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
{alt,ferdi,florian,wilhelm}@cs.uni-sb.de
http://www.cs.uni-sb.de/users/{alt,ferdi,martin,wilhelm}

Extended Abstract

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.

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.

Programs with hard real time constraints have to be subjected to a schedulability analysis by the compiler [SHH91, HS92]; 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 case assumption is that all accesses miss the cache. This is an overly pessimistic assumption which leads to a waste of hardware resources.

2 An Introduction to Program Analysis

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 [CC77a, CC77b, CC78]. 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*.

Usually the meaning of a language is given by functions for the statements of the language computing on a concrete domain. A domain is a complete partially ordered set of values. For such a semantics, an abstract version consists of a new (simpler) abstract domain and abstract functions which give the abstract meaning for every program statement.

For an abstract semantics and an input program, a system of recursive equations can be constructed. The variables in this system are the values of the abstract domain for every program point. In this equation system, a value at a program point depends on the values at all program points which can directly precede the execution of this program point. For example, the value after the exit of a loop depends on the value at the end of the body of the loop and on the value before the loop because it is possible that the loop is never executed.

A *control flow graph* can be constructed which describes every possible flow of control in the input program and therefore also all dependencies between the variables of the equation system.

The theory of abstract interpretation states that the recursive equation system can be solved if the abstract domain has only finite ascending chains, i.e. every chain of values $v_1 < v_2 < \cdots$ has finite length, and every semantic function is monotonic.

The program analyzer generator `PAG` [AM95, AMW95] offers the possibility to generate a program analyzer from a description of the abstract domain and of the abstract semantic functions. Additionally one has to specify 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.

In the presence of (recursive) procedures, the program analysis faces some additional problems [SP81]. In the best possible solution, one considers abstract versions of procedures which transform input values into output values of the abstract domain. So the analyzer has to tabulate input/output pairs of abstract values[1]. To guarantee termination of the analysis, the abstract domain has to be finite. This approach is called the *functional approach*.

The *call string approach* tries to cut down the complexity of the problem by statically limiting the number of input/output pairs. To do this the call graph is considered, where every procedure is represented by a node and edges correspond to calls. The goal is not to

[1] if one doesn't assume the existence of a compact representation for abstract functions

merge information that is obtained on different paths through the graph. But in presence of recursion, the graph is cyclic and has therefore an infinite number of paths. So only the information obtained on paths of a fixed length K is kept separated. All other information is merged with the join function. If K is 1, there are as many input/output pairs for every procedure as there are different calls to that procedure in the program text. This results in possible loss of information, but is still correct.
Both approaches are fully integrated in `PAG`.

3 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 $\frac{\mathit{capacity}}{\mathit{line\ size}}$ blocks.
- *associativity* is the number of cache locations where a particular block may reside. $\frac{\mathit{capacity}}{\mathit{line\ size} * \mathit{associativity}}$ 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* [Smi83]. 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.

4 Concrete Semantics

In the following, we consider a cache as a set of cache lines L:

$$L = \{l_0, \dots, l_{n-1}\}, \text{ where } n = \frac{\mathit{capacity}}{\mathit{line\ size}}$$

The store S is divided into blocks of size *line size*, so that one memory block can be transferred into one cache line.

$$S = \{s_0, \dots, s_{m-1}\}$$

The locations where a memory block may reside in the cache depend on the level of associativity of the cache memory.
For the absence of any memory block in a cache line, we introduce a new element I; $S' = S \cup \{I\}$.

[2] A *set* can be considered as a fully associative subcache.

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

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 2 (cache update)
A cache update function $\mathcal{U}$ is a function from a concrete cache state and a memory block to a concrete cache state:

$$\mathcal{U}_{\mathcal{M}} \ : C_c \times S \to C_c$$

Accesses to caches can be modeled in the following ways[3]:

- *direct mapped cache*:

  $$\mathcal{U}_{\mathcal{M}_{direct}}(c, s_x) = c[l_i \mapsto s_x] \text{ where } i = (x\%n);$$

  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*:

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

  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.

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

[3] For the sake of space, we consider only direct mapped caches and fully associative caches.

[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.

references[5].

We can describe the working of a cache by the aid of the update functions[6] $\mathcal{U}_\mathcal{M}$. 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 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 $\perp_\mathcal{M}$ that maps all cache lines to I.

5 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 3 (abstract cache state)
An *abstract cache state* is a mapping from the cache lines into the powerset of the memory blocks.

$$\hat{c} : L \to 2^{S'}$$

$\hat{C}$ 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*:

$$\hat{\mathcal{U}}_{\mathcal{M}_{direct}}(\hat{c}, s_x) = \hat{c}[l_i \mapsto \{s_x\}] \text{ where } i = (x\%n);$$

- *fully associative cache with LRU replacement strategy*:

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

---

[5] This is very restricted. Methods to overcome these restrictions will be presented elsewhere.

[6] In the literature on abstract interpretation (e.g. [WM95]), our concrete semantics is usually referred to as auxiliary semantics, which is sometimes constructed for the purpose of defining an appropriate abstract semantics.

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

Definition 4 (join function)
A *join function* is a binary function on abstract cache states.

$$\hat{\mathcal{J}}_{\mathcal{M}} : \hat{C} \times \hat{C} \mapsto \hat{C}$$

5.1 Join Functions for Direct Mapped Caches

For the direct mapped cache, Mueller at al. [MWH94, AMWH94, Mue94] use the following join functions:

$$\hat{\mathcal{J}}_{\mathcal{M}_{direct}}(\hat{c}_1, \hat{c}_2) = \hat{c}$$
$$\hat{c}(l) = \hat{c}_1(l) \cup \hat{c}_2(l)$$

$\hat{c}$ 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 [AMWH94] 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 all remaining references during the execution of the loop will be misses. This categorization of instructions is used in a timing tool to compute the WCET of a program.

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

5.2 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:

$$\hat{\mathcal{J}}^{\cap}_{\mathcal{M}_{assoc}}(\hat{c}_1, \hat{c}_2) = \hat{c}$$

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

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 1 ($\hat{\mathcal{J}}^{\cap}_{\mathcal{M}_{assoc}}$)

	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 following join functions:

$$\hat{\mathcal{J}}^{\cup}_{\mathcal{M}_{assoc}}(\hat{c}_1, \hat{c}_2) = \hat{c}$$

$$\hat{c}(l) = \hat{c}_1(l) \cup \hat{c}_2(l)$$

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.

6 Analysis of Loops

Loops are of special interest, since many programs spend most of their runtime within loops. 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.

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 2).

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

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

Figure 1: Loop transformation.

6.1 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 $\hat{c} : L \to 2^{S'}$ 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 1.

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

Table 1: Interpretation of abstract cache states for callstring(1).

6.2 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.

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

7 Conclusion

We have described the semantics of programs executed on machines with direct mapped and fully associative one level caches. The application of this method to set associative caches and to writes to caches has been ommitted for the sake of space. 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' for write back caches). It can be used to improve execution time calculations for programs. The analyses are specified as needed by the program analyzer generator `PAG`. Scalability is provided by the methods for interprocedural data flow analysis (functional, callstring(K)) that are integrated into `PAG`.

References

[AM95] 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.

[AMW95] Martin Alt, Florian Martin, and Reinhard Wilhelm. Generating dataflow analyzers with PAG. Technical Report A10-95, Universität des Saarlandes, 1995.

[AMWH94] 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.

[CC77a] 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.

[CC77b] 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.

[CC78] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. *Formal Description of Programming Concepts*, pages 237–277, 1978.

[HS92] Wolfgang A. Halang and Krzysztof M. Sacha. *Real-Time Systems.* World Scientific, 1992.

[Mue94] Frank Mueller. Static cache simulation and its applications. Phd thesis, Florida State University, July 1994.

[MWH94] 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.

[SHH91] 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.

[Smi83] A.J. Smith. Cache memories. *ACM Computing surveys*, 14(3):473–530, Sep 1983.

[SP81] 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.

[WM95] Reinhard Wilhelm and Dieter Maurer. *Compiler Design.* International Computer Science Series. Addison–Wesley, 1995.