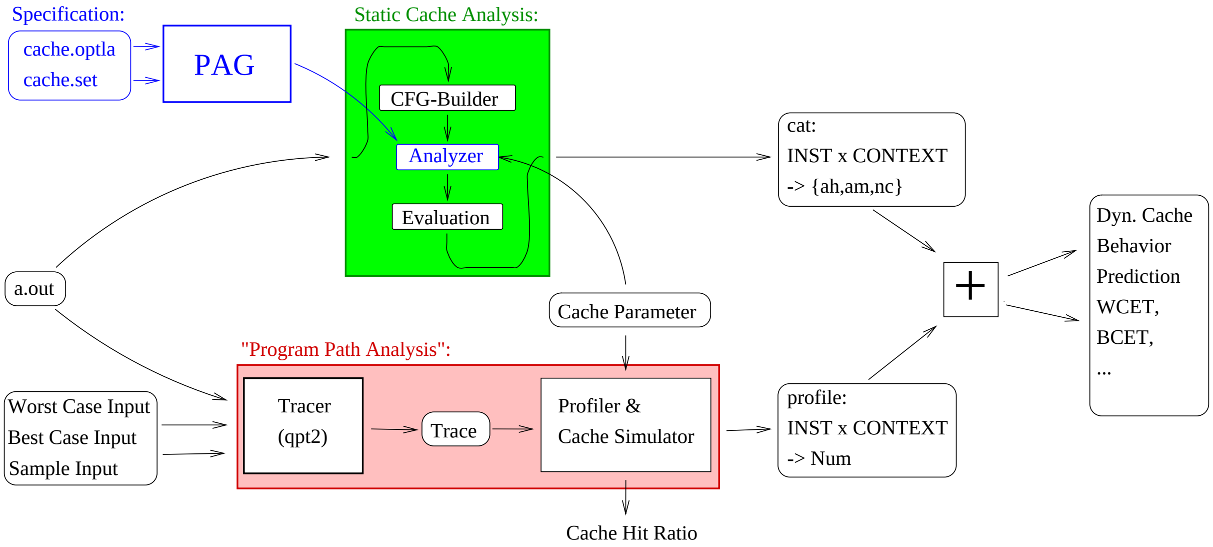


Figure 3: The structure of the analysis.

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.

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 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 cycles[8].

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

[8] This are the same parameters as used in [10].

always started with the empty cache, and we assume uninterrupted execution. For technical reasons, instructions in functions from dynamic link libraries[9] 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 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 Figure 4 in percent of the execution time ET.

Figure 4 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.

10 Comparison with Related Work

The work of Arnold, Müller, Whalley, and Harmon has been one of the starting points of our work. [9] 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 [10]. 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 [10][10]. 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 [6, 7] 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[11]. Unlike the method described in [9] 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 [7]. 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 [5], Lim et al. describe a general framework for the computation of WCETs of programs in the presence of pipelines and cache memories. Two kinds of pipeline and cache state information are associated with every program construct for which timing equations can be formulated. One describes the pipeline and cache state when the program construct is finished. The other can be combined with the state information from the previous construct to refine the WCET computation for that program construct. Unlike our method that is based on well explored theories and tools for abstract interpretation, the set of timing equations must be explicitly solved. An approximation to the solution for the set of timing equations has been proposed. The usage of an input and output state provides a way for a modularization for the timing analysis. Experimental results are reported for three small programs, but they cannot be easily compared with our experiments.

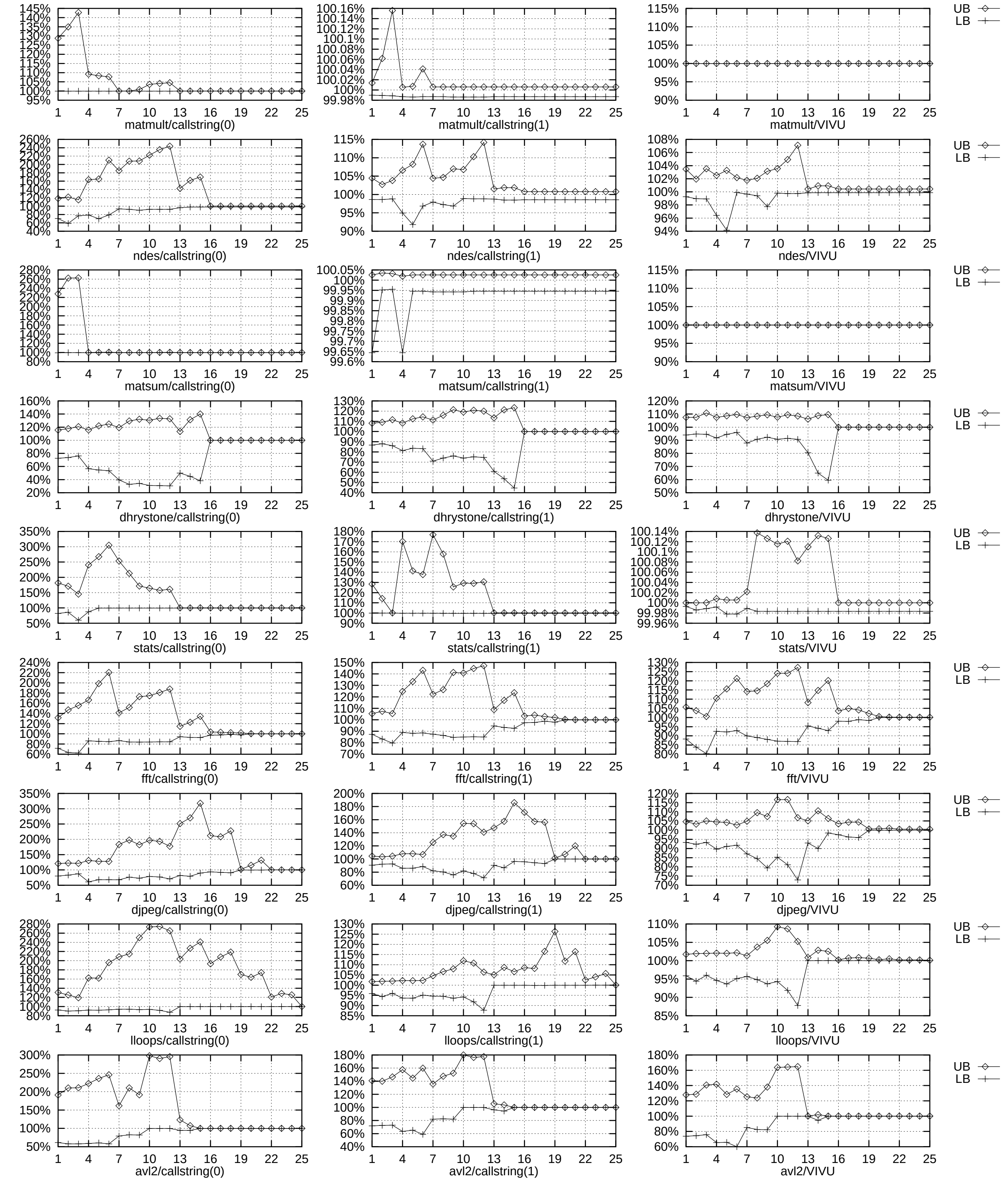
11 Conclusion and Future Work

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

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

[10] For the sake of space, the results of not all programs could be reported here.

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



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 4: 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.

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 frontends 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.

Acknowledgements

We like to thank Mark D. Hill, James R. Larus, Alvin R. Lebeck, Madhusudhan Talluri, and David A. Wood for making available the Wisconsin architectural research tool set (WARTS), Thomas Ramrath for the implementation of the PAG frontend for SPARC executables, Yau-Tsun Steven Li and Frank Müller for providing their benchmark programs, Martin Alt for an early implementation, the latter two for fruitful discussions, and the unknown referees for their helpful remarks.

References

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

[2] M. Alt and F. Martin. Generation of Efficient Interprocedural Analyzers with PAG. In *SAS'95, Static Analysis Symposium*, LNCS 983. Springer, 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*, 1994.

[4] 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*, 1977.

[5] 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), 1995.

[6] 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*, 1995.

[7] 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*, 1996.

[8] 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*, 1995.

[9] 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*, 1994.

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

[11] 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*. Prentice-Hall, 1981.

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

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