Cache Behavior Prediction by Abstract Interpretation

Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm

Universität des Saarlandes / Fachbereich Informatik
Postfach 15 11 50 / D-66041 Saarbrücken / Germany
Phone: +49 681 302 5573 Fax: +49 681 302 3065
{alt,ferdi,florian,wilhelm}@cs.uni-sb.de
http://www.cs.uni-sb.de/users/{alt,ferdi,martin,wilhelm}

Abstract. Abstract Interpretation is a technique for the static analysis of dynamic properties of programs. It is semantics based, that is, it computes approximative properties of the semantics of programs. On this basis, it allows for correctness proofs of analyzes. It thus replaces commonly used ad hoc techniques by systematic, provable ones, and it allows the automatic generation of analyzers from specifications as in the Program Analyzer Generator, PAG.
In this paper, abstract interpretation is applied to the problem of predicting the cache behavior of programs. Abstract semantics of machine programs for different types of caches are defined which determine the contents of caches. The calculated information allows to sharpen worst case execution times of programs by replacing the worst case assumption 'cache miss' by 'cache hit' at some places in the programs. It is possible to analyse instruction, data, and combined instruction/data caches for common (re)placement and write strategies. The analysis is designed generic with the cache logic as parameter.

Keywords: abstract interpretation, program analysis, cache memories, real time applications, worst case execution time prediction.

1 Cache Memories and Real-Time Applications

Caches are used to improve the access times of fast microprocessors to relatively slow main memories. They can reduce the number of cycles a processor is waiting for data by providing faster access to recently referenced regions of memory[1].

Programs with hard real time constraints have to be subjected to a schedulability analysis by the compiler [24, 9]; it has to be determined whether all timing constraints can be satisfied. WCETs (Worst Case Execution Times) for processes have to be used for this. For hardware with caches, the appropriate worst

[1] Hennessy and Patterson [11] describe typical values for caches in 1990 workstations and minicomputers: Hit time 1–4 clock cycles (normally 1); Miss penalty 8–32 clock cycles.

case assumption is that all accesses miss the cache. This is an overly pessimistic assumption which leads to a waste of hardware resources.

Correct information about the contents of the cache at program points could help to sharpen the worst case execution times. Such information can be computed by an abstract interpretation statically collecting information about cache contents. The way this information is computed, an abstraction of the concrete semantics of the programs, depends on the type of cache regarded and the cache replacement strategy. Several abstract semantics are described, for different types of caches.

2 Overview

In the following section we briefly sketch the underlying theory of abstract interpretation and present the analysis tool `PAG`. Section 4 describes related approaches for the prediction of cache behavior.

Cache memories are briefly described in section 5. In section 6 we give a semantics for programs that reflects only memory accesses (to fixed addresses) and its effects on cache memories for common cache architectures. In section 7 we present *must analyses* that computes a subset of the memory blocks that must be in the cache and *may analyses* that computes a superset of the memory blocks that may be in the cache and describe how the results of the analyses can be interpreted.

The functional and the callstring approach developed for the abstract interpretation of programs with recursive procedures is used in section 8 to compute the behavior of memory references within loops by combining the results of the must and may analyses. An example is given in section 9.

In section 10 we describe how the analyses can be transferred to the analysis of data caches or combined instruction/data caches for a restricted class of programs, and how a combination of the must and may analyses can be used for the analysis of writes to the cache for common cache organizations.

3 Program Analysis by Abstract Interpretation

Program analysis is a widely used technique to determine runtime properties of a given program without actually executing it. There is a common theory for all program analyses called abstract interpretation [6, 7, 8]. With this theory, termination and correctness of a program analysis can be easily proven. According to this theory a program analysis is determined by an *abstract semantics*.

The program analyzer generator `PAG` [1, 2] offers the possibility to generate a program analyzer from a description of the abstract domain and of the abstract semantic functions. These descriptions are given in two high level languages, which support the description even of complex domains and semantic functions. The domain can be constructed from some simple sets like integers by operators like building power sets or by constructing of functions. The semantic functions are described in a functional language which combines high expressiveness with efficient implementation. Additionally one has to write a join function combining two incoming values of the domain into a single one. This function is applied whenever a point in the program has two (or more) possible execution predecessors.

For the analysis of programs with (recursive) procedures `PAG` supports the *functional approach* and the *call string approach* [22].

4 Related Work

The computation of WCETs for real-time programs is an ongoing research activity. Park and Shaw [18] describe a method to derive WCETs from the structure of programs. In [20] Puschner and Koza propose methods to guide the computation of WCETs by user annotations. Both approaches do not take cache behavior into account.

The possibilities to use optimizing compilers to improve cache performance of programs has extensively been studied [13, 14, 19, 26]. But all the proposed program transformations and code reorganizations do not necessarily help in computing the worst case execution time of a program. An overview of 'Cache Issues in Real-Time Systems' is given in [4]. We restrict our examination here to the intrinsic cache behavior. In [16, 15] Arnold, Mueller, Whalley, and Harmon describe a data flow analysis for the prediction of instruction cache behavior of programs for direct mapped caches.

A method for the data cache analysis by graph coloring is described in [17, 21]. Similar to the Chow-Hennessy register allocator, variables are allocated to cache lines. The objective of the analysis is to show that throughout the live range of a cache line, no other memory access interferes with this particular cache line.

In [12] a general framework is described for the computation of WCETs of programs in the presence of pipelines and cache memories. Two kinds of pipeline and cache state information are associated with every program construct for which timing equations can be formulated. One describes the pipeline and cache state when the program construct is finished. The other one can be combined with the state information from the previous construct to refine the WCET computation for that program construct. An approximation to the solution for the set of timing equations has been proposed.

5 Cache Memories

A cache can usually be characterized by three major parameters:

- *capacity* is the number of bytes it may contain.
- *line size* (also called block size) is the number of contiguous bytes that are transferred from memory on a cache miss. The cache can hold at most $\frac{capacity}{line\ size}$ blocks.

– *associativity* is the number of cache locations where a particular block may reside.
$\frac{capacity}{line\ size * associativity}$ is the number of *sets*[2] of a cache.

If a block can reside in any cache location, then the cache is called *fully associative*. If a block can reside in exactly one location, then it is called *direct mapped*. If a block can reside in exactly A locations, then the cache is called *A-way set associative* [23]. The fully associative and the direct mapped cache are special cases of the A-way set associative cache where $A = n$ and $A = 1$ rsp.

6 Concrete Semantics

In the following, we consider a cache as a set of cache lines $L = \{l_0, \ldots, l_{n-1}\}$, where $n = \frac{capacity}{line\ size}$. The store $S = \{s_0, \ldots, s_{m-1}\}$ is divided into blocks of size *line size*, so that one memory block can be transferred into one cache line. The locations where a memory block may reside in the cache depend on the level of associativity of the cache memory. This is formalized by a relation between cache lines and memory blocks.

Definition 1 mapping relation. A mapping relation $\mathcal{M} \subseteq L \times S$ is a subset of the cartesian product of caches and stores. It defines the cache lines that may hold a particular memory block. The meaning of an element $(l, s) \in \mathcal{M}$ of a mapping relation is: the memory block s may be stored in cache line l.

Example 1 special mappings. The following mappings describe common cache organizations:

– *fully associative mapping*: $\mathcal{M}_{assoc} = L \times S$. A memory block may be held by any cache line
– *direct mapping*: $\mathcal{M}_{direct} = \{(l_i, s_x) \mid i = (x\%n), x \in \{0, \ldots, m-1\}\}$. % denotes the modulo division. A memory block may reside in exactly one cache line[3]
– *A-way set associative mapping*:
$\mathcal{M}_{A-way} = \bigcup_{a=0}^{A-1} \{(l_i, s_x) \mid i = (x\%(n/A)) * A + a, x \in \{1, \ldots, m\}\}$.
A memory block may reside in exactly A cache lines

For the absence of any memory block in a cache line, we introduce a new element I; $S' = S \cup \{I\}$.

Definition 2 concrete cache state. A *(concrete) cache state* is a mapping $c : L \to S'$. C_c denotes the set of all concrete cache states.

---
[2] A *set* can be considered as a fully associative subcache.
[3] The 'address' within the cache (and thereby the cache line) is usually determined by the lowest N bits of the address of a memory block, where $capacity = 2^N$.

In the case of an A-way set associative or fully associative cache, a cache line has to be selected for replacement when the cache is full and the processor requests further data. This is done according to a replacement strategy. Common strategies are *LRU* (Least Recently Used), *FIFO* (First In First Out), and *random*.

The replacement strategy is integrated into the *update* function that models the effects of referencing the cache.

Definition 3 cache update. A cache update function $\mathcal{U}_{\mathcal{M}} : C_c \times S \rightarrow C_c$ is a function from a concrete cache state and a memory block to a concrete cache state.

Accesses to caches can be modeled in the following ways:

- *direct mapped cache*: $\mathcal{U}_{\mathcal{M}_{direct}}(c, s_x) = c[l_i \mapsto s_x]$ where $i = (x\%n)$; where $c(l) = s$ means cache line l holds memory block s; and $c(l) = I$ means cache line l holds no valid memory block.
- *fully associative cache with LRU replacement strategy*:

$$\mathcal{U}_{\mathcal{M}_{assoc}}(c, s) = \begin{cases} [\, l_0 \mapsto s, \\ \quad l_i \mapsto c(l_{i-1}) \mid i = 1 \ldots h, \\ \quad l_i \mapsto c(l_i) \mid i = h+1 \ldots n-1 \;]; \text{if } \exists l_h : c(l_h) = s \\ [\, l_0 \mapsto s, \\ \quad l_i \mapsto c(l_{i-1}) \text{ for } i = 1 \ldots n-1 \,]; \text{otherwise} \end{cases}$$

  The order of the cache lines $l_1, l_2, \ldots$ is used to express the relative age of a memory block. The least recently referenced memory block is put in the first position. If the memory block has not been in the cache already, the 'oldest' memory block is removed from the cache.
- *A-way set associative cache with LRU replacement strategy*:

$$\mathcal{U}_{\mathcal{M}_{A-way}}(c, s) = c'$$

$$c' = \begin{cases} c[\, l_j \mapsto s, \\ \quad l_i \mapsto c(l_{i-1}) \mid i = j+1 \ldots h, \\ \quad l_i \mapsto c(l_i) \mid i = h+1 \ldots (j+A-1) \quad]; \\ \quad \text{where } \{(l_j, s), \ldots, (l_{j+A-1}, s)\} \\ \qquad \subset \mathcal{M}_{A-way} & \text{if } \exists l_h : c(l_h) = s \\ c[\, l_j \mapsto s, \\ \quad l_i \mapsto c(l_{i-1}) \mid i = j+1 \ldots (j+A-1) \,]; \\ \quad \text{where } \{(l_j, s), \ldots, (l_{j+A-1}, s)\} \\ \qquad \subset \mathcal{M}_{A-way} & \text{otherwise} \end{cases}$$

  An A-way set associative cache is partitioned into n/A *fully associative sets*. The *fully associative set* $\{l_j, \ldots, l_{j+A-1}\}$ is treated as the fully associative cache above. For all cache lines that are not in in the *set*, the cache state remains unchanged.

We represent programs by control flow graphs consisting of nodes and typed edges. The nodes represent *basic blocks*[4]. For each basic block it is known which memory blocks it references[5], i.e. there exists a mapping from control flow nodes to a list of memory blocks: $\mathcal{L}\ : V \rightarrow S^*$. The execution of a basic block successively loads all memory its blocks into the cache.

We can describe the working of a cache by the aid of the update functions $\mathcal{U}_{\mathcal{M}}$. It is applied for all memory references of a control flow node by walking in the control flow graph according to the execution of a program. The effect of a control flow node n, on a cache state c is[6]: $[\![n]\!]_{\mathcal{M}} c = \mathcal{U}_{\mathcal{M}}\ (\ldots(\mathcal{U}_{\mathcal{M}}\ (c, s_1))\ldots)s_x$ where $\mathcal{L}\ (n) = [s_1, \ldots, s_x]$.

The cache state at a computation point t_m is the composition of functions related to the elements of the trace $(t_1, \ldots, t_m)$ applied to the initial cache state $\bot_{\mathcal{M}}$ that maps all cache lines to I: $[\![(t_1, \ldots, t_m)]\!]'_{\mathcal{M}} \bot_{\mathcal{M}}$ where $[\![(t_1, \ldots, t_m)]\!]'_{\mathcal{M}} = ([\![(t_1, \ldots, t_{m-1})]\!]'_{\mathcal{M}} \circ [\![t_m]\!]_{\mathcal{M}})$ and $[\![\emptyset]\!]'_{\mathcal{M}} = \mathbf{id}$.

7 Abstract Semantics

In order to generate a program analyzer, the program analyzer generator `PAG` requires the specification of an abstract domain, abstract semantic functions, and a join function. The domain for our abstract interpretation is given by the *abstract cache states*:

Definition 4 abstract cache state. An *abstract cache state* $\hat{c} : L \rightarrow 2^{S'}$ is a mapping from the cache lines into the powerset of the memory blocks. $\hat{C}$ denotes the set of all abstract cache states.

The abstract semantic function describes the effects of a control flow node on an element of the abstract domain. The **abstract cache update** function $\hat{\mathcal{U}}_{\mathcal{M}}$ for abstract cache states is a canonical extension of the cache update function $\mathcal{U}_{\mathcal{M}}$ on concrete cache states:

- *direct mapped cache*: $\hat{\mathcal{U}}_{\mathcal{M}_{direct}}(\hat{c}, s_x) = \hat{c}[l_i \mapsto \{s_x\}]$ where $i = (x\%n)$
- *fully associative cache with LRU replacement strategy*: $\hat{\mathcal{U}}_{\mathcal{M}_{assoc}}(\hat{c}, s) = \hat{c}'$

$$\hat{c}' = \begin{cases} [\ l_0 \mapsto \{s\}, & \\ \ \ l_i \mapsto \hat{c}(l_{i-1}) - \{s\} \mid i = 1 \ldots h, & \\ \ \ l_i \mapsto \hat{c}(l_i) - \{s\} \mid i = h+1 \ldots n-1]; & \text{if } \exists l_h : \hat{c}(l_h) = \{s\} \\ [\ l_0 \mapsto \{s\}, l_i \mapsto \hat{c}(l_{i-1}) - \{s\} \text{ for } i = 1 \ldots n-1]; & \text{otherwise} \end{cases}$$

[4] A basic block is a sequence (of fragments) of instructions in which control flow enters at the beginning and leaves at the end without halt or possibility of branching except at the end. For our cache analysis, it is most convenient to have one memory reference per control flow node. Therefore, our nodes may represent the different fragments of machine instructions that access memory.

[5] This is very restricted. See Section 10.1 for weaker restrictions.

[6] In the literature on abstract interpretation (e.g. [25]), our concrete semantics is usually referred to as auxiliary semantics, which is sometimes constructed for the purpose of defining an appropriate abstract semantics.

– *A-way set associative cache with LRU replacement strategy*:

$$\hat{\mathcal{U}}_{\mathcal{M}_{A-way}}(\hat{c}, s) = \hat{c}'$$

$$\hat{c}' = \begin{cases} \hat{c}[\ l_j \mapsto \{s\}, & \\ \quad l_i \mapsto \hat{c}(l_{i-1}) - \{s\} \mid i = j+1 \ldots h, & \\ \quad l_i \mapsto \hat{c}(l_i) - \{s\} \mid i = h+1 \ldots (j+A-1) \quad]; & \\ \quad \text{where } \{(l_j, s), \ldots, (l_{j+A-1}, s)\} & \\ \qquad \subset \mathcal{M}_{A-way} & \text{if } \exists l_h : \hat{c}(l_h) = \{s\} \\ \hat{c}[\ l_j \mapsto \{s\}, & \\ \quad l_i \mapsto \hat{c}(l_{i-1}) - \{s\} \mid i = j+1 \ldots (j+A-1)\ ]; & \\ \quad \text{where } \{(l_j, s), \ldots, (l_{j+A-1}, s)\} & \\ \qquad \subset \mathcal{M}_{A-way} & \text{otherwise} \end{cases}$$

On control flow nodes with at least two[7] predecessors, *join*-functions are used to combine the abstract cache states.

Definition 5 join function. A *join function* $\hat{\mathcal{J}}_{\mathcal{M}} : \hat{C} \times \hat{C} \mapsto \hat{C}$ is a binary function on abstract cache states.

7.1 Join Functions for Direct Mapped Caches

For the direct mapped cache, Mueller at al. [16, 3, 15] use the following join functions: $\hat{\mathcal{J}}_{\mathcal{M}_{direct}}(\hat{c}_1, \hat{c}_2) = \hat{c}$ where $\hat{c}(l) = \hat{c}_1(l) \cup \hat{c}_2(l)$.

$\hat{c}$ computes for each cache line l a set of possible contents. If a cache line l on two different paths with cache states $\hat{c}_1$ and $\hat{c}_2$ holds different memory blocks $\hat{c}_1(l) = \{s_x\}$, $\hat{c}_2(l) = \{s_y\}$, and $x \neq y$, the set $\hat{c}(l) = \{s_x, s_y\}$ means that the cache line l holds either memory block s_x or s_y.

The goal is to determine for every control flow node n whether the references to the memory $\mathcal{L}\ (n)$ will result in cache hits or cache misses.

This can be computed from the abstract semantics by:

– if a memory block s is not in $\hat{c}(l)$ for an arbitrary l then it is definitely not in any cache line.
  This memory reference will *always miss* the cache.
– if $\hat{c}(l) = \{s\}$ for a cache line l then s is definitely in cache line l.
  This memory reference will *always hit* the cache.
– if $\hat{c}(l) = \{I, s\}$ for a cache line l then s is definitely in cache line l for the second and all following executions of n.

In [3] references to instruction caches are further categorized taking the loop nesting level of the instruction into account. An instruction within a loop is called *first miss* if the first reference to the instruction is a cache miss and all remaining references during the execution of the loop are cache hits. Likewise, a *first hit* indicates that the first reference to the instruction will be a hit and

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

all remaining references during the execution of the loop will be misses (see Table 1). This categorization of instructions is used in a timing tool to compute the WCET of a program.

Other program lines in the loop that map to the same cache line	The instruction is always executed in the loop and is in cache initially	In the worst case treat the instruction as:
no	no	first miss
no	yes	always hit
yes	no	always miss
yes	yes	first hit

Table 1. Categorizations of Instructions for the WCET analysis according to [3].

For fully associative caches and set associative caches, two different join functions have to be used. For the identification of 'always hits', the join function corresponds to set intersection, and for the identification of 'always miss', the join function corresponds to set union.

During the analysis for direct mapped caches there never occur empty sets. The interpretation of sets of one element is equivalent under union and intersection: $\#(A \cup B) = 1$ and $A \neq \emptyset$ and $B \neq \emptyset \Rightarrow (A \cup B) = (A \cap B)$.

7.2 Join Functions for Fully Associative Caches with LRU Replacement

For the fully associative cache with LRU replacement strategy we can use the following join function to determine if a memory block s is in the cache at a control flow node n: $\hat{\mathcal{J}}^{\cap}_{\mathcal{M}_{assoc}}(\hat{c}_1, \hat{c}_2) = \hat{c}$ where

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

The position of the memory blocks in the abstract cache state, i.e. the number of the cache line, represents the relative age of a memory block. If a memory block s has two different relative ages in two abstract cache states, i.e. is in different positions $s \in \hat{c}_1(l_x)$ and $s \in \hat{c}_2(l_y)$ then the join function takes the oldest relative age, i.e. the highest position.

Example 2 $\hat{\mathcal{J}}^{\cap}_{\mathcal{M}_{assoc}}$.

	l_0	l_1	l_2	l_3
$\hat{c}_1$	$\{s_a\}$	$\{s_b\}$	$\{s_c\}$	$\{s_d\}$
$\hat{c}_2$	$\{s_c\}$	$\{s_e\}$	$\{s_a\}$	$\{s_d\}$
$\hat{\mathcal{J}}^{\cap}_{\mathcal{M}_{assoc}}(\hat{c}_1, \hat{c}_2)$	$\{\}$	$\{\}$	$\{s_c, s_a\}$	$\{s_d\}$

An abstract cache state $\hat{c}$ at a control flow node n can be interpreted in the following way:

– If $s \in \hat{c}(l)$ for a cache line l then s is definitely in the cache. A reference to s will *always hit* the cache.
– If $s \in \hat{c}(l_x)$ then s will remain in the cache for at least $(\frac{capacity}{line\ size} - x)$ cache updates that put a 'new' element into the cache.

To determine if a memory block s is never in the cache at a control flow node n we use the join functions: $\hat{\mathcal{J}}^{\cup}_{\mathcal{M}_{assoc}}(\hat{c}_1, \hat{c}_2) = \hat{c}$ where $\hat{c}(l) = \hat{c}_1(l) \cup \hat{c}_2(l)$ Here we have the same join function as for the direct mapped cache.

An abstract cache state $\hat{c}$ at a control flow node n can be interpreted in the following way:

– if a memory block s is not in $\hat{c}(l)$ for an arbitrary l then it is definitely not in any cache line. This memory reference will *always miss* the cache.
– If $s \in \hat{c}(l_x)$ with x minimal then s will remain in the cache for at most $(\frac{capacity}{line\ size} - x)$ cache updates that put a 'new' element into the cache.

7.3 Join Functions for A-way Set Associative Caches with LRU Replacement

For the A-way set associative cache with LRU replacement strategy we can use the following join function to determine if a memory block s is in the cache at a control flow node n: $\hat{\mathcal{J}}^{\cap}_{\mathcal{M}_{A-way}}(\hat{c}_1, \hat{c}_2) = \hat{c}$ where

$$\hat{c}(l_x) = \{s_i \mid \exists l_a, l_b \text{ with } s_i \in \hat{c}_1(l_a), s_i \in \hat{c}_2(l_b) \text{ and } x = max(a, b)$$
$$\text{and } (l_a, s_i), (l_b, s_i), (l_x, s_i) \in \mathcal{M}_{A-way}\}.$$

An A-way set associative cache is partitioned into n/A fully associative sets. $\hat{\mathcal{J}}^{\cap}_{\mathcal{M}_{A-way}}(\hat{c}_1, \hat{c}_2)$ corresponds to the application of $\hat{\mathcal{J}}^{\cap}_{\mathcal{M}_{assoc}}$ to the fully associative sets of $\hat{c}_1$ and $\hat{c}_2$.

Example 3 $\hat{\mathcal{J}}^{\cap}_{\mathcal{M}_{A-way}}$. Let $\{l_0, l_1\}$ and $\{l_2, l_3\}$ be the fully associative sets of a two-way set associative cache with 4 lines.

	l_0	l_1	l_2	l_3
$\hat{c}_1$	$\{s_a\}$	$\{s_b\}$	$\{\}$	$\{s_e, s_d\}$
$\hat{c}_2$	$\{s_c\}$	$\{s_a\}$	$\{s_d\}$	$\{s_f\}$
$\hat{\mathcal{J}}^{\cap}_{\mathcal{M}_{assoc}}(\hat{c}_1, \hat{c}_2)$	$\{\}$	$\{s_a\}$	$\{\}$	$\{s_d\}$

An abstract cache state $\hat{c}$ at a control flow node n can be interpreted in the following way:

– If $s \in \hat{c}(l)$ for a cache line l then s is definitely in the cache. A reference to s will *always hit* the cache.
– If $s \in \hat{c}(l_x)$ and $\{l_j, \ldots, l_c, \ldots, l_{j+A-1}\}$ is the fully associative set of the cache with $j \leq x \leq j + A + 1$, then s will remain in the cache for at least $(j + A - 1) - x$ cache updates that put a 'new' element into the cache.

To determine if a memory block s is never in the cache at a control flow node n we use the same join functions and the same interpretation as in the fully associative case: $\hat{\mathcal{J}}^{\cup}_{\mathcal{M}_{A-way}} = \hat{\mathcal{J}}^{\cup}_{\mathcal{M}_{assoc}}$.

8 Analysis of Loops

Loops are of special interest, since many programs spend most of their runtime within loops. In a control flow graph, a loop is represented as a cycle. The *start node* of a loop has two incoming edges. One represents the entry into the loop, the other represents the control flow from the end of the loop to the beginning of the loop. The later is called *loop edge*[8].

A loop usually iterates more than once. Since the execution of the loop body usually changes the cache contents, it is useful to distinguish the first iteration from all others. This could be achieved by virtually unrolling each loop once.

Example 4. Let us consider a sufficiently large fully associative data cache with LRU replacement strategy and the following program fragment:

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

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

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

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

For nested loops, loop unrolling can be an expensive transformation which is exponential in the nesting depth. This problem is similar to the problem of analyzing procedures in program analysis, for which solutions exist (see Section 3).

For our analysis of cache behavior we transform loops into procedures to be able to use the existing methods and tools[9] (see Figure 1).

8.1 Callstring Approach

There are only a finite number of cache lines and for each program a finite number of memory blocks. This means, the domain of abstract cache states

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

[9] Ludwell Harrison III [10] also proposed this transformation for the analysis of loops.

```
                                proc loop_L();
⋮                               if P then
while P do                        BODY
      BODY          ⟹            loop_L();     (2)
end;                            end
⋮                               ⋮
                                loop_L();       (1)
                                ⋮
```

Fig. 1. Loop transformation.

$\hat{c} : L \longrightarrow 2^{S'}$ is finite. Additionally, the abstract cache update functions $\hat{\mathcal{U}}$ and the join functions $\hat{\mathcal{J}}$ are monotonic. This guarantees that abstract interpretations with both the callstring approach and the functional approach will terminate.

In the callstring approach, the high complexity of the functional approach can be circumvented. If we restrict the callstring length to 1 (*callstring*(1)), then for each transformed loop only two different incoming abstract cache states are considered: One for the call to the loop–procedure at the original place of the loop in the program (1) (see Figure 1); and one for the recursive call of the loop-procedure (2). The first call corresponds to the first iteration of the loop. The second call corresponds to all other iterations of the loop.

This means, we can interpret the abstract cache states $\hat{c}_f$ for the first iteration and $\hat{c}_o$ for all other iterations at a control flow node n within the loop–procedure according to Table 2. Note: For A-way set associative caches and fully associative caches the determination of 'always hit' and 'always miss' requires analysis with both $\hat{\mathcal{J}}^{\cap}_{\mathcal{M}}$ and $\hat{\mathcal{J}}^{\cup}_{\mathcal{M}}$. We call the analysis with $\hat{\mathcal{J}}^{\cap}_{\mathcal{M}}$ **must analysis** because it computes all blocks that must be in the cache. And we call the analysis with $\hat{\mathcal{J}}^{\cup}_{\mathcal{M}}$ **may analysis** because it computes all blocks that may be in the cache.

8.2 Functional Approach

During the analysis of a program, `PAG` tabulates for each procedure (and each loop that has been transformed into a procedure) all abstract cache states within the procedure for all different incoming abstract cache states.

This computes the same values as if the loops had been unrolled. In the worst case, the exponential growth in program code of the loop unrolling corresponds to exponentially many different incoming abstract cache states that are tabulated during the analysis. But often, there are much less different incoming abstract cache states than unrolled loop bodies for a deeply nested loop nest.

The functional approach gives the most detailed results for the abstract interpretation but may be very expensive.

Interpretation of the abstract cache state $\hat{c}_f$ for a reference to a memory block s:	Interpretation of the abstract cache state $\hat{c}_o$ for a reference to a memory block s:	Combination of $\hat{c}_f(s)$ and $\hat{c}_o(s)$:
always hit	always hit	always hit
always miss	always hit	first miss
always miss	always miss	always miss
always hit	always miss	first hit
always hit	–	first hit
always miss	–	always miss
–	always hit	first miss
–	always miss	always miss
–	–	always miss

Table 2. Interpretation of abstract cache states for callstring(1). The second part describes the categorization for a WCET analysis according to Table 1 if no classification into 'always hit' and 'always miss' is possible.

9 Example

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

`while` $..e..$ `do` $..b..; ..c..; ..a..; ..d..; ..c..$ `end`

The control flow graph and the result of the analysis with callstring(1) are shown in Figure 2. We assume that each variable fits exactly into one cache line. The nodes of the control flow graph are numbered 1 to 6, and each node is marked with the variable it accesses (a, b, c, d, e). For the analysis, we assume the loop has been implicitly transformed into a procedure according to Figure 1.

Each node is marked with the abstract cache states (in the same format as in Example 2) computed by the `PAG`–generated analyzer immediately before the abstract cache states are updated with the memory references. The loop entry edge is marked with the incoming abstract cache states. The loop exit edge is marked with the outgoing abstract cache states.

10 Data Caches and Combined Data/Instruction Caches

10.1 Scalar Variables

In the current design, the work is limited to the prediction of memory references to addresses that can be determined at analysis time. This allows for example for the prediction of instruction cache behavior.

Our analysis can also be used to predict the behavior of data caches or combined instruction/data caches for programs that use only scalar variables.

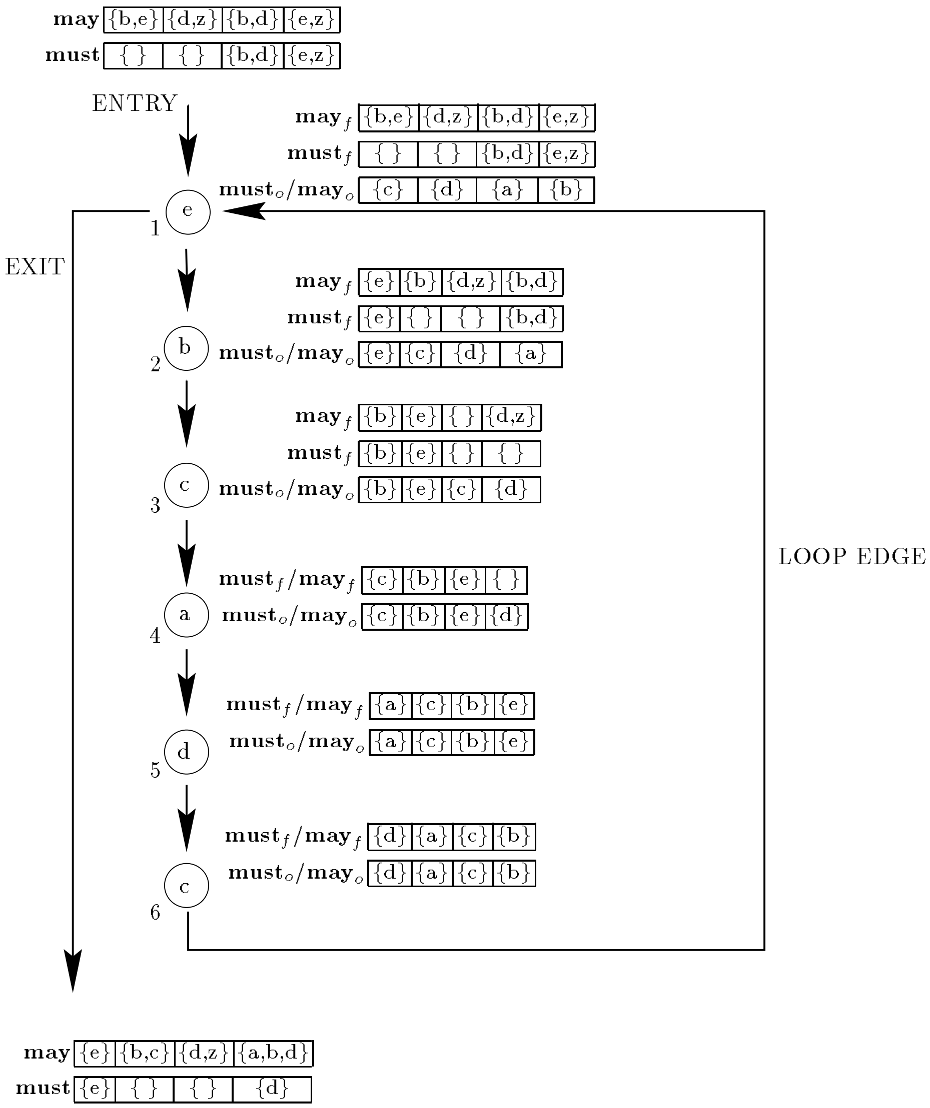

Fig. 2. Must and may analysis for a fully associative data cache with callstring(1). **must** and **may** are the abstract cache states for the must and the may analysis. $\mathbf{must}_f$ and $\mathbf{may}_f$ are the abstract cache states for the first loop iteration. $\mathbf{must}_o$ and $\mathbf{may}_o$ are the abstract cache states for all other iterations. The abstract cache states can be interpreted for each variable reference according to Table 2:

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

For this kind of programs, it is possible to compute for each data reference to a procedure parameter or a local variable the address within the procedure stack frame by a static stack level simulation [25]. For each call to a procedure, the address of the procedure stack frame depends only on a statically computable offset to the procedure stack frame of the caller.

For our abstract interpretation, we extend the function that maps control flow nodes to the list of referenced memory blocks by an argument that is the set of possible absolute stack frame addresses[10]: $\mathcal{L}' : V \times 2^{\mathbb{N}_0} \rightarrow S^*$.

Additionally, we assume a function $\mathcal{H}$ that maps call nodes to their relative stack frame offset or stack height: $\mathcal{H} : V \rightarrow \mathbb{N}_0$.

All abstract semantic functions and join functions have to be defined on pairs of abstract cache states and sets of actual stack frame addresses:

$$\hat{\mathcal{U}}'_{\mathcal{M}} : \hat{C} \times 2^{\mathbb{N}_0} \times S \rightarrow \hat{C} \times 2^{\mathbb{N}_0} \quad \text{and} \quad \hat{\mathcal{J}}'_{\mathcal{M}} : (\hat{C} \times 2^{\mathbb{N}_0}) \times (\hat{C} \times 2^{\mathbb{N}_0}) \rightarrow (\hat{C} \times 2^{\mathbb{N}_0})$$

Only the abstract semantic function for procedure calls[11] have to change the actual stack frame address.

$$\hat{\mathcal{U}}'_{\mathcal{M}}(\hat{c}, s, \{h_1, \ldots, h_x\}) = \begin{cases} \Big(\hat{\mathcal{U}}_{\mathcal{M}}(\hat{c}, s), \{h_1 + \mathcal{H}(n), \ldots, h_x + \mathcal{H}(n)\}\Big) & \\ \qquad\qquad\qquad\qquad \text{for a call node } n & \\ \Big(\hat{\mathcal{U}}_{\mathcal{M}}(\hat{c}, s), \{h_1, \ldots, h_x\}\Big) \qquad \text{otherwise} & \end{cases}$$

$$\hat{\mathcal{J}}'_{\mathcal{M}}((\hat{c}_1, H_1), (\hat{c}_2, H_2)) = \Big(\hat{\mathcal{J}}_{\mathcal{M}}(\hat{c}_1, \hat{c}_2), H_1 \cup H_2\Big)$$

For programs without recursive procedures, there are only finitely many stack frame addresses. This guarantees termination of the abstract interpretation. With the functional approach and the callstring approach where the procedure nesting depth of the program does not exceed the callstring length, the sets of stack frame addresses for the $\hat{\mathcal{U}}'$ and $\hat{\mathcal{J}}'$ functions contain always exactly one element. This means there is no loss of information.

For programs with recursive procedures, the number of stack frame addresses may grow infinitely during the analysis so that the analysis does not terminate. Cousot and Cousot [5] proposed a technique called *'widening'* that speeds up the analysis.

We use a 'widening' function $\bigtriangledown$ to restrict the number of stack frame addresses. When during the analysis the number of elements in a set of stack frame addresses exceeds a given limit R, $\bigtriangledown$ replaces this set by $\mathbb{N}_0$[12]. This can only occur when the join function is applied.

[10] This works for C-type languages where all procedures are 'global'. PASCAL-like languages with local procedures referencing local variables of other procedures can't be modeled in this way.

[11] This holds only for procedures of the original program. The newly introduced loop–procedures do not change the procedure stack frame address.

[12] `PAG` includes a *'negative'* set representation, so that this operation is efficiently implemented.

$$\nabla(\{h_1, \ldots, h_x\}) = \begin{cases} \{h_1, \ldots, h_x\} \text{ if } x \leq R \\ \\ \mathbb{N}_0 \qquad \text{otherwise} \end{cases}$$

An occurrence of $\mathbb{N}_0$ means a total loss of information on the stack frame address. Accordingly, the update and join functions can not compute any relevant information, but map all abstract cache states to the most undefined cache state $\top_{\mathcal{M}}$: $\hat{\mathcal{U}'}_{\mathcal{M}}(\hat{c}, s, \mathbb{N}_0) = \hat{\mathcal{J}'}_{\mathcal{M}}((\hat{c}_1, \mathbb{N}_0), (\hat{c}_2, H)) = \hat{\mathcal{J}'}_{\mathcal{M}}((\hat{c}_1, H), (\hat{c}_2, \mathbb{N}_0)) = (\top_{\mathcal{M}}, \mathbb{N}_0)$.

$$\top_{\mathcal{M}_{direct}} = \top^{\cup}_{\mathcal{M}_{assoc}} = \top^{\cup}_{\mathcal{M}_{A-way}} = [l_i \mapsto S \mid i = 1 \ldots n]$$

$$\top^{\cap}_{\mathcal{M}_{assoc}} = \top^{\cap}_{\mathcal{M}_{A-way}} = [l_i \mapsto \{\} \mid i = 1 \ldots n]$$

10.2 Writes

So far, we have ignored writing to a cache and only considered reading from a cache. There are two common cache organizations with respect to writing to the cache [11]:

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

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

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

Writes to write through/write allocate caches can be treated as reads. For no write allocate caches, the update functions have to be adapted. For A-way set associative caches ($A > 1$) and fully associative caches, a write access to a block s is treated as a read access, if s is already in the concrete or abstract cache state. Otherwise, and for direct mapped caches[13], the write access is ignored, i.e. the update functions is the identity function for this case.

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

[13] This is to preserve the interpretation of sets of one element as always hits.

an unmodified line is replaced[14]. To keep track of modified cache lines, we extend the cache states by a 'dirty' bit, where d means modified, p means unmodified[15]: $c : L \longrightarrow \{d, p\} \times S'$ and $\hat{c} : L \longrightarrow 2^{\{d,p\} \times S'}$.
The update functions distinguish **r**eads and **w**rites. The dirty bit is set to d only on writes:

$$\begin{aligned} \mathcal{U}_{\mathcal{M}} &: C_c \times (\{r, w\} \times S) \rightarrow C_c \\ \hat{\mathcal{U}}_{\mathcal{M}} &: \hat{C} \times (\{r, w\} \times S) \rightarrow \hat{C} \\ \hat{\mathcal{U}'}_{\mathcal{M}} &: \hat{C} \times 2^{\mathbb{N}_0} \times (\{r, w\} \times S) \rightarrow \hat{C} \times 2^{\mathbb{N}_0} \end{aligned}$$

Let n be a control flow node, s_a be one read or write memory reference at n, $\hat{c}_1^{\cup}$ the abstract cache state for the *may* analysis immediately before s_a is referenced, and $\hat{c}_2^{\cup} = \hat{\mathcal{U}}_{\mathcal{M}}(\hat{c}_1^{\cup}, m, s_a), m \in \{r, w\}$ the abstract cache state immediately after s_a was referenced, $\hat{c}_1^{\cap}$ the abstract cache state for the *must* analysis immediately before s_a is referenced, and $\hat{c}_2^{\cap} = \hat{\mathcal{U}}_{\mathcal{M}}(\hat{c}_1^{\cap}, m, s_a), m \in \{r, w\}$ the abstract cache state immediately after s_a was referenced.

$$\begin{tikzcd} & \text{must} & \text{may} \\ & \hat{c}_1^{\cap} \arrow[dd, "\hat{\mathcal{U}}_{\mathcal{M}}"] & \hat{c}_1^{\cup} \arrow[dd, "\hat{\mathcal{U}}_{\mathcal{M}}"] \\ n : s_a & & \\ & \hat{c}_2^{\cap} & \hat{c}_2^{\cup} \end{tikzcd}$$

Let l_x the cache line where s_a has been stored in $\hat{c}_2^{\cup}$. Then $\hat{c}_1^{\cup}(l_x)$ contains all possible memory blocks that may have been replaced by s_a.

- If $\{s \mid (d, s) \in \hat{c}_1^{\cup}(l_x)\} = \emptyset$, then no dirty memory block has been replaced. This reference has definitively caused **no write back**.
- If there is a dirty line $s \in \{s \mid (d, s) \in \hat{c}_1^{\cup}(l_x)\}$ and s is an always hit in $\hat{c}_1^{\cap}$ and s is a always miss in $\hat{c}_2^{\cup}$, then a dirty memory block has been replaced. This reference has definitively caused a **write back**.
- If $\{s \mid (d, s) \in \hat{c}_1^{\cup}(l_x)\} \neq \emptyset$ then for a WCET analysis we have to consider a possible write back.

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

11 State of the Implementation and Future Work

The presented techniques have been validated with an ANSI-C frontend that has been interfaced to `PAG`. We are currently developing a `PAG` interface for executables based on the Wisconsin architectural research tool set (WARTS).

---

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

[15] For the abstract interpretation: $d > p$ and $\top_{\{d,p\}} = d$.

12 Conclusion

We have described several semantics of programs executed on machines with several types of one level caches. Abstract interpretations based on these semantics statically analyze the intrinsic cache behavior of programs. The information computed allows interpretations such as 'always hit', 'always miss', 'first hit', 'first miss', and 'write back'. It can be used to improve execution time calculations for programs. The analyses are specified as needed by the program analyzer generator `PAG`.

Acknowledgements

We like to thank Susan Horwitz for making available the ANSI-C frontend, and Mark D. Hill, James R. Larus, Alvin R. Lebeck, Madhusudhan Talluri, and David A. Wood for making available the Wisconsin architectural research tool set (WARTS).

References

1. Martin Alt and Florian Martin. Generation of efficient interprocedural analyzers with PAG. In *SAS'95, Static Analysis Symposium*, pages 33–50. Springer-Verlag LNCS 983, September 1995.
2. Martin Alt, Florian Martin, and Reinhard Wilhelm. Generating dataflow analyzers with PAG. Technical Report A10-95, Universität des Saarlandes, 1995.
3. Robert Arnold, Frank Mueller, David B. Whalley, and Marion Harmon. Bounding worst-case instruction cache performance. In *IEEE Symposium on Real-Time Systems*, pages 172–181, Dec 1994.
4. Swagato Basumallick and Kelvin Nilsen. Cache issues in real-time systems. In *Proceedings of the 1994 ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, June 1994.
5. P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the second International Symposium on Programming*, pages 106–130, Dunod, Paris, France, 1976.
6. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, January 1977.
7. P. Cousot and R. Cousot. Static determination of dynamic properties of generalized type unions. In *Proceedings of an ACM Conference on Language Design for Reliable Software*, volume 12(3), pages 77–94, Raleigh, NC, March 1977.
8. P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. *Formal Description of Programming Concepts*, pages 237–277, 1978.
9. Wolfgang A. Halang and Krzysztof M. Sacha. *Real-Time Systems*. World Scientific, 1992.
10. Ludwell Harrison. *Personal communication on Abstract Interpretation, Dagstuhl Seminar*, 1995.

11. J.L Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1990.
12. Sung-Soo Lim, Young Hyun Bae, Gye Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon, and Chong Sang Kim. An accurate worst case timing analysis for risc processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, July 1995.
13. Scott McFarling. Program optimization for instruction caches. In *Architectural Support for Programming Languages and Operating Systems*, pages 183–191, Boston, Massachusetts, April 1989. Association for Computing Machinery ACM.
14. Abraham Mendlson, Shlomit S. Pinter, and Ruth Shtokhamer. Compile time instruction cache optimizations. *Computer Architecture News*, 22(1):44–51, March 1994.
15. Frank Mueller. Static cache simulation and its applications. Phd thesis, Florida State University, July 1994.
16. Frank Mueller, David B. Whalley, and Marion Harmon. Predicting instruction cache behavior. In *Proceedings of the 1994 ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, June 1994.
17. Kelvin D. Nilsen and Bernt Rygg. Worst-case execution time analysis on modern processors. In *Proceedings of the 1995 ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, June 1995.
18. Chang Yun Park and Alan C. Shaw. Experiments with a program timing tool based on source-level timing schema. *IEEE Computer*, 25(5):48–57, May 1991.
19. Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 16–27, White Plains, New York, June 1990.
20. P. Puschner and Ch. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1:159–176, 1989.
21. J. Rawat. Static analysis of cache performance for real-time programming. Masters thesis, Iowa State University, May 1993.
22. Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–233. Prentice-Hall, 1981.
23. A.J. Smith. Cache memories. *ACM Computing surveys*, 14(3):473–530, Sep 1983.
24. Alexander D. Stoyenko, V. Carl Hamacher, and Richard C. Holt. Analyzing hard-real-time programs for guaranteed schedulability. *IEEE Transactions on Software Engineering*, 17(8), August 1991.
25. Reinhard Wilhelm and Dieter Maurer. *Compiler Design.* International Computer Science Series. Addison–Wesley, 1995.
26. Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. *SIGPLAN Notices*, 26(6):30–44, June 1991. Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.

Specification

For the sake of simplicity and space, we assume only references to fixed addresses, and we consider only direct mapped caches and the must analysis for fully associative caches:

```
DOMAIN   store_lines = set(int)
         ACACHE      = int -> store_lines
         // Set of all abstract cache states

GLOBAL  cache_size      : int
        store_line_size : int
        maxsym          : int // number of memory locations
        cache_mode      : int
        // cache_mode = 1 is direct mapped
        // cache_mode = 2 is fully associative

NODE  uses            : *Symbol // The externally defined
// function 'uses' returns at every control flow node the
// list of memory locations which are referenced

PROBLEM cache
direction : forward
carrier   : ACACHE      // the used domain
init      : [ -> {-1}]  // initialize all nodes with the abstract
                        // cache where all cache lines are empty
combine   : join

TRANSFER   default : list_update(uses,@);
// for each statement update the cache with all memory references

SUPPORT
list_update([]    ,acache) = acache;
list_update(obj:xs,acache) = list_update(xs,update(obj,acache));

update :: int,ACACHE -> ACACHE;
update(loc,acache) =
         letrec sline = loc / store_line_size;
                pos = is_in_cache(loc,acache); in
           if pos = -1  // loc is not in cache
                then update_cache_out(acache,pos,sline)
                else update_cache_in(acache,pos,sline)
             endif;

join(a,b) = if cache_mode = 1  then a lub b  // standard set union
            else merge(a,b,maxsym/store_line_size,[ ->{}])
                           // for fully associative with LRU
            endif;

find_cache_line::ACACHE,int,int -> int;
find_cache_line(_,_,-1)    = -1;
```

```
find_cache_line(acache,sline,n) = if sline ? acache(n) then n
                            else find_cache_line(acache,sline,n-1)
                             endif;

merge::ACACHE,ACACHE,int,ACACHE -> ACACHE;
merge(_,_,-1,acache)   = acache;
merge(a,b,line,acache) = letrec
                      s1 = find_cache_line(a,line,cache_size);
                      s2 = find_cache_line(b,line,cache_size);
                      zz = max(s1,s2); in
               if s1>=0 && s2>=0 then
                   merge(a,b,line-1,acache\[zz->acache(zz) ~ line]
               else merge(a,b,line-1,acache)
               endif;

is_in_cache_associative::int,ACACHE,int -> int;
is_in_cache_associative(_,_,-1)       = -1;
is_in_cache_associative(sline,acache,pos) =
   if sline ? acache(pos) then pos
     else is_in_cache_associative(sline,acache,pos-1)
   endif;

is_in_cache_direct::int,ACACHE -> int;
is_in_cache_direct(sline,acache) = let pos = sline % cache_size;
                                in if sline ? acache(pos) then pos
                                   else -1 endif;

is_in_cache::int,ACACHE -> int;
is_in_cache(sline,acache) =
  if cache_mode = 1 then is_in_cache_direct(sline,acache)
  else is_in_cache_associative(sline,acache,cache_size)
  endif;

// update function for cache states  -----------------------------

shift::ACACHE,int -> ACACHE;
shift(acache,-1)  = acache;
shift(acache,pos) = shift(acache\[pos->acache(pos-1)],pos-1);


update_cache_out_direct(acache,pos,sline) =
   update_cache_in_direct(acache,pos,sline);

update_cache_in_direct(acache,pos,sline)  =
   acache\[pos->{sline}];
```

```
update_cache_in_associative(acache,pos,sline) =
  if acache(pos) = {sline} then shift(acache,pos)\[0->{sline}]
  else update_cache_out_associative
                       (acache\[pos->acache(pos)~sline],pos,sline)
  endif

update_cache_out_associative(acache,pos,sline) =
  let acache = shift(acache,cache_size-1);
  in acache\[0->{sline}];

update_cache_in(acache,pos,sline) =
   if cache_mode = 1 then update_cache_in_direct(acache,pos,sline)
                     else update_cache_in_associative
                                                (acache,pos,sline)
   endif;

update_cache_out(acache,pos,sline) =
  if cache_mode = 1 then update_cache_out_direct(acache,pos,sline)
  else update_cache_out_associative(acache,pos,sline)
  endif;

max(x,y) = if x < y then y else x endif;
```