Symbolic Cache Analysis for Real-Time Systems

Johann Blieberger (blieb@auto.tuwien.ac.at)
Department of Computer-Aided Automation, Technical University of Vienna, Treitlstr. 1, A-1040 Vienna, Austria

Thomas Fahringer (tf@par.univie.ac.at) and
Bernhard Scholz (scholz@par.univie.ac.at)
Institute for Software Technology and Parallel Systems, University of Vienna, Liechtensteinstr. 22, A-1090 Vienna, Austria

Abstract. Caches impose a major problem for predicting execution times of real-time systems since the cache behavior depends on the history of previous memory references. Too pessimistic assumptions on cache hits can result in worst-case execution time estimates that are prohibitive for real-time systems.

This paper presents a novel approach for deriving a highly accurate analytical cache hit function for programs at compile-time based on the assumption that no external cache interference (e.g. process dispatching or DMA activity) occurs. First, a *symbolic tracefile* of an instrumented program is generated based on *symbolic evaluation* which is a static technique to determine the dynamic behavior of programs. Symbolic expressions and recurrences are used to describe all memory references in a program which are then stored chronologically in the symbolic tracefile. Second, a cache hit function for several cache architectures is computed based on a *novel cache evaluation technique*. Our approach goes beyond previous work by precisely modelling program control flow and program unknowns, modelling large classes of programming languages and cache architectures, and providing very accurate cache hit predictions.

Examples for the SPARC architecture are used to illustrate the accuracy and effectiveness of our symbolic cache prediction.

Keywords: Cache Hit Prediction, Symbolic Evaluation, Static Analysis, Worst-Case Execution Time

1. Introduction

Due to high-level integration and superscalar architectural designs the computational capability of microprocessors has increased significantly in the last few years. Unfortunately the gap between processor cycle time and memory latency increases. In order to fully exploit the potential of processors, the memory hierarchy must be efficiently utilized.

To guide scheduling for real-time systems, information about execution times is required at compile-time. Modelling caches presents a major obstacle towards predicting execution times for modern computer architectures. Worst-case assumptions – e.g. every memory access results in a cache miss[1] – can cause very poor execution time estimates.

© 1999 *Kluwer Academic Publishers. Printed in the Netherlands.*

The focus of this paper is on accurate cache behavior analysis. Note that modelling caches is only one performance aspect that must be considered in order to determine execution times. There are many other performance characteristics (Blieberger, 1994; Blieberger and Lieger, 1996; Blieberger, 1997; Fahringer, 1996; Park, 1993; Healy et al., 1995) to be analyzed which however are beyond the scope of this paper.

In this paper we introduce a novel approach for deriving a highly accurate analytical function of the *precise* number of cache hits[2] implied by a program. Our approach is based on *symbolic evaluation* (cf. e.g. Fahringer and Scholz, 1997) which at compile-time collects runtime properties (control and data flow information) of a given program. The number of cache hits is described by symbolic expressions and recurrences defined over the program's input data so as to maintain the relationship between the cache cost function and the input data.

Figure 1 depicts an overview of our framework described in this paper. The C-program is compiled which results in an instrumented C-program. The source-code level instrumentation inserts code at those points, where main memory data is referenced (read or written). Then, the instrumented source-code is symbolically evaluated and a *symbolic tracefile* is created. All memory references of a program are described by symbolic expressions and recurrences which are stored in a symbolic tracefile. Based on the cache parameters, which describe the cache architecture, an analytical cache hit function is computed by symbolically evaluating the symbolic tracefile. Note that our model strictly separates machine specific cache parameters from the program model which substantially alleviates portability of our approach to other cache architectures and programming languages.

Performing a worst-case cache analysis according to our approach can be divided into the following steps:

1. Build the symbolic tracefile based on the instrumented program sources by using symbolic evaluation.

2. Compute an analytical cache hit function by symbolically evaluating the symbolic tracefile.

3. Find a closed form expression for the cache hit function.

4. Determine a lower bound of the cache hit function in order to derive the worst-case caching behavior of the program.

Steps 1 and 2 are treated in this paper. These steps guarantee a precise description of the cache hits and misses.

Step 3 requires to solve recurrence relations. We have implemented a recurrence solver which is described in (Fahringer and Scholz, 1997;

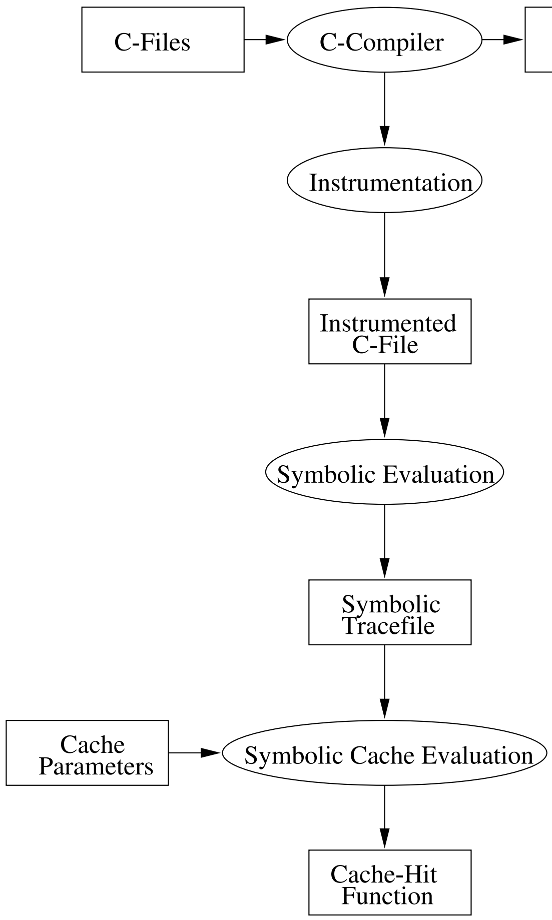


Figure 1. Overview of predicting cache performance

Fahringer and Scholz, 1999). The current implementation of our recurrence solver handles recurrences of the following kind: linear recurrence variables (incremented inside a loop by a symbolic expression defined over constants and invariants), polynomial recurrence variables (incremented by a linear symbolic expression defined over constants, invariants and recurrence variables) and geometric recurrence variables (incremented by a term which contains a recurrence variable multiplied by an invariant). Our algorithm (Fahringer, 1998b) for computing lower and upper bounds of symbolic expressions based on a set of constraints is used to detect whether a recurrence variable monotonically increases or decreases. Even if no closed form can be found for a recurrence variable, monotonicity information may be useful, for instance, to determine whether a pair of references can ever touch the same address. The current implementation of our symbolic evaluation framework models assignments, GOTO, IF, simple I/O and array statements, loops and procedures.

The result of Step 3 is a conservative approximation of the number of exact cache hits and misses, i.e., the computed upper and lower bounds are used to find a lower bound for the cache hit function. The output form of Step 3 (suitably normalized) is a case-structure that possibly comprises several cache hit functions. The conditions attached to the different cases correspond to the original program structure and are affected by the cache architecture.

In Step 4 we only have to determine the minimum of the cache hit functions of the case-structure mentioned above. Note that it is not necessary to determine the worst-case input data because the program structure implies the worst-case cache behavior.

Steps 3 and 4 are described in detail in (Fahringer and Scholz, 1997; Fahringer and Scholz, 1999; Fahringer, 1998b).

The rest of the paper is organized as follows. In Section 2 we discuss our architecture model for caches. In Section 3 we describe symbolic evaluation and outline a new model for analyzing arrays. Section 4 contains the theoretical foundations of symbolic tracefiles and illustrates a practical example. In Section 5 symbolic cache evaluation techniques are presented for direct mapped and set associative caches. In Section 6 we provide experimental results. Although our approach will be explained and experimentally examined based on the C-programming language, it can be similarly applied to most other procedural languages including Ada and Fortran. In Section 7 we compare our approach with existing work. Finally, we conclude this paper in Section 8.

2. Caches

The rate at which the processor can execute instructions is limited by the memory cycle time. This limitation has in fact been a significant problem because of the persistent mismatch between processor and main memory speeds. Caches – which are relatively small high-speed memories – have been introduced in order to hold the contents of most recently used data of main memory and to exploit the phenomenon of locality of reference (see (Hennessy and Patterson, 1990)). The advantage of a cache is to improve the average access time for data located in main memory. The concept is illustrated in Figure 2.

The cache contains a small portion of main memory. A cache hit occurs, when the CPU requests a memory reference that is found in the cache. In this case the reference (memory word) is transmitted

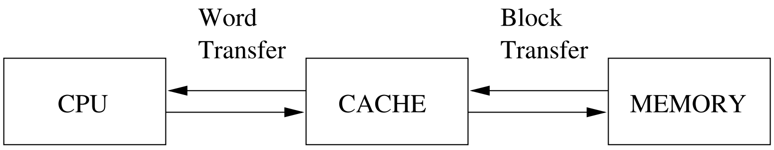


Figure 2. CPU, Cache and Main Memory

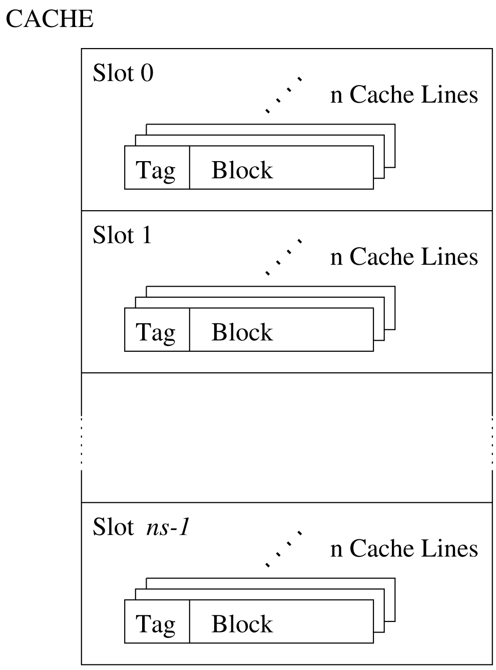
to the CPU. Otherwise, a cache miss occurs which causes a block of memory (a fixed number of words) to be transferred from the main memory to the cache. Consequently, the reference is transmitted from the cache to the CPU. Commonly the CPU is stalled on a cache miss. Clearly, memory references that cause a cache miss are significantly more costly than if the reference is already in the cache.

In the past, various cache organizations (Hennessy and Patterson, 1990) were introduced. Figure 3(a) depicts a general cache organization. A cache consists of *ns* slots. Each slot can hold n *cache lines* and one cache line contains a block of memory consisting of *cls* contiguous bytes and a tag that holds the first address bits of the memory block. Figure 3(b) shows how an address is divided into three fields to find data in the cache: the *block offset* field used to select the desired data from the block, the *index* field to select the slot and the tag field used for comparison. Note that not all bits of the index are used if $n > 1$.

A cache can be characterized by three major parameters. First, the *capacity* of a cache determines the number of bytes of main memory it may contain. Second, the line size *cls* gives the number of contiguous bytes that are transferred from memory on a cache miss. Third, the associativity determines the number of cache lines in a slot. If a block of memory can reside in exactly one location, the cache is called *direct mapped* and a cache set can only contain one cache line. If a block can reside in any cache location, the cache is called *fully associative* and there is only one slot. If a block can reside in exactly n locations and n is the size of a cache set, the cache is called *n-way set associative*.

In case of fully associative or set associative caches, a memory block has to be selected for replacement when the cache set of the memory block is full and the processor requests further data. This is done according to a *replacement strategy* (Smith, 1982). Common strategies are *LRU* (Least Recently Used), *LFU* (Least Frequently Used), and *random*.

Furthermore, there are two common cache policies with respect to write accesses of the CPU. First, the *write through* caches write data to memory and cache. Therefore, both memory and cache are in line. Second, *write back* caches only update the cache line where the data



(a) Cache Organization

Tag	Index	Block Offset

(b) The 3 Portions of an Address

Figure 3. Cache Organization

item is stored. For write back caches the cache line is marked with a *dirty bit*. When a different memory block replaces the modified cache line, the cache updates the memory.

A write access of the CPU to an address that does not reside in the cache is called a *write miss*. There are two common cache organizations with respect to write misses. First, the *write-allocate* policy loads the referenced memory block into the cache. This policy is generally used for write back caches. Second, the no-write-allocate policy updates the cache line only if the address is in cache. This policy is often used for write through cache and has the advantages that memory always contains up-to-date information and the elapsed time needed for a write access is constant.

Caches can be further classified. A cache that holds only instructions is called *instruction cache.* A cache that holds only data is called *data cache.* A cache that can hold instructions and data is called a *mixed* or *unified cache.*

Cache design has been extensively studied. Good surveys can be found in (Alt et al., 1996; Mueller, 1997; Ottosson and Sjoedin, 1997; Li et al., 1996; Li et al., 1995; Healy et al., 1995; Arnold et al., 1994; Nilsen and Rygg, 1995; Liu and Lee, 1994; Hennessy and Patterson, 1990).

3. Symbolic Evaluation

Symbolic evaluation[3] (Cheatham et al., 1979; Ploedereder, 1980; Fahringer and Scholz, 1997; Fahringer and Scholz, 1999) $\mathcal{F}$ is a constructive description of the semantics of a program. Moreover, symbolic evaluation is not merely an arbitrary alternative semantic description of a program. As in the relationship between arithmetic and algebra the specific (arithmetic) computations dictated by the program operators are generalized and "delayed" using the appropriate formulas. The dynamic behavior is *precisely* represented.

Symbolic evaluation satisfies a commutativity property.

$$\begin{tikzcd}
(\mathcal{S}_{\text{conc}}[\![\mathfrak{p}]\!], \mathfrak{i}) \arrow[r, "\text{Symbolic Evaluation}"] \arrow[d, "\text{Set parameters to } \mathfrak{i}"'] & (\mathcal{F}[\![\mathfrak{p}]\!], \mathfrak{i}) \arrow[d, "\text{Substitute } \mathfrak{i} \text{ into result}"] \\
\mathcal{S}_{\text{conc}}[\![\mathfrak{p}]\!]\mathfrak{i} \arrow[r, "\text{Conventional Execution}"] & \mathcal{F}[\![\mathfrak{p}]\!]\mathfrak{i}
\end{tikzcd}$$

If a program $\mathfrak{p}$ is conventionally executed with the standard semantics $\mathcal{S}_{\text{conc}}[\![\mathfrak{p}]\!]$ over a given input $\mathfrak{i}$, the result of the symbolically evaluated program $\mathcal{F}[\![\mathfrak{p}]\!]$ instantiated by $\mathfrak{i}$ is the same. Clearly, symbolic evaluation can be seen as a compiler, that translates a program into a different language. Here, we use as a target language *symbolic expressions* and *recurrences* to model the semantics of a program.

The semantic domain of our symbolic evaluation is a novel representation called *program context* (Fahringer and Scholz, 1997; Fahringer and Scholz, 1999). Every statement is associated with a program context c that describes the variable values, assumptions regarding and constraints between variable values and a path condition. The path condition holds for a given input if the statement is executed. Formally, a context c is defined by a triple $[s, t, p]$ where s is a state, t a state condition and p a path condition.

– The state s is described by a set of variable/value pairs $\{v_1 = e_1, \ldots, v_n = e_n\}$ where v_i is a program variable and e_i a symbolic expression describing the value of v_i for $1 \leq i \leq n$. For all program variables v_i there exists exactly one pair $v_i = e_i$ in state s.

- The state condition contains constraints on variable values such as those implied by loops, variable declarations and user assertions.
- Path condition is a predicate, which is true if and only if the program statement is reached.

Note that all components of a context — including state information — are described as symbolic expressions and recurrences. An unconditional sequence of statements ℓ_j $(1 \leq j \leq r)$ is symbolically evaluated by $[s_0, t_0, p_0]$ ℓ_1 $[s_1, t_1, p_1]$ $\dots$ ℓ_r $[s_r, t_r, p_r]$. The initial context $[s_0, t_0, p_0]$ represents the context that holds before ℓ_1 and $[s_r, t_r, p_r]$ the context that holds after ℓ_r. If ℓ_i in the sequence $\dots$ $[s_i, t_i, p_i]$ ℓ_i $[s_{i+1}, t_{i+1}, p_{i+1}]$ $\dots$ does not contain any side effects (implying a change of a variable value) then $s_i = s_{i+1}$.

Furthermore, a context $c = [s, t, p]$ is a logical assertion $\overline{c} = s \wedge t \wedge p$, where $\overline{c}$ is a predicate over the set of program variables and the program input which are free variables. If for all input values $\overline{c_{i-1}}$ holds before executing the statement ℓ_i then $\overline{c_i}$ is the strongest post condition (Dijkstra, 1976) and the program variables are in a state satisfying $\overline{c_i}$ after executing ℓ_i.

For further technical details we refer the reader to (Fahringer and Scholz, 1997; Fahringer and Scholz, 1999; Blieberger and Burgstaller, 1998; Blieberger et al., 1999). In the following we discuss a novel approach to evaluate arrays.

3.1. Arrays

Let `a` be a one-dimensional array with n $(n \geq 1)$ array elements. Consider the simple array assignment `a[i]=v`. The element with index `i` is substituted by the value of `v`. Intuitively, we may think of an array assignment being an array operation that is defined for an array. The operation is applied to the array and changes its internal state. The arguments of such an array operation are a value and an index of the new assigned array element. A sequence of array assignments implies a chain of operations. Formally, an array is represented as an element of an *array algebra* $\mathbb{A}$. The array algebra $\mathbb{A}$ is inductively defined as follows.

1. If n is a symbolic expression then $\perp_n \in \mathbb{A}$.
2. If $a \in \mathbb{A}$ and α, β are symbolic expressions then $a \oplus (\alpha, \beta) \in \mathbb{A}$.
3. Nothing else is in $\mathbb{A}$.

```
int a[100],x;
...
```
$c_0 = [s_0 = \{x = \underline{x}, a = \bot_{100}\}, t_0 = true, p_0 = true]$

ℓ_1 : `a[x]=1;`

$c_1 = [s_1 = \delta\,(s_0\,; a = \bot_{100} \oplus (1, \underline{x}))\,, t_1 = t_0, p_1 = p_0]$

ℓ_2 : `a[x+1]=1-x;`

$c_2 = [s_2 = \delta\,(s_1\,; a = \bot_{100} \oplus (1, \underline{x}) \oplus (1 - \underline{x}, \underline{x} + 1))\,, t_2 = t_1, p_2 = p_1]$

ℓ_3 : `a[x]=x;`

$c_3 = [s_3 = \delta\,(s_2\,; a = \bot_{100} \oplus (1, \underline{x}) \oplus (1 - \underline{x}, \underline{x} + 1) \oplus (\underline{x}, \underline{x}))\,, t_3 = t_2, p_3 = p_2]$

ℓ_4 : `a[x+1]=1+x;`

$c_4 = [s_4 = \delta\,(s_3\,; a = \bot_{100} \oplus (1, \underline{x}) \oplus (1 - \underline{x}, \underline{x} + 1) \oplus (\underline{x}, \underline{x}) \oplus (1 + \underline{x}, \underline{x} + 1))\,,$
$t_4 = t_3, p_4 = p_3]$

...

Figure 4. C-program fragment

In the state of a context, an array variable is associated with an element of the array algebra $\mathbb{A}$. Undefined array states are denoted by $\bot_n$, where n is the *size* of the array and determines the number of array elements. An array assignment is modelled by a $\oplus$*-function.* The semantics of the $\oplus$-function is given by

$$a \oplus (\alpha, \beta) = (v_1, \dots, v_{\beta-1}, \alpha, v_{\beta+1}, \dots, v_n)$$

where $(v_1, \dots, v_n)$ represents the elements of array a and β denotes the index of the element with a new value α. For the following general array assignment

$$\begin{array}{ll} & \dots \\ & [s_{i-1} = \{\dots, a = \underline{a}, \dots\}, t_{i-1}, p_{i-1}] \\ \ell_i : & \mathtt{a[}\beta\mathtt{]} \ \mathtt{=} \ \alpha; \\ & [s_i = \{\dots, a = \underline{a} \oplus (\alpha, \beta), \dots\}\,, t_i = t_{i-1}, p_i = p_{i-1}] \\ & \dots \end{array}$$

$[s_{i-1}, t_{i-1}, p_{i-1}]$ is the context before and $[s_i, t_i, p_i]$ the context after statement ℓ_i. The symbolic value of variable `a` before evaluating the statement ℓ_i is denoted by $\underline{a}$. Furthermore, an element a in $\mathbb{A}$ with at least one $\oplus$-function is a $\oplus$*-chain.* Every $\oplus$-chain can be written as $\bot_n \bigoplus_{k=1}^{m} (\alpha_k, \beta_k)$. The *length of a chain* $|a|$ is the number of $\oplus$-functions in chain a.

The C-program fragment in Figure 4 illustrates the evaluation of several array assignments. The context of statement ℓ_j is represented by $c_j = [\dots]$. At the beginning of the program fragment the value of variable `x` is a symbolic expression denoted by $\underline{x}$. Array `a` is undefined

($\perp_{100}$). For all array assignment statements the state and path conditions are set to *true* because the code fragment implies no branches.

Most program statements imply a change of only a single variable's value. In order to avoid large lists of variable values in state descriptions only those variables whose value changes after evaluation of the associated statement are explicitly specified. For this reason we introduce a function δ,

$$s_i = \delta(s_j; v_1 = e_1, \ldots, v_l = e_l)$$

which specifies a state s_i whose variable binding is equal to that of state s_j except for variable v_i $(1 \leq i \leq l)$. Variable v_i is assigned a new value e_i.

Therefore, in the previous example, state s_1 is the same as state s_0 except for the symbolic value of array `a`.

After the last statement array `a` is symbolically described by $a = \perp_{100} \oplus (1, \underline{x}) \oplus (1 - \underline{x}, \underline{x} + 1) \oplus (\underline{x}, \underline{x}) \oplus (1 + \underline{x}, \underline{x} + 1)$. The left-most $\oplus$-function relates to the first assignment of the example program — the right-most one to the last statement.

Note that the last two statements overwrite the values of the first two statements. Therefore, a simplified representation of `a` is given by $\perp_{100} \oplus (\underline{x}, \underline{x}) \oplus (1 + \underline{x}, \underline{x} + 1)$.

Although the equivalence of two symbolic expressions is undecidable (Fahringer, 1998a; Haghighat and Polychronopoulos, 1996), a wide class of equivalence relations can be solved in practice. The set of conditions among the used variables in the context significantly improves the evaluation of equivalence relation. A partial *simplification operator* θ is introduced to simplify $\oplus$-chains. Operator θ is defined as follows.

$$\theta\left(\perp_n \bigoplus_{l=1}^{m} (\alpha_l, \beta_l)\right) = \begin{cases} \perp_n \bigoplus_{l=1, l \neq i}^{m} (\alpha_l, \beta_l), & \text{if } \exists 1 \leq i < j \leq m : \beta_i = \beta_j \\ \perp_n \bigoplus_{l=1}^{m} (\alpha_l, \beta_l), & \text{otherwise} \end{cases}$$

The partial simplification operator θ seeks for two equal β expressions in a $\oplus$-chain. If a pair exists, the result of θ will be the initial $\oplus$-chain without the $\oplus$-function, which refers to the β expression with the smaller index i. If no pair exists, the operator returns the initial $\oplus$-chain; the chain could not be simplified. Semantically, the right-most β expression relates to the latest assignment and overwrites the value of the previous assignment with the same *symbolic index*.

The partial simplification operator θ reduces only one redundant $\oplus$-function. In the previous example θ must be applied twice in order to simplify the $\oplus$-chain. Moreover, each $\oplus$-function in the chain is a potentially redundant one. Therefore, the chain is potentially simplified in less than $|a|$ applications of θ. A partially complete simplification is an iterative application of the partial simplification operator and it is written as $\theta^*(a)$. If $\theta^*(a)$ is applied to a, further applying of θ will not simplify a anymore: $\theta(\theta^*(a)) = \theta^*(a)$.

In order to access elements of an array we need to model a symbolic access function. *Operator* ρ in a symbolic expressions e (described by a $\oplus$-chain) reads an element with index i of an array `a`. If index i can be found in the $\oplus$-chain, ρ yields the corresponding symbolic expression otherwise ρ is the undefined value $\perp$. In the latter case it is not possible to determine whether the array element with index i was written. Let a be an element of $\mathbb{A}$ and $a = \perp_n \bigoplus_{l=1}^{m}(\alpha_l, \beta_l)$. The operator ρ is defined as

$$\rho\left(\perp_n \bigoplus_{l=1}^{m}(\alpha_l, \beta_l), i\right) = \begin{cases} \alpha_l, & \text{if } \exists l = \max\{l | 1 \leq l \leq m \wedge \beta_l = i\} \\ \perp, & \text{otherwise} \end{cases}$$

where i is the *symbolic index* of the array element to be found. In general determining whether the symbolic index i matches with a $\oplus$-function is undecidable. In practice a wide class of symbolic relations can be solved by our techniques for comparing symbolic expressions (Fahringer, 1998a). If our symbolic evaluation framework cannot prove that the result of ρ is β_l or $\perp$ then ρ is not resolvable and remains unchanged in symbolic expression e.

We present four examples in Figure 5, which are based on the value of `a` at the end of the program fragment in Figure 4. For every example we insert one of the following statements at the end of the code fragment shown in Figure 4. For (1) `x=a[x];` (2) `x=a[x+1];` (3) `x=a[x-1];` and (4) `x=a[y];` where `y` is a new variable with the symbolic value of $\underline{y}$. The figure shows the symbolic value of x after the inserted statement.

Note that in the first equation the element with index $\underline{x}$ is uniquely determined. The second equation is resolved as well. In the third example the index $\underline{x} - 1$ does not exist in the $\oplus$-chain.

Therefore, the access returns the undefined symbol $\perp$. In the last equation we do not have enough information to determine a unique value for array element with index i. Here, we distinguish between several cases to cover all possibilities.

1. $x = \rho(a, \underline{x}) = \underline{x}$
2. $x = \rho(a, \underline{x} + 1) = \underline{x} + 1$
3. $x = \rho(a, \underline{x} - 1) = \perp$
4. $x = \rho(a, \underline{y}) = \begin{cases} \underline{x}, & \text{if } \underline{y} = \underline{x} \\ \underline{x} + 1, & \text{if } \underline{y} = \underline{x} + 1 \\ \perp, & \text{otherwise} \end{cases}$

Figure 5. Examples of ρ

3.2. Array operations inside of loops

Modelling loops implies a problem with *recurrence variables*[4]. We will use functions to model recurrences as follows: $i(k+1) = i(k) + 1$ where $i(k+1)$ is the value of a scalar variable `i` at the end of iteration $k+1$.

Our symbolic evaluation framework detects recurrence variables, determines the recurrence system and finally tries to find closed forms for recurrence variables at the loop exit by solving the recurrence system. The *recurrence system* is given by the *boundary conditions* (initial values for recurrence variables in the loop preheader), the *recurrence relations* (implied by the assignments to the recurrence variables in the loop body) and the *recurrence condition* (loop or exit condition).

We have implemented a recurrence solver (Scheibl et al., 1996) written on top of Mathematica. The recurrence solver tries to determine closed forms for recurrence variables based on their recurrence system which is directly obtained from the program context. The implementation of our recurrence solver is largely based on methods described in (Gerlek et al., 1995; Lueker, 1980) and improved by our own techniques (Fahringer and Scholz, 1997; Fahringer and Scholz, 1999).

Similar to scalar variables the array manipulation inside of loops are described by recurrences. A recurrence system over $\mathbb{A}$ consists of a boundary condition and a recurrence relation

$$a(0) = b, b \in \mathbb{A}$$
$$a(k+1) = a(k) \bigoplus_{l=1}^{m} (\alpha_l(k), \beta_l(k))$$

where $\alpha_l(k)$ and $\beta_l(k)$ are symbolic expressions and k is the recurrence index with $k \geq 0$. Clearly, every instance of the recurrence is an element of $\mathbb{A}$. Without changing the semantics of an array recurrence, θ^* can be applied to simplify the recurrence relation.

Operator ρ needs to be extended for array recurrences, such that arrays written inside of loops can be accessed, e.g. $\rho(a(z), i)$. The symbolic expression z is the number of loop iterations determined by the loop exit condition and i is the index of the accessed element. Furthermore, the recurrence index k is bounded to $0 \leq k \leq z$. To determine a possible $\oplus$-function, where the accessed element is written, a *potential index set* $X_l(i)$ of the l-th $\oplus$-function is computed.

$$\forall 1 \leq l \leq m : X_l(i) = \{ k | \beta_l(k) = i \wedge 0 \leq k \leq z \}$$

$X_l(i)$ contains all possible $\beta_l(k)$, $0 \leq k \leq z$ equal to the index i. If an index set has more than one element, the array element i is written in different loop iterations by the l-th $\oplus$-function. Only the last iteration that writes array element i is of interest. Consequently, we choose the element with the greatest index. The *supremum* $x_l(i)$ of an index set $X_l(i)$ is the greatest index such that

$$\forall 1 \leq l \leq m : x_l(i) = \max X_l(i)$$

Finally, we define operator ρ as follows.

$$\rho\left(a(z), i\right) = \begin{cases} \alpha_l(x_l(i)), & \text{if } \exists 1 \leq l \leq m : x_l(i) = \max_{1 \leq l \leq m} x_l(i) \\ \rho(a(0), i), & \text{otherwise} \end{cases}$$

The maximum of the supremum indices $x_l(i)$ determines the symbolic value $\alpha_l(x_l(i))$. If no supremum index exists, ρ returns the access to the value before the loop.

The example code of the program in Figure 6 shows how to symbolically evaluate an array access. The recurrence of $i(k)$ is resolved in state s_3 of ℓ_3. Due to the missing information about $\underline{a}$ the recurrence of array `a` is not resolvable but our symbolic evaluation still models the dynamic behavior of the example code.

4. Symbolic Tracefile

Tracing is the method of generating a sequence of instruction and data references encountered during program execution. The trace data is commonly stored in a tracefile and analyzed at a later point in time. For tracing, instrumentation is needed to insert code at those points in a program, where memory addresses are referenced. The tracefile is created as a side-effect of execution. Tracing requires a careful analysis of the program to ensure that the instrumentation correctly reflects the data or code references of a program. Moreover, the instrumentation

```
int n,i;
char a[100],s=0;
...
```

$c_0 = [s_0 = \{n = \underline{n}, i = \bot, a = \underline{a}, s = 0\}, t_0 = true, p_0 = true]$

ℓ_1 : `for(i=0;i<n-1;i++){`

$c_1 = [s_1 = \delta\,(s_0; i = i(k), a = a(k), s = s(k))\,, t_1 = (i(0) = 0 \wedge a(0) = \underline{a} \wedge s(0) = 0)\,,$
$p_1 = i(k) < \underline{n} - 1]$

ℓ_2 : `s=s+a[i];`

$c_2 = [s_2 = \delta\,(s_1; s = s(k) + \rho(a(k), i(k)))\,, t_2 = t_1\,, p_2 = p_1]$

ℓ_3 : `a[i]=a[i]+a[i+1];`

$c_3 = [s_3 = \delta\,(s_2; a = a(k) \oplus (\rho(a(k), i(k)) + \rho(a(k), i(k)+1), i(k)))\,, t_3 = t_2, p_3 = p_2]$

ℓ_4 : `}`

$c_4 = [s_4 = \delta(s_0; i = \max(0, \underline{n} - 1), a = a(max(0, \underline{n} - 1))),$
$t_4 = (a(0) = \underline{a} \wedge a(k+1) = a(k) \oplus (\rho(a(k), k) + \rho(a(k), k+1), k) \wedge$
$s(0) = 0 \wedge s(k+1) = s(k) + \rho(a(k), k))\,,$
$p_4 = true]$

```
...
```

Figure 6. C-program fragment

can be done at the source-code level or machine code level. For our framework we need a source-code level instrumentation. In the past a variety of different cache profilers were introduced, e. g. MTOOL (Goldberg and Hennessy, 1991), PFC-Sim (Callahan et al., 1990), CPROF (Lebeck and Wood, 1994).

The novelty of our approach is to compute the trace data symbolically at compile-time without executing the program. A *symbolic tracefile* is a constructive description for all possible memory references in chronological order. It is represented as symbolic expressions and recurrences.

In the following we discuss the instrumentation of the program in Figure 6. The SPARC assembler code is listed in Figure 7. The first part of the code is a loop preparation phase. In this portion of code the contents of variable `n` is loaded into a work register. Additionally, the address of `a` is built up in register `%g2`. Inside the loop, the storage location of `n` is not referenced anymore and there are four read accesses `s`, `a[i]`, `a[i]`, `a[i+1]` and two write accesses `s`, `a[i]`. Furthermore, the variable `i` is held in a register. Based on this information we can instrument the example program. In Figure 8 the instrumented program is shown where function `r_ref(`r`,`nb`)` denotes a read reference of address r with the length of nb bytes. For a write reference the function `w_ref()` is used.

```
     sethi %hi(n),%o3
     ld [%o3+%lo(n)],%g5;        read &n
     mov 0,%o1
     add %g5,-1,%g5
     cmp %o1,%g5
     bge .LL3
     sethi %hi(s),%o2
     sethi %hi(a),%g2
     or %g2,%lo(a),%g2
     add %g2,1,%o4
     mov %g2,%o0
.LL5:
     ldub [%o2+%lo(s)],%g2;      read &s
     ldub [%o0],%g3;             read &a[i]
     add %g2,%g3,%g2
     stb %g2,[%o2+%lo(s)];       write &s
     ldub [%o0],%g2;             read &a[i]
     ldub [%o1+%o4],%g3;         read &a[i+1]
     add %g2,%g3,%g2
     stb %g2,[%o0];              write &[i]
     cmp %o1,%g5
     bl .LL5
     add %o0,1,%o0
.LL3:
     retl
```

Figure 7. SPARC code of example in Figure 6

A *symbolic tracefile* is created by using a chain algebra. The references are stored as a chain. A symbolic trace file $t \in \mathbb{T}$ is inductively defined as follows.

1. $\perp \in \mathbb{T}$.
2. If $t \in \mathbb{T}$ and r and nb are symbolic expressions then $t \oplus \sigma(r, nb) \in \mathbb{T}$.
3. If $t \in \mathbb{T}$ and r and nb are symbolic expressions then $t \oplus \lambda(r, nb) \in \mathbb{T}$.
4. Nothing else is in $\mathbb{T}$.

Semantically, function σ is a write reference to the memory with symbolic address r whereby the number of referenced bytes is denoted by nb. We have similar semantics for read references λ, where r is the address and nb is the number of referenced bytes.

For instance, a 32-bit bus between the cache and CPU can only transfer a *word references* with 4 bytes. Therefore, a `double` data item (comprises 8 bytes) $\lambda(r,8)$ must be loaded in two consecutive steps by $\lambda(r,4) \oplus \lambda(r+4,4)$. For a word reference we do not need the number of referenced bytes anymore because it is constant. In the example above it is legal to rewrite $\lambda(r,8)$ as $\lambda(r) \oplus \lambda(r+4)$. This notation is extensively used in the examples of Section 5.

For loops we need recurrences

$$\mathfrak{t}(0) = \mathfrak{t}, \quad \mathfrak{t} \in \mathbb{T}$$
$$\mathfrak{t}(k+1) = \mathfrak{t}(k) \bigoplus_{l=1}^{m} \mu_l(k)$$

where $\mu_l(k)$ is a read or write reference ($\lambda(r_l(k), nb)$ or $\lambda(r_l(k), nb)$).

Symbolic evaluation is used to automatically generate the symbolic tracefile of a C-program. Instead of symbolically evaluating instrumentation calls we associate `w_ref` and `r_ref` with specific semantics. A pseudo variable $\mathfrak{t} \in \mathbb{T}$ is added to the program. A read reference `r_ref(`r`,`nb`)` is translated to $t \oplus \lambda(r, nb)$, where t is the state of the pseudo variable $\mathfrak{t}$ before evaluating the instrumentation. The same is done for write references except that λ is replaced by σ.

Let us consider the example in Figure 8. Before entering the loop, $\mathfrak{t}$ needs to log reference `r_ref(&n,4)`. Therefore, $\mathfrak{t}$ is equal to $\perp \oplus \lambda(\&n, 4)$ where $\&n$ denotes the address of variable `n`. Inside the loop a recurrence is used to describe $\mathfrak{t}$ symbolically. The boundary condition $\mathfrak{t}(0)$ is equal to $\perp \oplus \lambda(\&n, 4)$ and reflects the state before the loop. The recurrence relation is given by

$$\begin{aligned}\mathfrak{t}(k+1) = t(k) &\oplus \lambda(\&s,1) \oplus \lambda(\&a[k],1) \oplus \sigma(\&s,1) \\ &\oplus \lambda(\&a[k],1) \oplus \lambda(\&a[k+1],1) \oplus \sigma(\&a[k],1).\end{aligned}$$

Note that an alternative notation of $\&a[k]$ is $a + k$ where a is the start address of array `a`. Finally, the last value of k in the recurrence $\mathfrak{t}(k)$ is $\max(0, n-1)$ which is determined by the loop condition.

For the symbolic tracefile only small portions of the final program context are needed. Therefore, we extract the necessary parts from the final context to describe the symbolic tracefile. Here, the state condition and symbolic value $\mathfrak{t}$ are of relevance. For example in Figure 8 the symbolic tracefile is given by

$$\begin{aligned}\mathfrak{t} &= \mathfrak{t}\left(\max(0, n-1)\right), \\ \mathfrak{t}(0) &= \perp \oplus \lambda(\&n,4) \wedge \\ \mathfrak{t}(k+1) &= \mathfrak{t}(k) \oplus \lambda(\&s,1) \oplus \lambda(\&a[k],1) \oplus \sigma(\&s,1) \oplus \lambda(\&a[k],1) \\ &\quad \oplus \lambda(\&a[k+1],1) \oplus \sigma(\&a[k],1)\end{aligned} \tag{1}$$

```
int n,i;
char a[100],s=0;
...
```

$$c_0 = [s_0 = \{n = \underline{n}, i = \perp, a = \underline{a}, s = 0, \mathfrak{t} = \perp\}, t_0 = true, p_0 = true]$$

$\ell_1 :$ `r_ref(&n,4);`

$$c_1 = [s_1 = \delta\,(s_0, \mathfrak{t} = \perp \oplus \lambda(\&n, 4))\,, t_1 = t_0, p_1 = p_0]$$

$\ell_2 :$ `for(i=0;i<n-1;i++){`

$$c_2 = [s_2 = \delta(s_1; i = i(k), a = a(k), \mathfrak{t} = \mathfrak{t}(k)),$$
$$t_2 = (i(0) = 0 \wedge a(0) = \underline{a} \wedge \mathfrak{t}(0) = \perp \oplus \lambda(\&n, 4))\,, p_2 = i(k) < \underline{n} - 1]$$

$\ell_3 :$ `r_ref(&s,1); r_ref(&a[i],1); w_ref(&s,1);`

$$c_3 = [s_3 = \delta\,(s_2, \mathfrak{t} = \mathfrak{t}(k) \oplus \lambda(\&s, 1) \oplus \lambda(\&a[i], 1) \oplus \sigma(\&s, 1))\,, t_3 = t_2, p_3 = p_2]$$

$\ell_4 :$ `s=s+a[i];`

$$c_4 = [s_4 = \delta\,(s_3; s = s(k) + \rho(a(k), i(k)))\,, t_4 = t_3, p_4 = p_3$$

$\ell_5 :$ `r_ref(&a[i],1); r_ref(&a[i+1],1); w_ref(&a[i],1);`

$$c_5 = [s_5 = \delta\,(s_4, \mathfrak{t} = \mathfrak{t}(k) \oplus \lambda(\&s, 1) \oplus \lambda(\&a[i], 1) \oplus \sigma(\&s, 1) \oplus$$
$$\lambda(\&a[i], 1) \oplus \lambda(\&a[i+1], 1) \oplus \sigma(\&a[i], 1))\,, t_5 = t_4, p_5 = p_4]$$

$\ell_6 :$ `a[i]=a[i]+a[i+1];`

$$c_6 = [s_6 = \delta\,(s_5; a = a(k) \oplus (\rho(a(k), i(k)) + \rho(a(k), i(k)+1), i(k)))\,, t_6 = t_5, p_6 = p_5$$

$\ell_7 :$ `}`

$$c_7 = [s_7 = \delta(s_0; i = \max(0, \underline{n} - 1)), a = a(\max(0, \underline{n} - 1)),$$
$$s = s(\max(0, \underline{n} - 1)), \mathfrak{t} = \mathfrak{t}(\max(0, \underline{n} - 1))),$$
$$t_7 = (a(0) = \underline{a} \wedge a(k+1) = a(k) \oplus (\rho(a(k), k) + \rho(a(k), k+1), k) \wedge$$
$$s(0) = 0 \wedge s(k+1) = s(k) + \rho(a(k), k) \wedge$$
$$\mathfrak{t}(0) = \perp \oplus \lambda(\&n, 4) \wedge \mathfrak{t}(k+1) = \mathfrak{t}(k) \oplus \lambda(\&s, 1) \oplus \lambda(\&a[i], 1) \oplus \sigma(\&s, 1)$$
$$\oplus \lambda(\&a[i], 1) \oplus \lambda(\&a[i+1], 1) \oplus \sigma(\&a[i], 1))\,,$$
$$p_7 = true]$$

```
...
```

Figure 8. C-program fragment with symbolic tracefile

The length of the symbolic tracefile corresponds to the number of read/write references. If either the number of reads or the number of writes are of interest we selectively count elements (λ and σ). For instance the number of read references is $|\mathfrak{t}|_\lambda = 1 + 4\max(0, n-1)$, the number of write references is $|\mathfrak{t}|_\sigma = 2\max(0, n-1)$, and the overall number of memory references is given by $|\mathfrak{t}| = 1 + 6\max(0, n-1)$.

5. Symbolic Evaluation of Caches

A symbolic tracefile of a program describes all memory references (issued by the CPU) in chronological order. Based on the symbolic tracefile we can derive an analytical function over the input, which computes the number of hits. The symbolic tracefile contains all information to obtain the hit function. Moreover, the symbolic cache analysis is decoupled from the original program. Thus, our approach can be used to tailor the cache organization due to the needs of a given application. In the following we introduce two formalisms to compute a hit function for direct mapped and set associative data caches. To symbolically simulate the cache hardware, hit sets are introduced. Hit sets symbolically describe which addresses are held in the cache and keep track of the number of hits.

5.1. Direct Mapped Caches

Direct mapped caches are the easiest cache organization to analyze. For each item of data there is exactly one location in a direct mapped cache where it can be placed[5] and the cache contains *ns* cache lines. The size of a cache line, *cls*, determines the amount of data that is moved between main memory and the cache. In the following we introduce the symbolic cache evaluation of direct mapped caches with write through and no-write-allocate policy. Compare Section 2.

A new *cache evaluation operator* $\odot$ is defined to derive a *hit set* for a given tracefile $\mathfrak{t}$, where a hit set is a pair $H = (C, h)$. The first component of H is a *symbolic cache*, which is element of $\mathbb{A}$ – the second component represents the number of cache hits and is a symbolic expression.

Symbolic cache C of a hit set H has *ns* elements and each element corresponds to a cache line of the cache. More formally, the algebraic operation $C \oplus (r, \beta)$ loads the memory block with start address r into the cache whereby β is the index of the cache line. Note that when the CPU issues address r, the start address $\overline{r}$ of the corresponding memory block must be selected to describe the reference. Moreover, a *cache placement function* χ maps a reference to an index of cache C such that the load operation of reference r is written as $C \oplus (\overline{r}, \chi(r))$. In the following we assume that function χ is a modulo operation $\chi(r) = r \bmod (ns * cls)$.

5.1.1. *Definition of the cache evaluation operator*

Let $H_i \odot \mathfrak{t} = H_f$, where $H_i = (\perp_{ns}, 0)$ denotes the initial hit set, $H_f = (C_f, h_f)$ the final hit set, and $\mathfrak{t}$ the tracefile. The final hit set H_f is the analytical description of the number of cache hits and the final state of the cache. In the following we describe the operator $\odot$ inductively.

First, for an empty tracefile $\perp$ the hit set is

$$H \odot \perp = H.$$

Second, if a write reference is the first reference in the tracefile, it does not change the hit set at all and is to be removed.

$$H \odot \left(\perp \oplus \sigma(r) \bigoplus_{l=1}^{m} \mu_l \right) = H \odot \left(\perp \bigoplus_{l=1}^{m} \mu_l \right) \tag{2}$$

where μ_l is either a read reference $\lambda(r_l)$ or write reference $\sigma(r_l)$. Third, for read references a new hit set must be computed

$$H \odot \left(\perp \oplus \lambda(r) \bigoplus_{l=1}^{m} \mu_l \right) = (C', h') \odot \left(\perp \bigoplus_{l=1}^{m} \mu_l \right) \tag{3}$$

where $h' = h + d$ and

$$d = \begin{cases} 1, & \text{if } \rho\,(C, \chi(r)) = \overline{r} \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

and

$$C' = \begin{cases} C \oplus (\overline{r}, \chi(r)), & \text{if } d = 0 \\ C, & \text{otherwise} \end{cases} \tag{5}$$

Increment d is 1 if reference r is in the cache. Otherwise, d is zero and the reference r must be loaded. For loading data item with address r into the cache, C' is assigned the new symbolic value $C \oplus (\overline{r}, \chi(r))$.

In order to symbolically describe the conditional behavior of caches (data item is in the cache or not), we introduce a γ-function (see (Fahringer and Scholz, 1997)).

$$\gamma(c;\ x_1 = e_1, \ldots, x_k = e_k;\ x_1 = f_1, \ldots, x_k = f_k) \tag{6}$$

where $\gamma(c;\ x_1 = e_1, \ldots, x_k = e_k;\ x_1 = f_1, \ldots, x_k = f_k)$ is semantically equivalent to $(c \wedge x_1 = e_1 \wedge \cdots \wedge x_k = e_k) \vee (\neg\, c \wedge x_1 = f_1 \wedge \cdots \wedge x_k = f_k)$. c is a conditional expression and $\neg c$ the negation of c. Moreover, x_i $(1 \leq i \leq k)$ represent variable values and e_i, f_i are symbolic expressions.

Based on the definition of γ (6) we can aggregate formulas given in (3),(4), and (5). Depending on condition $\rho(C, \chi(r)) = \overline{r}$ either the number of cache hits h' is incremented by one or the symbolic cache is assigned a new symbolic value $C' = C \oplus (\overline{r}, \chi(r))$.

$$H \odot \left(\perp \oplus \lambda(r) \bigoplus_{l=1}^{m} \mu_l \right) = (C', h') \odot \left(\perp \bigoplus_{l=1}^{m} \mu_l \right), \tag{7}$$
$$\gamma\left(\rho(C, \chi(r)) = \overline{r}; C' = C, h' = h + 1; C' = C \oplus (\overline{r}, \chi(r)), h' = h\right)$$

Similar to tracefiles, hit sets are written as a pair. The first component of the pair symbolically describes the hit set. The second component contains constraints on variable values such as conditionals and recurrences stemming from loops.

Furthermore, for recursively-defined tracefiles we need to generalize hit sets to hit set recurrences. Let $t(k+1) = t(k) \bigoplus_{l=1}^{m} \mu_l(k)$ be the tracefile recurrence relation and H the initial hit set, the hit set recurrence is expressed by

$$\begin{aligned} H(0) &= H \odot t(0) \\ H(k+1) &= H(k) \odot \left(\perp \bigoplus_{l=1}^{m} \mu_l(k) \right) \end{aligned} \tag{8}$$

5.1.2. *Example*

For the sake of demonstration, we study our example of Figure 6 with a cache size of 4 cache lines and each cache line comprises one byte. The cache placement function $\chi(r)$ is $r \bmod 4$. It maps the memory addresses to slots of the cache. Moreover, all references are already transformed to word references and references `&n`, `&s`, and `&a[0]` are aligned to the first cache line. Note that in our example a word reference can only transfer one byte from the CPU to the cache and vice versa.

The initial hit set is $H_i = (\perp_4, 0)$. Based on the symbolic tracefile given in (1) the hit set recurrence is to be derived. First of all we apply operator $\odot$ to the hit set recurrence according to (8).

$$\begin{aligned} H_f &= H(z), \\ H(0) &= H_i \odot (\perp \oplus \lambda(\&n) \oplus \lambda(\&n+1) \oplus \lambda(\&n+2) \oplus \lambda(\&n+3)) \\ H(k+1) &= H(k) \odot (\perp \oplus \lambda(\&s) \oplus \lambda(\&a[k]) \oplus \sigma(\&s) \oplus \lambda(\&a[k]) \\ &\qquad \oplus \lambda(\&a[k+1]) \oplus \sigma(\&a[k])) \end{aligned}$$

The final hit set is given by $H_f = H(z)$ where $z = \max(0, n-1)$ is the highest index k of the recurrence and is determined by the loop condition. In the following we evaluate the boundary condition of the hit set recurrence. We successively apply the evaluation rule (7) of operator $\odot$ to the initial hit set $(\perp_4, 0)$.

$$\begin{aligned} &H(0) \\ &= (\perp_4, 0) \odot (\perp \oplus \lambda(\&n) \oplus \lambda(\&n+1) \oplus \lambda(\&n+2) \oplus \lambda(\&n+3)) \\ &= (\perp_4 \oplus (\&n, 0), 0) \odot (\perp \oplus \lambda(\&n+1) \oplus \lambda(\&n+2) \oplus \lambda(\&n+3)) \\ &= (\perp_4 \oplus (\&n, 0) \oplus (\&n+1, 1), 0) \odot (\perp \oplus \lambda(\&n+2) \oplus \lambda(\&n+3)) \\ &= (\perp_4 \oplus (\&n, 0) \oplus (\&n+1, 1) \oplus (\&n+2, 2), 0) \odot (\perp \oplus \lambda(\&n+3)) \end{aligned}$$

$$= (\bot_4 \oplus (\&n, 0) \oplus (\&n+1, 1) \oplus (\&n+2, 2) \oplus (\&n+3, 3), 0) \odot \bot$$
$$= (\bot_4 \oplus (\&n, 0) \oplus (\&n+1, 1) \oplus (\&n+2, 2) \oplus (\&n+3, 3), 0)$$

Note that condition $\rho(C, \chi(r)) = \overline{r}$ of rule (7) is false for all read references in the boundary condition. After evaluating the boundary condition there is still no cache hit and the cache is fully loaded with the contents of variable `n`. In the next step we analyze the loop iteration. We continue to apply operator $\odot$ to the recurrence relation.

$$\begin{aligned} H(k+1) &= (C'_k, h'_k) \\ &\odot (\bot \oplus \lambda(\&a[k]) \oplus \sigma(\&s) \oplus \lambda(\&a[k]) \oplus \lambda(\&a[k+1]) \oplus \sigma(\&a[k])) \wedge \\ &\quad \gamma(\rho(C_k, 0) = \&s; C'_k = C_k, h'_k = h_k + 1; \\ &\quad\quad C'_k = C_k \oplus (\&s, 0), h'_k = h_k) \end{aligned}$$

where C_k and h_k denote symbolic cache and number of hits in the kth iteration of the hit set recurrence. The global variable `s` is mapped to the first cache line. If the first slot of the cache contains the address of `s` then a cache hit occurs and the number of hits is incremented, otherwise the new element is loaded and the number of hits remains the same. We further apply operator $\odot$ and obtain

$$\begin{aligned} H(k+1) &= (C''_k, h''_k) \\ &\odot (\bot \oplus \sigma(\&s) \oplus \lambda(\&a[k]) \oplus \lambda(\&a[k+1]) \oplus \sigma(\&a[k])) \wedge \\ &\quad \gamma(\rho(C_k, 0) = \&s; C'_k = C_k, h'_k = h_k + 1; \\ &\quad\quad C'_k = C_k \oplus (\&s, 0), h'_k = h_k) \wedge \\ &\quad \gamma(\rho(C'_k, k \bmod 4) = k; C''_k = C'_k, h''_k = h'_k + 1; \\ &\quad\quad C''_k = C'_k \oplus (k, k \bmod 4), h''_k = h'_k). \end{aligned}$$

In the next step we eliminate the write reference $\sigma(\&s)$ according to rule (2) and further apply operator $\odot$ to $\lambda(\&a[k])$.

$$\begin{aligned} H(k+1) &= (C'''_k, h'''_k) \odot (\bot \oplus \lambda(\&a[k+1]) \oplus \sigma(\&a[k])) \wedge \\ &\quad \gamma(\rho(C_k, 0) = \&s; C'_k = C_k, h'_k = h_k + 1; \\ &\quad\quad C'_k = C_k \oplus (\&s, 0), h'_k = h_k) \wedge \\ &\quad \gamma(\rho(C'_k, k \bmod 4) = k; C''_k = C'_k, h''_k = h'_k + 1; \\ &\quad\quad C''_k = C'_k \oplus (k, k \bmod 4), h''_k = h'_k) \wedge \\ &\quad \gamma(\rho(C''_k, k \bmod 4) = k; C'''_k = C''_k, h'''_k = h''_k + 1; \\ &\quad\quad C'''_k = C''_k \oplus (k, k \bmod 4), h'''_k = h''_k) \\ &= (C''_k, h''_k + 1) \odot (\bot \oplus \lambda(\&a[k+1]) \oplus \sigma(\&a[k])) \wedge \\ &\quad \gamma(\rho(C_k, 0) = \&s; C'_k = C_k, h'_k = h_k + 1; \\ &\quad\quad C'_k = C_k \oplus (\&s, 0), h'_k = h_k) \wedge \end{aligned}$$

$$\gamma(\rho(C'_k, k \bmod 4) = k; C''_k = C'_k, h''_k = h'_k + 1;$$
$$C''_k = C'_k \oplus (k, k \bmod 4), h''_k = h'_k)$$

Here, we can simplify the γ-function. The contents of symbolic cache C''_k at $k \bmod 4$ is k because the reference $\&a[k]$ is loaded from the step before the previous one. Note that the write reference $\sigma(\&s)$ does not destroy the reference $\&a[k]$. In the last step the references $\lambda(\&a[k+1])$ and $\sigma(\&a[k])$ are evaluated. We continue with

$$\begin{aligned}
H(k+1) =& (C'''_k, h'''_k) \wedge \\
&\quad \gamma(\rho(C_k, 0) = \&s; C'_k = C_k, h'_k = h_k + 1; \\
&\qquad C'_k = C_k \oplus (\&s, 0), h'_k = h_k) \wedge \\
&\quad \gamma(\rho(C'_k, k \bmod 4) = k; C''_k = C'_k, h''_k = h'_k + 1; \\
&\qquad C''_k = C'_k \oplus (k, k \bmod 4), h''_k = h'_k) \wedge \\
&\quad \gamma(\rho(C''_k, k+1 \bmod 4) = k+1; C'''_k = C'_k, h'''_k = h''_k + 2; \\
&\qquad C'''_k = C'_k \oplus (k+1, k+1 \bmod 4), h'''_k = h''_k + 1) \\
=& (C''_k \oplus (k+1, k+1 \bmod 4), h''_k + 1) \wedge \\
&\quad \gamma(\rho(C_k, 0) = \&s; C'_k = C_k, h'_k = h_k + 1; \\
&\qquad C'_k = C_k \oplus (\&s, 0), h'_k = h_k) \wedge \\
&\quad \gamma(\rho(C'_k, k \bmod 4) = k; C''_k = C'_k, h''_k = h'_k + 1; \\
&\qquad C''_k = C'_k \oplus (k, k \bmod 4), h''_k = h'_k).
\end{aligned}$$

The third γ-function can be reduced since element $k+1$ has never been written before because condition $\rho(C''_k, k+1 \bmod 4) = k+1$ is false. The hit set recurrence is still conditional. Further investigations are necessary to derive a closed form for the number of hits. We know that the number of cache lines is four. We consider all four modulo classes of index k which for the given example results in an unconditional recurrence.

- $k \bmod 4 = 0$: The condition $\rho(C_k, 0) = \&s$ of the first γ-function is false since $\rho(C_k, 0)$ can be rewritten as k, if $k > 1$ or $\perp$ otherwise. The condition of second γ-function $\rho(C'_k, k \bmod 0) = k$ is false as well because the cache line has been loaded with the reference $\&s$ before. For the case $k \bmod 4 = 0$ the hit set recurrence is reduced to an unconditional recurrence.

$$H(k+1) = (C_k \oplus (\&s, 0) \oplus (k, 0) \oplus (k+1, 1), h_k + 1) \qquad (9)$$

- $k \bmod 4 = 1$: In the first γ-function the condition $\rho(C_k, 0) = \&s$ can never be true because in the previous step of the recurrence the cache line 1 has been loaded with the contents of $\&a[k-1]$. Furthermore, the element $\&a[k]$ has been fetched in the previous step and, therefore, the condition of the second γ-function evaluates to true and the hit set recurrence can be written as

$$H(k+1) = (C_k \oplus (\&s, 0) \oplus (k+1, 1), h_k + 2) \tag{10}$$

– $k \bmod 4 = 2$, $k \bmod 4 = 3$: For both cases the conditions of the γ-functions are true. The load reference $\&s$ does not interfere with $\&a[k]$ and $\&a[k+1]$. The recurrence is given by

$$H(k+1) = (C_k \oplus (\&s, 0) \oplus (k+1, 1), h_k + 3) \tag{11}$$

Now, we can extract the number of hits from hit sets (9), (10), (11). The modulo classes can be rewritten such that k is replaced by $4i$ and the modulo class.

$$\begin{aligned} h_0 &= 0 \\ h_{4i} &= h_{4i-1} + 1 \\ h_{4i+1} &= h_{4i} + 2 \\ h_{4i+2} &= h_{4i+1} + 3 \\ h_{4i+3} &= h_{4i+2} + 3 \end{aligned}$$

The boundary conditions stem from the number of hits of $H(0)$. The recurrence is linear and after resolving it, we obtain

$$h_z = \begin{cases} 9i & \text{if } \exists i : z = 4i, \\ 9i+1 & \text{if } \exists i : z = 4i+1, \\ 9i+3 & \text{if } \exists i : z = 4i+2, \\ 9i+6 & \text{otherwise.} \end{cases} \tag{12}$$

The index z of the final hit set $H_f = H(z)$ is determined by $z = \max(0, n-1)$. The analytical cache hit function h_z, given by (12), can be approximated by $\frac{9}{4}\max(0, n-1) = 2.25 \max(0, n-1)$.

In the example above the conditional recurrence collapsed to an unconditional one. In general, we can obtain closed forms only for specific – although very important – classes of conditional recurrences. If recurrences cannot be resolved, we employ approximation techniques as described in (Fahringer, 1998a).

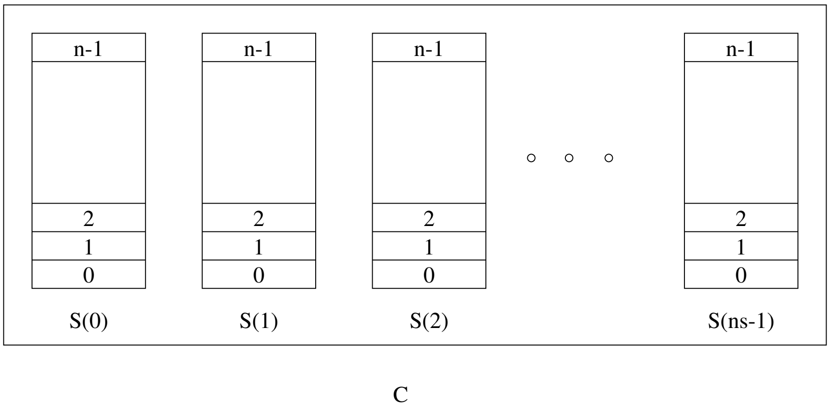


Figure 9. An n-way Set Associative Cache

5.2. Set Associative and Fully Associative Caches

In this section we investigate n-way set associative write through data caches with write-allocate policy and least recently used replacement (LRU) strategy. The organization of set-associative is more complex than direct mapped data caches due to placing a memory block to n possible locations in a slot (compare Section 2).

Similar to direct mapped caches we define a cache evaluation operator $\odot$ to derive a hit set for a given tracefile $\mathfrak{t}$. For set associative caches a hit set is a tuple $H = (C, h, \tau^{\max})$ where C is a symbolic cache, h the number of hits, and $\tau^{\max}$ is a symbolic counter that is incremented for every read or write reference. Note that the symbolic counter is needed to keep track of the least recently used reference of a slot. Figure 9 illustrates the symbolic representation of C for set associative caches. C is an array of ns slots. Each slot, denoted as $S(\iota)$ where $0 \leq \iota \leq ns - 1$, can hold n cache lines. Array C and slots $S(\iota)$ are elements of array algebra $\mathbb{A}$.

More formally, algebraic operation $S \oplus ((r, \tau), \beta)$ loads the memory block with start address r into set S whereby β is the index $(0 \leq \beta < n)$ and τ the current symbolic value of $\tau^{\max}$. Reading value r from S is denoted by $\rho_r(S, \beta)$ while reading the time stamp is written $\rho_\tau(S, \beta)$. A whole set is loaded into cache C via $C \oplus (S, \iota)$. Note that when the CPU issues address r, the start address $\overline{r}$ of the corresponding memory block must be selected to describe the reference. Similar to direct mapped caches, a cache placement function χ maps a memory reference to slot S such that the load operation of reference r is written as $C \oplus (\rho(C, \chi(r)) \oplus ((\overline{r}, \tau), \nu(r)))$ where $\nu(r)$ is a function determining the index of slot S according to the LRU strategy and is defined by

$$\nu(r) = \begin{cases} \min_\iota(\iota \mid \rho(S,\iota) = \perp), & \text{if there exists a } \iota \text{ such that } \rho(S,\iota) = \perp \\ \min_\tau(\iota \mid \rho_\tau(S,\iota)), & \text{otherwise.} \end{cases}$$

Note that the first case determines if there is a spare location in slot S. If so, the first spare location is determined by ν. The second case computes the least recently used cache line of slot S.

5.2.1. *Definition of the cache evaluation operator*

Let $H_i \odot \mathfrak{t} = H_f$, where $H_i = (\perp_{ns}, 0, 0)$ denotes the initial hit set, $H_f = (C_f, h_f, \tau_f)$ the final hit set, and $\mathfrak{t}$ the tracefile. The final hit set H_f is the analytical description of the number of cache hits and the final state of the cache. In the following we describe the operator $\odot$ inductively.

First, for an empty tracefile $\perp$ the hit set is

$$H \odot \perp = H.$$

Second, if a read or write operation $\mu(r)$ is the first memory reference in the tracefile, a new hit set is deduced as follows

$$H \odot \left(\perp \oplus \mu(r) \bigoplus_{l=1}^{m} \mu_l \right) = (C', h', \tau^{\max} + 1) \odot \left(\perp \bigoplus_{l=1}^{m} \mu_l \right) \tag{13}$$

where $h' = h + d$. The symbolic counter $\tau^{\max}$ is incremented by one. Furthermore, the slot of reference r is determined by $S = \rho\left(C, \chi(r)\right)$ and increment d is given by

$$d = \begin{cases} 1, & \text{if } \bigvee_{j=1}^{n} \rho_r\left(\rho\left(C,\chi(r)\right), j\right) = \overline{r} \\ 0, & \text{otherwise.} \end{cases} \tag{14}$$

If there exists an element in slot S, which is equal to $\overline{r}$, a cache hit occurs and increment $d = 1$ and reference r must be updated with a new time stamp.

$$C' = C \oplus \left(S', \chi(r)\right) \tag{15}$$

where

$$S' = S \oplus \left((\overline{r}, \tau^{\max}), \pi(r)\right). \tag{16}$$

Function $\pi(r)$ looks up the index, where reference r is stored in slot S; $\pi(r)$ can be described by a recurrence. If $d = 0$, a cache miss occurs and the reference is loaded into the cache

$$C' = C \oplus \left(S', \chi(r)\right) \tag{17}$$

where

$$S' = S \oplus ((\overline{r}, \tau^{\max}), \nu(r))\,. \qquad (18)$$

We can aggregate formulas (13) – (18) with γ-functions. Depending on condition $\bigvee_{j=1}^{2} \rho_r(\rho(C, \chi(r)), j) = \overline{r}$ a new element is updated with a new time stamp or loaded into the cache.

$$\begin{aligned} H \odot \left(\bot \oplus \mu(r) \bigoplus_{l=1}^{m} \mu_l \right) &= (C', h', \tau^{\max} + 1) \odot \left(\bot \bigoplus_{l=1}^{m} \mu_l \right), \\ & C' = C \oplus (S', \chi(r)) \wedge S = \rho(C, \chi(r)) \wedge \\ & \gamma(\textstyle\bigvee_{j=1}^{2} \rho_r(\rho(C, \chi(r)), j) = \overline{r}; \\ & \quad h' = h + 1, S' = S \oplus ((\overline{r}, \tau^{\max}), \pi(r))\,; \\ & \quad h' = h, S' = S \oplus ((\overline{r}, \tau^{\max}), \nu(r))\,, \\ & \qquad \gamma(\exists \iota : \rho(S, \iota) = \bot; \\ & \qquad\quad \nu(r) = \min_{\iota}(\iota \mid \rho(S, \iota) = \bot); \\ & \qquad\quad \nu(r) = \min_{\tau}(\iota \mid \rho_\tau(S, \iota)))) \end{aligned} \qquad (19)$$

Note that γ functions are nested in formula (19). A nested γ is recursively expanded (compare (6)) such that the expanded boolean expression is added to the corresponding true or false term of the higher-level γ-function. Furthermore, for recursively-described tracefiles we need to generalize hit sets to hit set recurrences (compare (8)).

5.2.2. *Example*

We symbolically evaluate the example of Figure 6 with a 2-way set associative cache and two slots and a cache line size of one byte. For this cache organization a word reference can transfer one byte from the CPU to the cache and vice versa. Thus, the cache size is the same as in Section 5.1, only the cache organization has changed. The cache placement function $\chi(r)$ is $r \bmod 2$. We assume that the references of the symbolic tracefile are already transformed to word references and references `&n`, `&s`, and `&a[0]` are aligned to the first slot.

The initial hit set is $H_i = (\bot_2, 0, 0)$. Based on the tracefile given in (1) the hit set recurrence is to be derived. Similar to example in Section 5.1 we apply operator $\odot$ to the hit set recurrence according to (8).

$$\begin{aligned} H(0) = (\bot_2 & \oplus (\bot_2 \oplus ((\&n, 0), 0) \oplus ((\&n + 2, 2), 1)) \\ & \oplus (\bot_2 \oplus ((\&n + 1, 1), 0) \oplus ((\&n + 3, 3), 1))\,, 0, 4)\,. \end{aligned}$$

For all read references in the boundary no cache hit occurred. The cache is loaded with the contents of variable `n` and the number of cache hits is zero. In the next step we evaluate the recurrence relation. We continue to apply operator $\odot$ according to rule (19).

$$
\begin{aligned}
H(k+1) = (C'_k, h'_k, \tau_k + 1) & \\
&\odot (\bot \oplus \lambda(\&a[k]) \oplus \sigma(\&s) \oplus \lambda(\&a[k]) \\
&\quad \oplus \lambda(\&a[k+1]) \oplus \sigma(\&a[k])) \wedge \\
& C'_k = C_k \oplus (S'_k, 0) \wedge S_k = \rho(C_k, 0) \wedge \\
& \gamma(\textstyle\bigvee_{j=1}^{2} \rho_r(\rho(C_k, 0), j) = \&s; \\
&\quad h'_k = h_k + 1, S'_k = S_k \oplus ((\&s, \tau_k), \pi'_k(0)); \\
&\quad h'_k = h_k, S'_k = S_k \oplus ((\&s, \tau_k), \nu'_k(0)), \\
&\quad \gamma(\exists \iota : \rho(S_k, \iota) = \bot; \\
&\qquad \nu'_k(0) = \min_{\iota}(\iota \mid \rho(S_k, \iota) = \bot); \\
&\qquad \nu'_k(0) = \min_{\tau}(\iota \mid \rho_\tau(S_k, \iota))))
\end{aligned}
\tag{20}
$$

where C_k, h_k, and τ_k denote symbolic cache, number of hits and time stamp counter of the kth iteration of the hit set recurrence. In order to keep the description of hit set recurrences as small as possible we rewrite the outer γ-function of (20) as P. We further apply operator $\odot$ and obtain

$$
\begin{aligned}
& H(k+1) = (C''_k, h''_k, \tau_k + 2) \\
&\quad \odot (\bot \oplus \sigma(\&s) \oplus \lambda(\&a[k]) \oplus \lambda(\&a[k+1]) \oplus \sigma(\&a[k])) \wedge P' \wedge \\
&\quad C''_k = C'_k \oplus (S''_k, k \bmod 2) \wedge S'_k = \rho(C'_k, k \bmod 2) \wedge \\
&\quad \gamma(\textstyle\bigvee_{j=1}^{2} \rho_r(\rho(C'_k, k \bmod 2), j) = k; \\
&\qquad h''_k = h'_k + 1, S''_k = S'_k \oplus \left((k, \tau_k + 1), \pi''_k(k)\right); \\
&\qquad h''_k = h'_k, S''_k = S'_k \oplus \left((k, \tau_k + 1), \nu''_k(k)\right), \\
&\qquad \gamma(\exists \iota : \rho(S'_k, \iota) = \bot; \\
&\qquad\quad \nu''_k(k) = \min_{\iota}(\iota \mid \rho(S'_k, \iota) = \bot); \\
&\qquad\quad \nu''_k(k) = \min_{\tau}(\iota \mid \rho_\tau(S'_k, \iota)))).
\end{aligned}
$$

In the next step we evaluate write reference $\sigma(\&s)$ and get

$$
\begin{aligned}
H(k+1) = (C'''_k, h''_k, \tau_k + 3) & \\
&\odot (\bot \oplus \lambda(\&a[k]) \oplus \lambda(\&a[k+1]) \oplus \sigma(\&a[k])) \wedge \\
& P' \wedge P'' \wedge \\
& C'''_k = C''_k \oplus (S'''_k, 0) \wedge S''_k = \rho(C''_k, 0) \wedge \\
& \gamma(\textstyle\bigvee_{j=1}^{2} \rho_r(\rho(C''_k, 0), j) = \&s; \\
&\quad S'''_k = S''_k \oplus \left((\&s, \tau_k + 2), \pi'''_k(0)\right);
\end{aligned}
$$

$$\begin{aligned}
&S_k''' = S_k'' \oplus \big((\&s, \tau_k + 2), \nu_k'''(0)\big)\,,\\
&\gamma(\exists\iota : \rho(S_k'', \iota) = \bot;\\
&\qquad \nu_k'''(0) = \min_{\iota}(\iota \mid \rho(S_k'', \iota) = \bot);\\
&\qquad \nu_k'''(0) = \min_{\tau}(\iota \mid \rho_\tau(S_k'', \iota)))).
\end{aligned}$$

Here, we can simplify the γ-function because variable **s** has been read within the current iteration of the loop without being overwritten in the cache, the condition of the outer γ-function evaluates to true. Hence, we obtain

$$\begin{aligned}
H(k+1) = {} & (C_k''', h_k'', \tau_k + 3)\\
&\qquad \odot (\bot \oplus \lambda(\&a[k]) \oplus \lambda(\&a[k+1]) \oplus \sigma(\&a[k])) \wedge\\
& P' \wedge P'' \wedge\\
& C_k''' = C_k'' \oplus (S_k''', 0) \wedge S_k'' = \rho(C_k'', 0) \wedge\\
& S_k''' = S_k'' \oplus \big((\&s, \tau_k + 2), \pi_k'''(0)\big)
\end{aligned}$$

Similar to the previous step we can reduce both γ-functions.

$$\begin{aligned}
H(k+1) = {} & (C_k^{(iv)}, h_k^{(iv)}, \tau_k + 4) \odot (\bot \oplus \lambda(\&a[k+1]) \oplus \sigma(\&a[k])) \wedge\\
& P' \wedge P'' \wedge P''' \wedge\\
& C_k^{(iv)} = C_k''' \oplus (S_k^{(iv)}, k \bmod 2) \wedge S_k''' = \rho(C_k''', k \bmod 2) \wedge\\
& h_k^{(iv)} = h_k'' + 1 \wedge S_k^{(iv)} = S_k''' \oplus \Big((k, \tau_k + 3), \pi_k^{(iv)}(k)\Big)
\end{aligned}$$

Read reference $\&a[k+1]$ produces a cache miss. Thus, the next step can be simplified too.

$$\begin{aligned}
H(k+1) = {} & (C_k^{(v)}, h_k^{(v)}, \tau_k + 5) \odot (\bot \oplus \sigma(\&a[k])) \wedge\\
& P' \wedge P'' \wedge P''' \wedge P^{(iv)} \wedge\\
& C_k^{(v)} = C_k^{(iv)} \oplus (S_k^{(v)}, k \bmod 2) \wedge\\
& S_k^{(iv)} = \rho(C_k^{(iv)}, k \bmod 2) \wedge\\
& h_k^{(v)} = h_k^{(iv)}, S_k^{(v)} = S_k^{(iv)} \oplus \Big((k, \tau_k + 4), \nu_k^{(v)}(k)\Big)\,,\\
& \gamma(\exists\iota : \rho(S_k^{(iv)}, \iota) = \bot;\\
&\qquad \nu_k^{(v)}(k) = \min_{\iota}(\iota \mid \rho(S_k^{(iv)}, \iota) = \bot);\\
&\qquad \nu_k^{(v)}(k) = \min_{\tau}(\iota \mid \rho_\tau(S_k^{(iv)}, \iota)))
\end{aligned}$$

In the last step the $\odot$ operator is applied to write reference $\&s$. It is a cache hit and we can eliminate the γ functions.

$$\begin{aligned}&H(k+1) = (C_k^{(vi)}, h_k^{(v)}, \tau_k + 6) \odot (\bot) \wedge P' \wedge P'' \wedge P''' \wedge P^{(iv)} \wedge P^{(v)} \wedge \\ &\quad C_k^{(vi)} = C_k^{(v)} \oplus (S_k^{(vi)}, k \bmod 2) \wedge S_k^{(v)} = \rho(C_k^{(v)}, k \bmod 2) \wedge \\ &\quad S_k^{(vi)} = S_k^{(v)} \oplus \Big((k, \tau_k + 5), \pi_k^{(vi)}(k)\Big)\end{aligned}$$

Arguments similar to those in Section 5.1 show that the conditions of the outer γ-function in P' and P'' are true for $k \geq 1$ and false for $k = 0$. Therefore, we can derive an unconditional recurrence relation for the number of cache hits ($k \geq 1$).

$$\begin{aligned}h_0 &= 0,\\ h_1 &= 1,\\ h_{k+1} &= h_k + 3.\end{aligned}$$

A closed form solution is given by ($k \geq 1$)

$$\begin{aligned}h_0 &= 0,\\ h_k &= 3k - 2.\end{aligned}$$

The index z of the final hit set $H_f = H(z)$ is determined by $z = \max(0, n-1)$. Thus, the analytical cache hit function is

$$h_z = \begin{cases} 0, & \text{for } n \leq 1,\\ 3n - 5, & \text{otherwise}\end{cases}$$

which shows that for our example the set associative cache performs better than the direct mapped cache of the same size.

6. Experimental Results

In order to assess the effectiveness of our cache hit prediction we have chosen a set of C-programs as a benchmark. We have adopted the symbolic evaluation framework introduced in (Fahringer and Scholz, 1997; Fahringer and Scholz, 1999) for the programming language C and the cache evaluation. The instrumentation was done by hand although an existing tool such as CPROF (Lebeck and Wood, 1994) could have instrumented the benchmark suite. Our symbolic evaluation framework computed the symbolic tracefiles and symbolically evaluated data caches. In order to compare predictions against real values we

```
int n; char a[100];

void sum()
{  int i;

   for(i=0;i<n-1;i++)
      a[i]=a[i]+a[i+1];
}
```

Figure 10. Benchmark Program

have measured the cache hits for a given cache and problem size. For measuring the empirical data we used the ACS cache simulator(Hunt, 1997). The programs of the benchmark suite were amended by the instrumentation routines of a provided library `bin2pdt`. The generated tracefiles were stored as PDATS (Johnson and Ha, 1994) files and later read by the ACS cache simulator.

The first program of the benchmark suite is example program in Figure 10. In contrast to Section 5 we have analyzed a direct mapped data cache with a cache line size greater than one byte. Furthermore, the first byte of array `a` is aligned to the first byte of an arbitrary cache line and the cache has more than one cache line. Our framework computes a cache hit function, where the number of cache hits is determined by $2(n-1) - \lceil \frac{n}{cls} \rceil$ and cls is the cache line size of 4, 8 and 16 bytes. Intuitively, we get $2(n-1)$ potential cache hits. For every new cache line a miss is implied. Therefore, we have to subtract the number of touched cache lines $\lceil \frac{n}{cls} \rceil$ from the number of read references.

Table Ia describes problem sizes n (n – first column), number of read (*R-Ref.* – second column) and write (*W-Ref.* – third column) references, and sum of read and write references (*T-Ref.* – fourth column). Tables Ib - Id compare measured with predicted cache hits for various data cache configurations (capacity/cache line size). For instance, D-Cache 256/4 corresponds to a cache with 256 bytes and a cache line size of 4 bytes. Every table comprises four columns. *M-Miss* tabulates the measured cache misses, *M-Hits* the measured cache hits, and *P-Hits* the predicted cache hits. In accordance with our accurate symbolic cache analysis we observe that the predicted hits are identical with the associated measurements for all cache configurations considered.

The same benchmark program was taken to derive the analytical cache hit function for set associative data caches. Note that the result is the same as for direct mapped caches. Even the empirical study with two way data caches of the same capacity delivered the same measurements given in Tables Ib - Id.

Table Ia. ExperimentsoftheC-programinFigure 10
ProblemSize

n	R-Ref.	W-Ref.	T-Ref.
10	19	9	28
100	199	99	298
1000	1999	999	2998
10000	19999	9999	29998

Table Ib. D-Cache256/4

n	M-Miss	M-Hit	P-Hit
10	4	15	15
100	26	173	173
1000	251	1748	1748
10000	2501	17498	17498

Table Ic. D-Cache16K/8

n	M-Miss	M-Hit	P-Hit
10	3	16	16
100	14	185	185
1000	126	1873	1873
10000	1251	18748	18748

Table Id. D-Cache64K/16

n	M-Miss	M-Hit	P-Hit
10	2	17	17
100	8	181	181
1000	64	1935	1935
10000	626	19373	19373

Table IIa. Experiment of `mcnt`

Problem Size n×m	R-Ref.	W-Ref.	T-Ref.
10×10	100	0	100
50×50	2500	0	2500
100×100	10000	0	10000
150×150	22500	0	22500
100×200	200000	0	200000

Table IIb. D-Cache 64K/16

n×m	M-Miss	M-Hit	P-Hit
10×10	50	50	50
50×50	1250	1250	1250
100×100	5000	5000	5000
150×150	11250	11250	11250
100×200	100000	100000	100000

The second program `mcnt` of the benchmark suite counts the number of negative elements of an $n \times m$-matrix. The counter is held in a register and does not interfere with the memory references of the matrix. Again, we analyzed the program with three different direct mapped cache configurations 256/4, 16K/8 and 64K/16. For the data cache sizes 256/4 and 16K/8 the cache hit function is zero. This is due

```
float f[N][N], u[N][N], new[N][N];

void jacobi_relaxation()
{
   int i,j;

   for(i=1;i<N-1;i++){
      for(j=1;j<N-1;i++){
         new[i][j] = 0.25 * (f[i][j] + u[i][j-1] +
                       u[i][j+1] + u[i-1][j] + u[i+1][j]);
      }
   }
}
```

Figure 11. Jacobi Relaxation

Table IIIa. Experiment of Jacobi Relaxation

Problem Size

n×n	R-Ref.	W-Ref.	T-Ref.
10×10	320	64	384
30×30	3920	784	4704
50×50	11520	2304	69120
90×90	48020	9604	288120

Table IIIb. D-Cache 256/4

n×n	M-Miss	M-Hit	P-Hit
10×10	222	98	98
30×30	2462	1458	1458
50×50	11332	188	188
90×90	48020	0	0

Table IIIc. D-Cache 512/4

n×n	M-Miss	M-Hit	P-Hit
10×10	222	98	98
30×30	2462	1458	1458
50×50	7102	4418	4418
90×90	47672	348	348

Table IIId. D-Cache 1K/4

n×n	M-Miss	M-Hit	P-Hit
10×10	222	98	98
30×30	2462	1458	1458
50×50	7102	4418	4418
90×90	32882	15138	15138

to the usage of `double` elements of the matrix. Only for the 64K/16 configuration the program can benefit from a data cache and the cache hits are given by $\lceil \frac{n \cdot m}{2} \rceil$. In Tables IIa and IIb the analytical function is compared to the measured results. Similar to the first benchmark the cache hit function remains the same for set associative data caches with the same capacity and the measurements are identical to Table IIb.

The third program `jacobi_relaxation` in Figure 11 calculates the Jacobi relaxation of an $n \times n$ `float` matrix. In a doubly nested loop the value of the resulting matrix `new` is computed. Both loop variables `i` and `j` are held in registers. Therefore, for direct mapped data caches interference can only occur between the read references of arrays `f` and `u`. We investigated the Jacobi relaxation code with a cache configuration of 256/4, 512/4 and 1K/4. The number of cache hits is given by

$$h_f = \begin{cases} 2(n-3)^2, & \text{if } 4 \le n \le \frac{ns}{2} \\ 4(n-3), & \text{if } \frac{ns}{2}+1 \le n \le ns-3 \\ 2(n-3), & \text{if } n = ns-2 \\ (n-3)^2, & \text{if } n = ns-1 \\ (n-3), & \text{if } ns \le n \le ns+1 \\ 0, & \text{otherwise} \end{cases}$$

according to Section 4 where ns is the number of cache lines. We compared the measured cache hits with the values of the cache hit function. The results of our experiments are shown in Tables IIIa - IIId.

The fourth program `gauss_jordan` in Figure 12 is a linear equation solver. Note that this program contains an if-statement inside the loop. Variables `i`, `ip`, `j`, and `k` are held in registers. For direct mapped data caches interference can only occur between the read references of array `a`. We have analyzed the Gauss Jordan algorithm with a cache configuration of 256/4.

We could classify three different ranges of n where the behavior of the hit function varies.

$$h_f = \begin{cases} n(2n^2-n-2), & \text{if } 2 \le n \le 32 \\ C(n), & \text{if } 33 \le n \le 128 \\ P(n), & \text{if } n \ge 129 \end{cases}$$

$C(n)$ must be described for each n in the range. Furthermore, $P(n)$ is a function containing 64 cases. Note that the number 64 stems from the number of cache lines. For sake of demonstration we only enlist some

```
float a[N,N];

void gauss_jordan(void)
{
   int i,ip,j,k;
   for(i=0;i<N;i++){
      for (ip=0;ip < N*(N-i);ip ++){
          j = ip / (N-i) ;
          k = i + ip - (ip / (N-i)) * (N-i);
          if (i != j){
             a[j,k] = a[j,k] -
                (a[j,i] * a[i,k]) *  a[i,i];
          }
      }
   }
}
```

Figure 12. Gauss Jordan

cases.

$$P(n) = \begin{cases} 2n^2 - 2n & \text{if } n \equiv 0 \bmod 64 \\ \frac{7813}{8192}n^3 + \frac{12255}{4096}n^2 + \frac{5296949}{8192}n - \frac{667183}{1024} & \text{if } n \equiv 1 \bmod 64 \\ \frac{961}{1024}n^3 + \frac{6111}{2048}n^2 + \frac{345331}{512}n - \frac{379593}{512} & \text{if } n \equiv 2 \bmod 64 \\ \frac{7813}{8192}n^3 + \frac{12255}{4096}n^2 + \frac{5755845}{8192}n - \frac{1696833}{2048} & \text{if } n \equiv 3 \bmod 64 \\ \frac{465}{512}n^3 + \frac{3023}{1024}n^2 + \frac{93219}{128} - \frac{58077}{64} & \text{if } n \equiv 4 \bmod 64 \\ \vdots & \vdots \\ \frac{7813}{8192}n^3 + \frac{12255}{4096}n^2 + \frac{5547181}{8192}n - \frac{2989859}{2048} & \text{if } n \equiv 63 \bmod 64 \end{cases}$$

In Tables IVa and IVb we compare the measured results with function values of the hit function.

The ability to determine accurate number of cache hits depends on the complexity of the input programs. The quality of our techniques to resolve recurrences, analyse complex array subscript expressions, loop bounds, branching conditions, interprocedural effects, and pointer operations impacts the accuracy of our cache hit function. For instance, if closed forms cannot be computed for recurrences, then we introduce approximations such as symbolic upper and lower bounds (Fahringer, 1998a). We have provided a detailed analysis of codes that can be handled by our symbolic evaluation in (Fahringer and Scholz, 1999).

Table IVa. Experiment of Gauss Jordan

Problem Size

n×n	R-Ref.	W-Ref.	T-Ref.
200×200	15999600	3999900	19999500
400×400	127999200	31999800	159999000
600×600	431998800	107999700	539998500
2000×2000	15999996000	3999999000	19999995000

Table IVb. D-Cache 256/4

D-Cache 256/4

n×n	M-Hit	P-Hit
200×200	7060901	7060901
400×400	47324017	47324017
600×600	184781660	184781660
2000×2000	5825464317	5825464317

7. Related Work

Traditionally, the analysis of cache behavior for worst-case execution time estimates in real-time systems (Park, 1993; Puschner and Koza, 1989; Chapman et al., 1996) was far too complex. Recent research (Arnold et al., 1994) has proposed methods to estimate tighter bounds for WCET in systems with caches. Most of the work has been successfully applied to instruction caches (Liu and Lee, 1994) and pipelined architectures (Healy et al., 1995). Lim et al. (1994) extend the original timing schemas, introduced by Puschner and Koza (1989), to handle pipelined architectures and cached architectures. Nearly all of these methods rely on frequency annotations of statements. If the programmer provides wrong annotations, the quality of the prediction can be doubtful. Our approach does not need user (programmer) interaction since it derives all necessary information from the program's code[6] and it does not restrict program structure such as (Ghosh et al., 1997).

A major component of the framework described in (Arnold et al., 1994) is a static cache simulator (Mueller, 1997) realized as a data flow analysis framework. In (Alt et al., 1996) an alternate formalization which relies on the technique of abstract interpretation is presented. Both of these approaches are based on data-flow analysis but do not properly model control flow. Among others, they cannot deal with dead paths and zero-trip loops all of which are carefully considered by our symbolic evaluation framework (Fahringer and Scholz, 1997; Blieberger, 1997).

Implicit path enumeration (IPET) (Li et al., 1995; Li et al., 1996) allows to express semantic dependencies as constraints on the control flow graph by using integer linear programming models, where frequency annotations are still required. Additionally, the problem of IPET is that it only counts the number of hits and misses and cannot keep track of the history of cache behavior. Only little work has been done to introduce history variables (Ottosson and Sjoedin, 1997). While IPET can model if-statements correctly (provided the programmer supplies correct frequency annotations), it lacks adequate handling of loops. Our symbolic tracefiles exactly describe the data and control flow behavior of programs which among others enables precise modeling of loops. In (Theiling and Ferdinand, 1998) IPET was enriched with information of the abstract interpretation described in (Alt et al., 1996).

A graph coloring approach is used in (Rawat, 1993) to estimate the number of cache misses for real-time programs. The approach only supports data caches with random replacement strategy[7]. It employs standard data-flow analysis and requires compiler support for placing variables in memory according to the results of the presented algorithm. Alleviating assumptions about loops and cache performance improving transformations such as loop unrolling make their analysis less precise than our approach. It is assumed that every memory reference that is accessed inside of a loop at a specific loop iteration causes a cache miss. Their analysis does not consider that a reference might have been transmitted to the cache due to a cache miss in a previous loop iteration.

Much research has been done to predict cache behavior in order to support performance oriented program development. Most of these approaches are based on estimating cache misses for loop nests. Ferrante et al. (1991) compute an upper bound for the number of cache lines accessed in a sequential program which allows them to guide various code optimizations. They determine upper bounds of cache misses for array references in innermost loops, the inner two loops, and so on. The number of cache misses of the innermost loop that causes the cache to overflow is multiplied by the product of the number of iterations of the overflow loop and all its containing loops. Their approximation technique may entail polynomial evaluations and suffers by a limited control flow modeling (unknown loop bounds, branches, etc.).

Lam et al. (1991) developed another cache cost function based on the number of loops carrying cache reuse which can either be temporal (relating to the same data element) or spatial (relating to data elements in the same cache line). They employ a reuse vector space in combination with localized iteration space. Cross interference (elements from different arrays displace each other from the cache) and self interferences (interference between elements of the same array) are modeled. Loop bounds are not considered even if they are known constants.

Temam et al. (1994) examine the source code of numerical codes for cache misses induced by loop nests.

Fahringer (1996; 1997) implemented an analytical model that estimates the cache behavior for sequential and data parallel Fortran programs based on a classification of array references, control flow modeling (loop bounds, branches, etc. are modeled by profiling), and an analytical cache cost function.

Our approach goes beyond existing work by correctly modeling control flow of a program even in the presence of program unknowns and branches such as if-statements inside of loops. We cover larger classes of programming languages and cache architectures, in particular data caches, instruction caches and unified caches including direct mapped caches, set associative, and fully associative caches. We can handle most important cache replacement and write policies. Our approach accurately computes cache hits, whereas most other methods are restricted to approximations.

Closed form expressions and conservative approximations can be found according to the steps described in Section 1.

Symbolic evaluation can also be used for WCET analysis without caching (Blieberger, 1997), thereby solving the dead paths problem of program path analysis (Park, 1993; Altenbernd, 1996). In addition, it can be used for performing "standard" compiler optimizations, thus being an optimal framework for integrating optimizing compilers and WCET analysis (compare Engblom et al. (1998) for a different approach).

8. Conclusion and Future Work

In this paper we have described a novel approach for estimating cache hits as implied by programs written in most procedural languages (including C, Ada, and Fortran). We generate a *symbolic tracefile* for the input program based on *symbolic evaluation* which is a static technique to determine the dynamic behavior of programs. Symbolic expressions and recurrences are used to describe all memory references in a program which are then stored chronologically in the symbolic tracefile. A cache hit function for several cache architectures is computed based on a *cache evaluation technique*.

In the following we describe the contributions of our approach: While most other research targets upper bounds for cache misses, we focus on deriving the accurate number of cache hits. We can automatically determine an analytical cache hit function at compile-time without user interaction. Symbolic evaluation enables us to represent the cache hits as a function over program unknowns (e.g. input data). Our approach allows a comparison of various cache organizations for a given program with respect to cache performance. We can easily port our techniques across different architectures by strictly separating machine specific parameters (e.g. cache line sizes, replacement strategies, etc.) from machine-independent parameters (e.g. loop bounds, array index expressions, etc.). A novel approach has been introduced to model arrays as part of symbolic evaluation which maintains the history of previous array references.

We have shown experiments that demonstrate the effectiveness of our approach. The predicted cache behavior for our example codes perfectly match with the measured data.

Although we have applied our techniques to direct mapped data caches with write through and no write-allocate policy and set associative data caches with write through and write-allocate policy, it is possible to generalize our approach for other cache organizations as well. Moreover, our approach is also applicable for instruction and unified caches.

In addition our work can be extended to analyze virtual memory architectures. A combined analysis of caching and pipelining via symbolic evaluation will be conducted in the near future (compare Healy et al. (1999) for a different approach).

The quality of our cache hit function depends on the complexity (e.g. recurrences, interprocedural effects, pointers, etc.) of the input programs. If, for instance, we cannot find closed forms for recurrences, then we employ approximations such as upper bounds. We are currently extending our symbolic evaluation techniques to handle larger classes of input programs. Additionally, we are building a source-code level instrumentation system for the SPARC processor architecture. We investigate the applicability of our techniques for multi-level data and instruction caches. Finally, we are in the process to conduct more experiments with larger codes.

Notes

[1] A cache miss occurs if referenced data is not in cache and needs to be loaded from main memory.

[2] A cache hit occurs if referenced data is in cache.

[3] Symbolic evaluation is not to be confused with symbolic execution (see e.g. (King, 1976)).

[4] All variables which are written inside a loop – including the loop variable – are called recurrence variables.

[5] A slot consists of one cache line. See Section 2.

[6] Clearly our approach cannot bypass undecidability.

[7] Random replacement seems very questionable for real-time applications because of its indeterministic behavior.

References

Alt, M., C. Ferdinand, F. Martin, and R. Wilhelm: 1996, 'Cache Behaviour Prediction by Abstract Interpretation'. In: *Proc. of Static Analysis Symposium*. pp. 52–66.

Altenbernd, P.: 1996, 'On the False Path Problem in Hard Real-Time Programs'. In: *Proc. Euromicro Workshop on Real-Time Systems*.

Arnold, R., F. Mueller, D. Whalley, and M. Harmon: 1994, 'Bounding Worst-Case Instruction Cache Performance'. In: *Proc. of the IEEE Real-Time Systems Symposium*. pp. 172–181.

Blieberger, J.: 1994, 'Discrete Loops and Worst Case Performance'. *Computer Languages* **20**(3), 193–212.

Blieberger, J.: 1997, 'Data-Flow Frameworks for Worst-Case Execution Time Analysis'. (submitted).

Blieberger, J. and B. Burgstaller: 1998, 'Symbolic Reaching Definitions Analysis of Ada Programs'. In: *Proc. of the Ada-Europe International Conference on Reliable Software Technologies*. pp. 238–250.

Blieberger, J., B. Burgstaller, and B. Scholz: 1999, 'Interprocedural Symbolic Analysis of Ada Programs with Aliases'. In: *Proc. of the Ada-Europe International Conference on Reliable Software Technologies*. pp. 136–145.

Blieberger, J. and R. Lieger: 1996, 'Worst-Case Space and Time Complexity of Recursive Procedures'. *Real-Time Systems* **11**(2), 115–144.

Callahan, D., K. Kennedy, and A. Portfield: 1990, *Analyzing and Visualizing Performance of Memory Hierachies, Instrumentation for Visualization*. ACM Press.

Chapman, R., A. Burns, and A. Wellings: 1996, 'Combining Static Worst-Case Timing Analysis and Program Proof'. *The Journal of Real-Time Systems* **11**(2), 145–171.

Cheatham, T., H. Holloway, and J. Townley: 1979, 'Symbolic Evaluation and the Analysis of Programs'. *IEEE Trans. Software Eng.* **5**(4), 403–417.

Dijkstra, E.: 1976, *A discipline of programming*. New Jersey: Prentice Hall.

Engblom, J., P. Altenbernd, and A. Ermedahl: 1998, 'Facilitating Worst-Case Execution Times Analysis for Optimized Code'. In: *Proc. Euromicro Workshop on Real-Time Systems*.

Fahringer, T.: 1996, *Automatic Performance Prediction of Parallel Programs.* Boston, USA: Kluwer Academic Publishers.

Fahringer, T.: 1997, 'Estimating Cache Performance for Sequential and Data Parallel Programs'. In: *Proc. of the International Conference and Exhibition on High-Performance Computing and Networking.*

Fahringer, T.: 1998a, 'Efficient Symbolic Analysis for Parallelizing Compilers and Performance Estimators'. *Journal of Supercomputing, Kluwer Academic Publishers* **12**(3).

Fahringer, T.: 1998b, 'Symbolic Analysis Techniques for Program Parallelization'. *Journal of Future Generation Computer Systems, Elsevier Science, North-Holland* **13**(1997/98), 385–396.

Fahringer, T. and B. Scholz: 1997, 'Symbolic Evaluation for Parallelizing Compilers'. In: *Proc. of the ACM International Conference on Supercomputing.*

Fahringer, T. and B. Scholz: 1999, 'A Unified Symbolic Evaluation Framework for Parallelizing Compilers'. Technical Report AURORA TR1999-15 (http://www.vcpc.univie.ac.at/aurora/), University of Vienna, Austria.

Ferrante, J., V. Sarkar, and W. Trash: 1991, 'On estimating and enhancing cache effectiveness'. In: *Proc. of the Workshop on Languages and Compilers for Parallel Computing.*

Gerlek, M. P., E. Stoltz, and M. Wolfe: 1995, 'Beyond Induction Variables: Detecting and Classifying Sequences Using a Demand-Driven SSA Form'. *ACM Transactions on Programming Languages and Systems (TOPLAS)* pp. 17(1):85–122.

Ghosh, S., M. Martonosi, and S. Malik: 1997, 'Cache Miss Equations: An Analytical Representation of Cache Misses'. In: *International Conference on Supercomputing.*

Goldberg, A. and J. Hennessy: 1991, 'Performance Debugging Shared Memory Multiprocessor Programs with MTOOL'. In: *Proc. of the Supercomputing Conference.*

Haghighat, M. and C. Polychronopoulos: 1996, 'Symbolic Analysis for Parallelizing Compilers'. *ACM Transactions on Programming Languages and Systems* **18**(4), 477–518.

Healy, C., D. Whalley, and M. Harmon: 1995, 'Integrating the Timing Analysis of Pipelining and Instruction Caching'. In: *Proc. of the IEEE Real-Time Systems Symposium.* pp. 288–297.

Healy, C. A., R. D. Arnold, F. Mueller, D. Whalley, and M. G. Harmon: 1999, 'Bounding Pipeline and Instruction Cache Performance'. *IEEE Transactions on Computers* **48**(1).

Hennessy, J. and D. Patterson: 1990, *Computer Architecture - A Quantitative Approach.* San Mateo, CA: Morgan Kaufmann.

Hunt, B.: 1997, 'Acme Cache Simulator'. Parallel Architecture Research Laboratory (PARL), Klipsch School of Electrical and Computer Engineering, New Mexico. URL http://tracebase.nmsu.edu/~acme/acs.html.

Johnson, E. and J. Ha: 1994, 'PDATS Lossless Address Trace Compression For Reducing File Size and Access Time'. In: *Proc. of the IEEE International Phoenix Conference on Computers and Communication.*

King, J.: 1976, 'Symbolic Execution and Program Testing'. *Commun. ACM* **19**(7), 385–394.

Lam, M., E. Rothberg, and M. Wolf: 1991, 'The Cache Performance and Optimization of Blocked Algorithms'. In: *Proc. of the Int. Conference on Architectural Support for Prog. Languages Operating Systems.*

Lebeck, A. and D. Wood: 1994, 'Cache Profiling and the SPEC Benchmarks: A Case Study'. *IEEE Computer* **27**(10).

Li, Y., S. Malik, and A. Wolfe: 1995, 'Performance Estimation of Embedded Software with Instruction Cache Modeling'. In: *Proc. of the ACM/IEEE International Conference on Computer-Aided Design*.

Li, Y., S. Malik, and A. Wolfe: 1996, 'Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches'. In: *Proc. of the IEEE Real-Time Systems Symposium*.

Lim, S., Y. Bae, G. Jang, B. Rhee, S. Min, C. Park, H. Shin, K. Park, and C. Kim: 1994, 'An Accurate Worst Case Timing Analysis Technique for RISC Processors'. In: *Proc. of the IEEE Real-Time Systems Symposium*. pp. 97–108.

Liu, J. and H. Lee: 1994, 'Deterministic Upperbounds of the Worst-Case Execution Times of Cached Programs'. In: *Proc. of the IEEE Real-Time Systems Symposium*. pp. 182–191.

Lueker, G. S.: 1980, 'Some Techniques for Solving Recurrences'. *ACM Computing Surveys* **12**(4), 419–435.

Mueller, F.: 1997, 'Generalizing Timing Predictions to Set-Associative Caches'. In: *Proc. of the EuroMicro, Workshop on Real Time Systems*.

Nilsen, K. and B. Rygg: 1995, 'Worst-Case Execution Time Analysis on Modern Processors'. In: *Proc. of ACM SIGPLAN, Workshop on Languages, Computer and Tool Support for Real-Time Systems*. pp. 20–30.

Ottosson, G. and M. Sjoedin: 1997, 'Worst Case Execution Time Analysis for Modern Hardware Architectures'. In: *Proc. of ACM SIGPLAN, Workshop on Languages, Compilers and Tools for Real-Time Systems*.

Park, C.: 1993, 'Predicting Program Execution Times by Analyzing Static and Dynamic Program Paths'. *The Journal of Real-Time Systems* **5**, 31–62.

Ploedereder, E.: 1980, 'A Semantic Model for the Analysis and Verification of Programs in General, Higher-Level Languages'. Ph.D. thesis, Harvard University.

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

Rawat, J.: 1993, 'Static Analysis of Cache Performance for Real-Time Programming'. Master's thesis, Iowa State University of Science and Technology, Dept. of Computer Science.

Scheibl, M., A. Celic, and T. Fahringer: 1996, 'Interfacing Mathematica from the Vienna Fortran Compilation System'. Technical Report, Institute for Software Technology and Parallel Systems, Univ. of Vienna.

Smith, A.: 1982, 'Cache Memories'. *Computing Surveys* **14**(3).

Temam, O., C. Fricker, and W. Jalby: 1994, 'Cache Interference Phenomena'. In: *Proc. of the ACM SIGMETRICS Conference*.

Theiling, H. and C. Ferdinand: 1998, 'Combining Abstract Interpretation and ILP for Microarchitecture Modelling and Program Path Analysis'. In: *Proc. of the IEEE Real-Time Systems Symposium*.

Authors' Vitae

Johann Blieberger
Johann Blieberger is currently Assistant Professor at the Department of Computer-Aided Automation, Technical University of Vienna.

His research interests include areas of analysis of algorithms and data structures, theoretical analysis of scheduling algorithms, worst-case execution time analysis, real-time programming languages, and symbolic evaluation.

He studied Computer Science at the TU Vienna and received his doctoral degree in 1985. From 1984 to 1988 he has been with Siemens AG Austria. He is a member of ACM and IEEE Computer Society.

Readers may contact Johann Blieberger at the Department of Computer-Aided Automation, Technical University Vienna, Treitlstr. 1, A-1040 Vienna, Austria; WWW: http://www.auto.tuwien.ac.at/~blieb and e-mail: blieb@auto.tuwien.ac.at.

Thomas Fahringer
Thomas Fahringer received a Masters degree in 1988 and a Ph.D. in 1993, all in Computer Science from the Technical University of Vienna, Austria. He was a visiting scientist at the Engineering Design Research Center at Carnegie Mellon University from 1988 - 1990. Since 1990 he is an Assistant Professor of Computer Science, and since 1998, associate professor, at the Institute for Software Technology and Parallel Systems, University of Vienna. His research focuses on software tools for distributed and parallel architectures in particular optimizing and parallelizing compilers, symbolic program and performance analysis, performance-oriented program development, and real-time systems. Readers may contact Fahringer at the Institute for Software Technology and Parallel Systems, University of Vienna, Liechtensteinstr. 22, A-1090 Vienna, Austria; WWW: http://www.par.univie.ac.at and e-mail: tf@par.univie.ac.at.

Bernhard Scholz
Bernhard Scholz is going to enrol a position as an Assistant Professor of Computer Science at the Dept. of Computer Languages, TU Vienna. His research focuses on symbolic analysis of programs, compiler, real-time programming and parallel computing. He studied computer science at the TU Vienna and received his Master degree in 1997. From 1997 to 1998 he has been with Baring Asset Management, London. Since 1998 he has joined Prof. Zima's research group at the Institute for Software Technology and Parallel Systems, University of Vienna. Readers may contact Scholz at the Dept. of Computer Languages,

A. B. C. .

Figure 13. A) Johann Blieberger. B) Thomas Fahringer. C) Bernhard Scholz.

Technical University Vienna, Argentinierstr. 8, A-1040 Vienna, Austria.
e-mail: scholz@par.univie.ac.at