, , 1–31 ()
© Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Timing Analysis for Instruction Caches

FRANK MUELLER mueller@informatik.hu-berlin.de
Humboldt University Berlin, Institut f. Informatik, 10099 Berlin (Germany), phone: (+49) (30) 2093-3011, fax:-3010

Editor: Wolfgang A. Halang

Abstract.
This paper contributes a comprehensive study of a framework to bound worst-case instruction cache performance for caches with arbitrary levels of associativity. The framework is formally introduced, operationally described and its correctness is shown. Results of incorporating instruction cache predictions within pipeline simulation show that timing predictions for set-associative caches remain just as tight as predictions for direct-mapped caches. The low cache simulation overhead allows interactive use of the analysis tool and scales well with increasing associativity.

The approach taken is based on a data-flow specification of the problem and provides another step toward worst-case execution time prediction of contemporary architectures and its use in schedulability analysis for hard real-time systems.

Keywords: Hard Real-Time Systems, Static Timing Analysis, Set-Associative Instruction Cache, Worst-Case Execution Time

1. Introduction

Real-time systems are commonly composed of a set of tasks that are either invoked sporadically or periodically. Once invoked each task has a point in time, called deadline, at which its execution should have finished. For such a set of tasks, real-time schedulability theory provides the means to check if the deadlines can always be met – otherwise the safety of the controlled system is jeopardized [20, 5, 7, 30]. If a task does not finish within its deadline, it may not produce a result in time to react to its environment and it may cause other tasks to miss their deadlines.

The schedulability theory for real-time systems relies on *a priori* knowledge of the worst-case execution time (WCET) of hard real-time tasks to check if the deadline of a task can be met. A safe upper bound to approximate the WCET of a task can be provided through static analysis by determining the longest paths through a program segment corresponding to a task and simulating the features of contemporary architectures at the level of processor cycles. Static analysis is commonly automated through a set of tools since *ad hoc* testing methods and approaches relying on timing during program execution result in underestimations of the WCET, even for simple tasks, due to the complexity of contemporary architectures. This paper presents a system of tools that perform timing prediction by statically analyzing optimized code without requiring interaction from the user.

This work provides the means to bound the WCET for instruction caches with arbitrary levels of associativity. It is based on prior work with the group at Florida

State University, which included bounding the instruction cache performance for direct mapped cache [25, 22, 3, 11] and later extensions for set-associate caches [23, 32]. In addition to the prior work, this paper specifies a data-flow framework that describes the problem of cache prediction, also fixes some problems in the instruction categorizations for set-associative caches and provides a number of examples to motivate and illustrate the methods. Thus, this paper presents an approach to include common features of contemporary architectures for static prediction of the WCET. Overall, this work fills an important gap between WCET prediction of contemporary architectures and its use in schedulability analysis for hard real-time systems.

The paper is structured as follows. In Section 2, the work is motivated, the problem is described and the approach for its solution is sketched through examples. In Section 3, interprocedural program analysis is introduced. In Section 4, the method of static cache simulation is defined. In Section 5, a data-flow framework for static cache simulation is formulated and a corresponding algorithm is provided. In Section 6, definitions are provided that allow the specification of the cache behavior for instruction references. In Section 7, the correctness of the methods is shown. In Section 8, examples are presented to illustrate the description of caching behavior. In Section 9, the timing analyzer is described that extends path analysis to interpret the cache behavior provided by static cache analysis. In Section 10, measurements are presented to show the results of timing prediction for caches on some sample programs. In Section 11, future work is discussed. In Section 12, related work is discussed. Finally, in Section 13, conclusions are presented.

2. Motivation and Problem Description

Modern processors generally use instruction and data caches to bridge the increasing gap between ever-faster processors and only moderately faster memory. Most caches are split caches, *i.e.*, instruction cache and data cache are separate. The level of associativity for such caches typically ranges between 1 and 8, as depicted in Table 1 [8].

Table 1. Cache Associativity for various Processors

Associativity	Processors
1	most SPARC, MIPS and Alpha chips
2	Intel Pentium, AMD K6, Alpha 21264/21364, PowerPC 602/603, MIPS R5000/R10000
4	Pentium II, AMD K5, Motorola 68040/68060, PowerPC 604, Cyrix x86, SPARC R1 (HaL)
8	PowerPC 601/620/740/750

2.1. Cache Organization

The content of a a cache memory is stored in a set of *cache lines* (aka. cache blocks), each of which store some data of the next lower memory level. For instance, an

instruction cache between the CPU and main memory with a *line size* (or block size) of 4 instructions stores 4 consecutive instructions in each line. These 4 instructions are referred to as a *program line* when they reside in the lower memory level (main memory) since they are part of some program.

Let L be the number of lines in a cache memory and a be the address of the first instruction in a program line. A *direct-mapped cache* is a cache organization where each program line can be mapped into one cache line. This cache line l is determined as $l = a \ mod \ L$, *e.g.*, in Figure 1(a) $l = 13 \ mod \ 8 = 5$. Conversely, a *fully associative cache* allows a program line to be mapped to any cache line (see Figure 1(b)). An *n-way associative cache* supports a mapping of a program line into a set of n different cache lines and is generally called a *set-associative cache*. Let S be the number of sets of a cache. The set s that a program line is placed in is determined as $s = a \ mod \ S$, *e.g.*, $s = 13 \ mod \ 4 = 1$ in Figure 1(c), which could be cache line 2 or 3. The selection of a cache line within this set is directed by the *replacement policy* for set-associative caches. The most common replacement policy is the least-recently-used (LRU), where the content of the oldest (least recently referenced) cache line is overwritten with the content of the new program line. A cache line is referenced when the CPU issues a data request of a cached program line. In Figure 1(b), cache line 0 is assumed to be the LRU line while in Figure 1(c) cache line 2 is the LRU line. The random policy is less common and is typically based on a pseudo-random number generator to provide reproducible results over repeated program executions. Other policies, such as FIFO, are mostly of a theoretical interest since they are not used for processors. The operational details of referencing cache contents and the remaining cache organization are omitted since they are not relevant for this paper and can be found elsewhere [14].

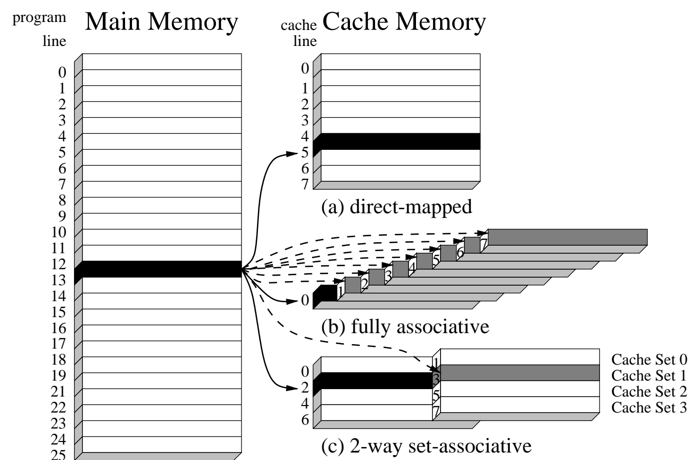


Figure 1. Example of Cache Placement

2.2. *Timing Analysis of Caches*

Timing analysis of program code to determine the the WCET is essential for schedulability analysis. The prediction of the WCET includes path analysis to cover the flow of execution through a program, *i.e.*, the possible execution paths within the control flow of a program. A tight prediction of the WCET can only be guaranteed when the characteristics of the target architecture are taken into account. In the past, unpipelined architectures had a known number of execution cycles for each instruction. For pipelined architectures, the overlap of instruction stages had to be simulated in addition. When cache memories are considered as well, instruction and data references may take a different time to be resolved depending on whether an item is cached or not. Consequently, a tight prediction for cached architectures should include the simulation of caches to determine whether a reference can be resolved in cache or needs to go to the next level of the memory hierarchy. Cache prediction then becomes the task of simulating the changing contents of cache memories relative to a certain stage of program execution, which includes the determination of where data is placed in the cache. For instruction cache simulation, this means that the above mapping rules must be applied to determine which cache line a program line maps into and, if set-associative caches are simulated, how the replacement policy interacts with this placement. The LRU policy can be simulated by aging of unreferenced cache lines and will be assumed for the remainder of this paper since it is the most common replacement policy. Other regular policies like FIFO can easily be incorporated by changing the rules of aging of cache lines and impose the same overhead. However, random replacement for pseudo-random number generators require an extension to the described methods with additional overhead.

2.3. *Cache Prediction*

The knowledge about the contents of a cache memory for a given point of a program's execution can be simulated by an exhaustive enumeration of all possible cache states for any paths taken during execution. Such a cache state is sometimes termed *concrete cache state* (CCS) since it reflects the actual contents of the cache that may be reached during program execution. The CCS represents for each cache line the program line that it contains and has therefore a storage overhead in the order of the number of cache lines. The enumeration of these concrete cache states for a program's execution is exceedingly time and space consuming due to the exponential complexity of the problem, which is rooted in the nature of control flow. Joining edges require that separate concrete cache states be kept for the remainder of execution.

An alternative representation is the *abstract cache state* (ACS), which will be introduced formally in the next section. Informally, it represents an abstract view of the possible cache contents by using sets. The ACS represents for each cache line the *set* of program lines that it *may* contain. The ACS can be regarded as the union of all CCS' at a point of execution where the program lines for a cache

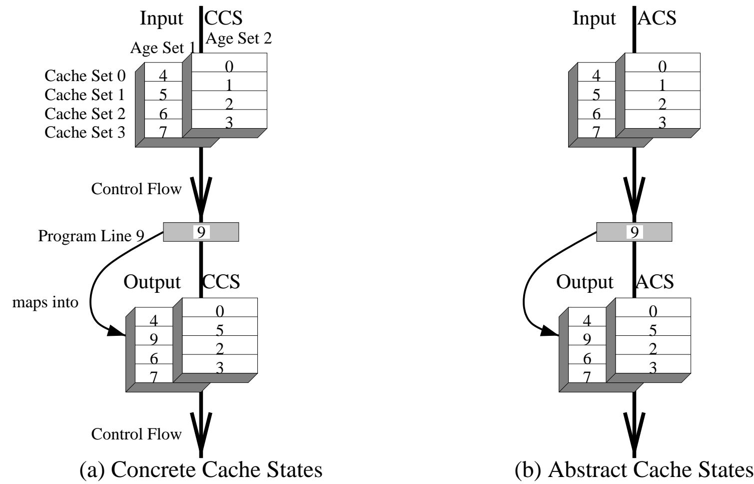


Figure 2. Cache States for Straight-Line Code

line are combined in one set. Thus, the storage complexity is no longer exponential since it is independent of the control-flow structure.

Example: Assume a two-way set-associative cache with four cache lines and LRU replacement. Consider Figure 2(a), which depicts a slice of straight-line code referencing program line 9 (shaded box). For a given input CCS before the fragment, a corresponding output CCS can be determined by applying the mapping and replacement rules. A program line contained in an *age set* with a higher number is older, *i.e.*, the LRU replacement is represented by aging. Notice that the age set, or simply the *set* a program line is contained in, differs from the cache set. Figure 2(b) depicts the input and output ACS, which do not differ from the corresponding CCS for this code fragment.

Figure 3(a) shows a conditional execution. The output CCS of the then-part (program line 9) differs from the output CCS of the else-part (program line 10). After the control flow joins again, both CCS' represent possible cache states and have to be considered for the remainder of program execution. Figure 3(b) depicts the corresponding ACS'. There is only one output ACS containing sets of program lines that may be cached at this point of execution. In effect, the output CCS' are merged into this output ACS. Merging conserves space but reduces the amount of information. For example, the output ACS does not show that either program lines 9 or 10 can be cached.

The loop in Figure 4(a) contains the conditional statement of the prior example. When we consider a single input CCS prior to the loop, two output CCS' describe the possible cache states after the first iteration, and a third CCS is produced if the alternate path of the first iteration is taken during any consecutive iteration. Figure 4(b) depicts the output ACS after the loop, which shows that either program lines 9 or 10 or both of them are cached, thereby correctly reflecting possible

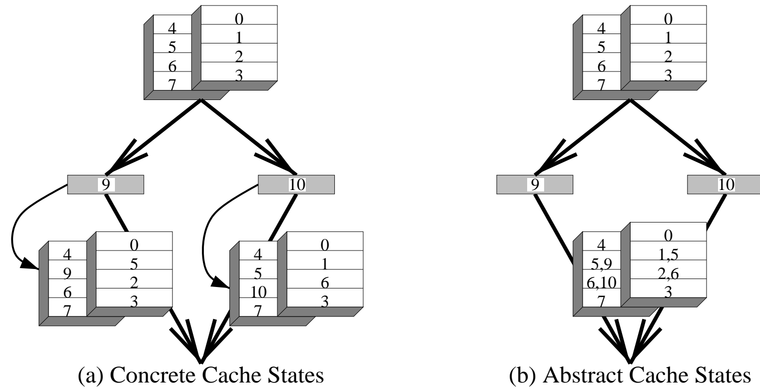


Figure 3. Cache States for Conditional Execution

CCS'. Consider the ACS on the edge from program line 8 to 10. This ACS contains program lines 7 and 11 in the last entry of age set 1. There would be four corresponding CCS': One for the first iteration derived from the input CCS before the loop and three for consecutive iterations derived from the output CCS after the loop. Observe that the ACS contains program line 10 while all other program lines within the same cache set (program lines 2 and 6) and not part of the loop. Thus, the ACS itself contains enough information to deduce that the first reference

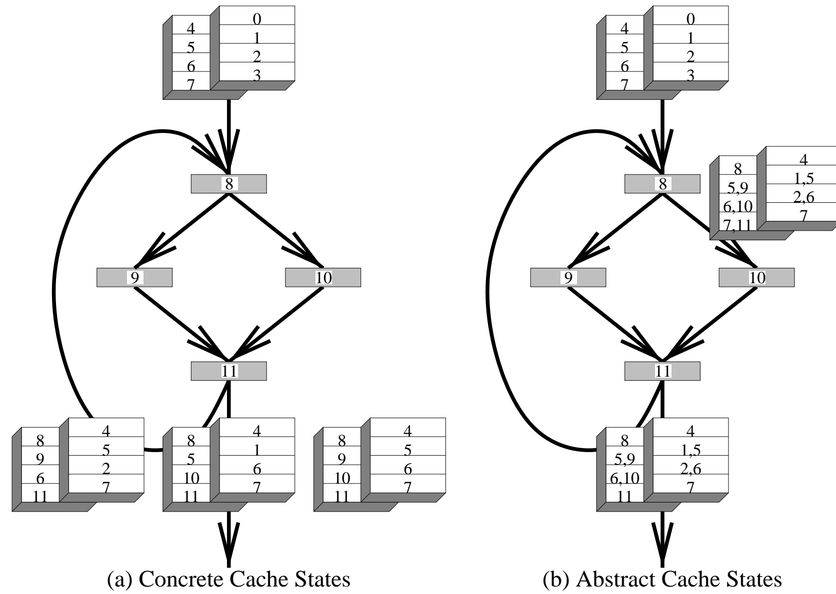


Figure 4. Cache States for a Loop

to program line 10 will result in a cache miss while any consecutive references will result in hits. This type of cache behavior will be termed a "first-miss" in the next section. In fact, the goal of cache prediction is to characterize the cache behavior of each instruction in such a way.

These examples illustrate that the ACS represents enough information to deduce a certain amount of cache behavior while requiring less storage overhead than CCS' to make simulation feasible. The ACS provides the foundation for the method of static cache simulation introduced in the next section.

2.4. Nested Loops and Function Calls

A program containing nested loops complicate the analysis when trying to characterize the cache behavior of a program line within the inner loop. Consider program line 10 in Figure 5(a). If the cache behavior was characterized as a first-miss, would

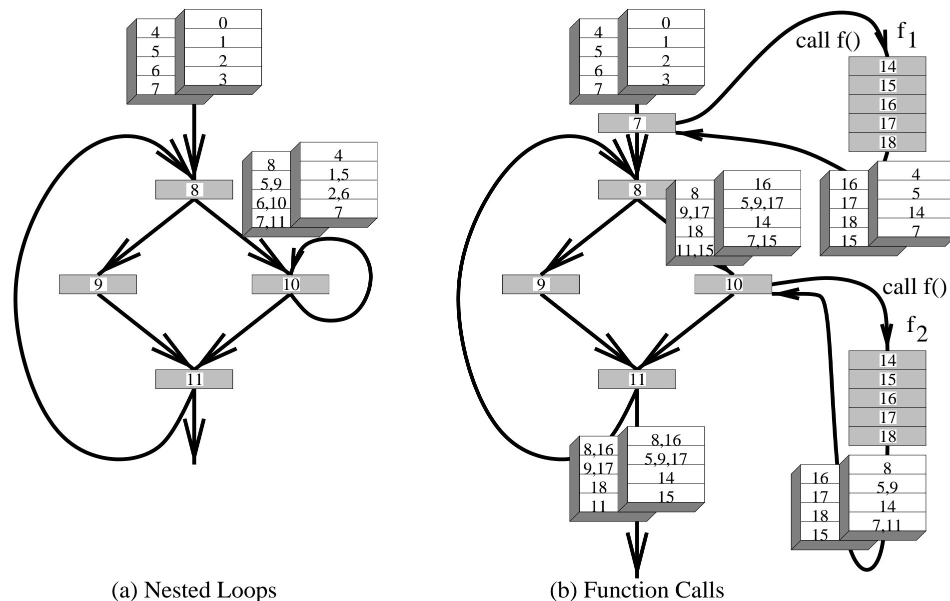


Figure 5. Abstract Cache States for Program Fragments

there be (i) one miss for the entire program execution and hits thereafter or (ii) one miss each time the inner loop is entered and hits thereafter? In Figure 5(a) program line 10 is the only line within the inner and outer loops mapping into the third cache set. Hence, case (i) holds. In Figure 5(b), a function f is called outside as well as within the loop. Function f contains program lines 14 and 18, which map into the same cache set and replace line 10. Hence, case (ii) holds. This illustrates that the cache behavior should be defined with regard to a nesting level for sake of precision. In fact, the framework described in this paper specifies a separate cache

behavior for *each* nesting level for any given instruction of a program. For case (i) depicted in Figure 5(a), program line 10 would be characterized as a first-miss on the outer loop level, as well as the inner loop level since it is contained in the input ACS of line 10 and it is the only line mapping into this cache set within the inner and outer loops. For case (ii) in Figure 5(b) line 10 is considered an "always-miss" at the loop level since it is not contained in the input ACS of line 10. An always-miss describes the behavior resulting in a cache miss for each reference *on this loop level.*

The function f in Figure 5(b) causes similar problems for determining the cache behavior. If the function is called from outside the loop, the references to program lines 14-18 will result in misses. When called within the loop, only the references to program lines 14 and 18 will result in misses, the other references will be hits. The distinction of this behavior cannot be represented in a "flat" description. Hence, the call site of a function is used to distinguish *function instances.* Function instance f_1 refers to the function when invoked from outside the loop and f_2 is invoked within the loop. We can now determine an output ACS for f_1 and another output ACS for f_2, as seen in Figure 5(b). Both output ACS' allow separate inferences about the possible cache contents so that the cache behavior of the program lines within each function instance are distinguished. We also consider each function instance as its own nesting level. For example, for f_2 we describe the caching behavior of program line 17 at the function instance level, the loop level of the caller and the function level of the caller (up to the initial caller, *e.g.*, the function main). The cache set contains lines 5, 9 and 17 and only two lines fit into a 2-way associate cache but line 5 is outside the loop. Hence, a first-miss describes the behavior for line 17 in f_2 on all nesting levels.

Finally, consider Figure 5(b) again. If the call site for f was within program line 9 instead of 10, the cache behavior would be quite different. When only the left path was taken within the loop, program lines 14 and 18 of f_2 would always result in cache hits, termed "always-hits". Conversely, sole execution of the right path would result in always-hits for program line 10. However, if the left and right paths alternate, cache misses would be incurred. Each time the paths alternate, lines 10, 14 and 18 would result in an initial miss. Since the sequence of taken paths within the loop may be data-dependent, there would be no static knowledge about taken paths available during simulation time. We assume the worst-case behavior to bound the WCET in this case, *i.e.*, the cache behavior of lines 10, 14 and 18 would be described as always-misses. Conversely, the best-case execution time would require a description as always-hits.

3. Interprocedural Analysis

The examples in the previous section motivate the distinction of function instances. This section formalizes the framework for an interprocedural analysis. Let us assume a non-recursive program p for now.

Definition 1. [Function Instance]
Let a *basic block* be a sequence of consecutive instructions where the last instruction is a branch and the first instruction is immediately preceded by a label or another branch instruction. Let $CFG(f) = (s, R, B, T)$ be the control-flow graph of function f, where B is the set of basic blocks (vertices), T is the set of control-flow transitions (edges) $v \rightarrow w$ for each branch from $v \in B$ to $w \in B$, $s \in B$ is the start vertex and $R \subseteq B$ is the set of vertices returning from f. (We use $B(f), T(f), ...$ to denote the set $B, T, ...$ of $CFG(f)$ in the following.) Let $CG(p) = (main, F, C)$ be the call graph of program p, where F is the set of functions (vertices), C is the set of function calls (edges) $f \rightarrow g$ if function f contains a call to function g and $main \in F$ is the initial function. Then, the set of function instances is defined recursively:

1. $main_0$ is a function instance.
2. If f_i is a function instance, $(f \rightarrow g) \in C$ and $b \in B(f)$ contains a call to g, then g_{b,f_i} is a function instance.
3. These are all function instances.

The function instance graph $FIG(p) = (main_0, I, K)$ of program p with the set of function instances I then contains an edge $(f_i \rightarrow g_k) \in K$ iff $(f \rightarrow g) \in C$ and $f_i, g_k \in I$.
Example: In Figure 6(a), let function f contain three calls: a call to g and two calls to h. There exists two outgoing edges from f since multiple calls to the same function are folded. Furthermore, function g calls i and k and function h calls k. The corresponding function-instance graph in Figure 6(b) contains two instances of h (for each call from f_0) and three instances of k (for the calls from g_0, h_0, h_1). Since g_0 is the only instance of g, i has only one instance i_0.

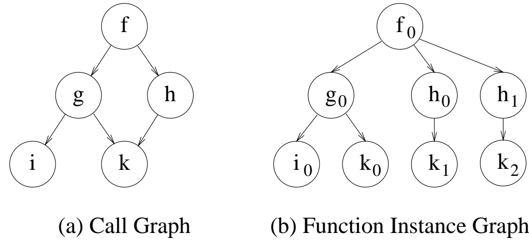


Figure 6. Call Graph and Function-Instance Graph

The function instance graph and the control-flow graphs of the functions can then be combined to describe the interprocedural control flow of a program. The interprocedural control flow has s of $main_0$ as a start node and contains for each function instance f_i a separate set of nodes and edges $CFG(f_i) = (s, R, B, T)$ corresponding to the original control-flow graph of f. However, edges $b \rightarrow c$ whose

source node contains a call $f_i \to g_k$ are replaced by (a) an edge $b \to s$ for to the start node s of g_k and (b) for all $r \in R(g_k)$ edges $r \to c$ are added. An example is depicted in Figure 5(b) where f_1 and f_2 are called from line 7 and 10 but return to line 8 and 11, respectively. Function instances are similar to function inlining in the sense that each instance is considered separately within the interprocedural analysis. It differs, on the other hand, since the simulation environment does not perform actual inlining through code duplication. Recursive calls require special treatment. Since our overall framework does not support recursion at the time, we omit the details in this paper as they can be found elsewhere [22]. The next section describes how caches can be simulated by interprocedural analysis.

4. Static Cache Simulation

The method of static cache simulation provides the means to predict the caching behavior of instruction and data references. The following sections formalize the handling of set-associative instruction caches. An instruction is assigned a category for each loop level to describe its worst-case behavior for repeated references:

1. **Always-Miss (m)**: The reference is not guaranteed to be in cache.
2. **Always-Hit (h)**: The reference is guaranteed to always be in cache.
3. **First-Miss (fm)**: The reference is not guaranteed to be in cache the first time it is accessed each time the loop is entered, but is guaranteed thereafter.
4. **First-Hit (fh)**: The reference is guaranteed to be in cache the first time it is accessed each time the loop is entered, but is not guaranteed thereafter.

The analysis for set-associative instruction caches is based on the following formal framework:

Definition 2. [Potentially Cached] A program line l can be potentially cached if there exists a sequence of transitions in the combined control-flow graphs and function-instance graph such that l is cached when it is reached in the current block.

The process of determining if a line is potentially cached may be performed by a path traversal yielding sets of concrete cache states. However, the combinatorial problem of traversing every possible sequence of paths leads to an exponential explosion in the search space wrt. the branching factor, *e.g.*, for nodes with conditional branches that have two successors in the control flow. Static cache simulation counters this complexity problem via interprocedural data-flow analysis (DFA) modified for the task of cache prediction. DFA within compilers yields sets of live objects, whereas static cache simulation yields sets of cached program lines. The latter sets are referred to as abstract cache states.

Definition 3. [Abstract Cache State (ACS)] The abstract cache state of a program line l within a path and a function instance is the set of program lines that can potentially be cached prior to the execution of l within the path and the function instance.

For direct-mapped caches, the ACS is a singleton set used to determine the category of an instruction describing the cache behavior. For an n-way set-associative cache, the ACS is an n-tuple of sets, used for the purpose of instruction categorization. However, n different sets, called age sets, are employed to support the operational framework of DFA simulating the cache invalidation protocol of set-associative caches. Examples of an ACS for 2-way associative caches could be seen in Figures 2 – 5. The next section shows how the ACS is determined.

5. Data-Flow Framework

Given the control-flow information of a program and a cache configuration, the ACS' for each path have to be calculated. Using DFA, each path has an input state and an output state, corresponding to the ACS before and after the execution of the path, respectively.

Before program execution, the cache is assumed to be invalid, *i.e.*, it does not contain any lines of the program. Thus, the input state of the first path contains only invalid lines. As a path is executed, its lines are cached, *i.e.*, they are added to the output state. When caching a line, it may replace a conflicting line within the current state. The data-flow equations of the ACS for direct-mapped instruction caches are defined as follows:

$$in(start, first) = \{invalid\} \quad (1)$$
$$in(p \neq start, first) = \bigcup_{q \in preds(p)} out(q, last) \quad (2)$$
$$in(p, l \neq first) = out(p, l-1) \quad (3)$$
$$out(p, l) = defs(p, l) \cup (in(p, l) \setminus confs(p, l)) \quad (4)$$
$$defs(p, l) = \{l\} \quad (5)$$
$$confs(p, l) = \{m \in in(p, l) | m \neq l \wedge m \rightarrow c \wedge l \rightarrow c\} \quad (6)$$

The input set *in* of the *first* program line within the *start* path of the interprocedural control flow describes the *invalid* ACS. The input state of the first line of any other path p is composed of the union of the output sets of its predecessors q and their *last* program line in the control flow. The input state of any other line is given by the output state of the previous line within the current path. The output set *out* after program line l in path p is composed of its input state plus the encountered program line l in *defs* minus any conflicting lines *confs* that map (denoted as $\rightarrow$) into the same cache line c as l. This concludes the explanation of the data-flow equations for direct-mapped instruction caches.

For set-associative instruction caches, conflicting lines are subject to the replacement policy. We assume the LRU policy where the least-recently used line is replaced within the cache set of an n-way set-associative cache. Other cached lines

"age" upon such a reference. Given the n-tuple of an ACS, this replacement process is simulated by "shifting" the replaced conflicting lines of the 1st cache state to the 2nd cache state. If any lines were shifted, they subsequently cause conflicting lines in the 2nd set to be shifted to the 3rd set, *i.e.*, the shifting operation cascades until the conflicting lines in the n-th set are kicked out of the cache. Finally, the input state of a path with predecessors in the control flow is obtained by the union of output states of its predecessors, *i.e.*, any potentially cached line is included along the control flow. For each set of the n-tuple, the union of the predecessors of the same tuple is calculated separately. The data-flow equations of the ACS for an n-way set-associative instruction cache include a set for each level of associativity and are defined as follows:

$$
\begin{array}{lll}
in(start, first, 1..n) & = \{invalid\}, \phi, ..., \phi & (1)\\
in(p \neq start, first, set) & = \bigcup_{q \in preds(p)} out(q, last, set) & (2)\\
in(p, l \neq first, set) & = out(p, l-1, set) & (3)\\
out(p, l, set) & = defs(p, l, set) \cup (in(p, l, set) \setminus confs(p, l, set)) & (4)\\
defs(p, l, 1) & = \{l\} & (5a)\\
defs(p, l, set \neq 1) & = confs(p, l, set-1) & (5b)\\
confs(p, l, 1) & = \{m \in in(p, l, 1) | m \neq l \wedge m \to c \wedge l \to c\} & (6a)\\
confs(p, l, set \neq 1) & = \{m \in defs(p, l, set) | m \neq l \wedge m \to c \wedge l \to c\} & (6b)
\end{array}
$$

The equations (1) to (4) for input and output states remain unchanged, except that each associativity level *set* is distinguished. The defined program lines *defs* (5a) and conflicting program lines *confs* (6a) for associativity level 1 are the same as before, namely line l and all lines in the input state that map into the same cache line as l, respectively. The defined lines *defs* (5b) of larger associativity levels are simply the conflicting lines *confs* of the previous level. The conflicting set *confs* (6b) of larger associativity levels contains those lines of the defined set *defs* (at the same associativity level) that map into the same cache line as l. Thus, *defs* and *confs* model the LRU policy through (5b) and (6b). Other replacement policies, such as FIFO, can be modeled by modifying the equations for *defs* and *confs*. The ACS' depicted in the context of Figures 2 – 5 can be derived using the above equations with the exception of invalid lines, which were omitted from the Figures.

Figure 7 depicts the algorithm to calculate the ACS for an n-way associative cache. The algorithm combines the sequence of equations for all program lines within one path and, as a result, uses `input(P, set)` and `output(P, set)` to denote $in(p, first, set)$ and $out(p, last, set)$, respectively. The *defs* and *confs* sets are combined as one set in `diff`. The first path of function `main` (the program start path) is invalidated wrt. the incoming ACS of the 1st tuple. For all other paths, the input states are calculated as the union of the predecessors' output states. The output path of the 1st tuple is determined as the input state of this tuple and the (newly cached) program lines less the conflicting lines. The conflicting lines cascade through the tuple space (*via* the `diff` sets): Together with the input state of the next tuple, the conflicting lines of the last tuple represent the output state of this next tuple while program lines and the new conflicting lines (of this tuple) are

subtracted before the next cascade. To keep the algorithm simple, it is assumed that there do not exist any conflicting program lines within a single path. This can be ensured by splitting paths into disjoint program lines. Instead of path splitting, the actual implementation uses an enhanced algorithm to shift the `diff` sets upon encountering conflicting lines within a single path.

This DFA requires a time overhead comparable to that of interprocedural DFA performed in optimizing compilers. The space overhead is $O(pl * bb * fi * n)$, where pl, bb, fi, n denote the number of program lines, basic blocks, function instances, and cache associativity, respectively. Notice that set-associative caches impose a factor of n, which is typically very small ($1 \leq n \leq 8$) for instruction caches in contemporary architectures (for direct-mapped caches $n = 1$). The correctness of iterative DFA has been discussed elsewhere [1].

6. Deriving Instruction Categorizations

Instructions have to be categorized for each loop level based on the ACS. Some additional data-flow information is required to determine these categories, namely the linear cache state, the dominator cache state and the post-dominator set for each path.

The linear cache state is based on the forward control-flow graph, *i.e.*, the acyclic graph resulting from the removal of backedges (backwards edges forming loops [1]) in the regular control-flow graph.

Definition 4. [Linear Cache State (LCS)] The linear cache state of a program line l within a path and a function instance is the set of program lines that can potentially be cached in the forward control-flow graph prior to the execution of l within the path and the function instance.

Informally, the LCS represents the hypothetical cache state in the absence of loops. This data-flow information represents the lines that may be cached between the entry of a loop to the current path on the first iteration. It will be used to determine whether a program line may be cached due to loops or due to the sequential control flow. The algorithm to calculate the ACS can also be used to calculate the LCS by simply using the forward control flow. (The problems of data-flow equations in a forward control-flow graph and in a backward control-flow graph are orthogonal [1]. The former follows the predecessors while the latter follows the successors. The former relates the output to the input of the same path, the latter reverses this relation.) As a result, the LCS is an n-tuple of sets of program lines, assuming an n-way set-associative cache.

The dominator cache state is defined next, *i.e.*, the set of cache lines that must be cached at some point (in analogy to the dominator set commonly employed in program analysis [1]). It can be calculated by DFA as well and results in a singleton set, even for set-associative caches.

Definition 5. [Dominator Cache State (DCS)] The dominator cache state of a program line l within a path and a function instance is the set of program lines that

Input: Function-Instance Graph of the program and control-flow graph for each function.
Output: Abstract Cache State for each path.
Algorithm: Let `prog_lines(P)` be the set of program lines of path P. Let `map_into_same_line(s, t)` return the subset of lines in `s` that map into the same cache line as any lines in `t`. Let n be the cache associativity. X ∪= Y is short for X := X ∪ Y; ditto for \=.

```
1:  input(main, 1):= all invalid lines;
2:  WHILE any change DO
3:     FOR each instance of a path P in the program DO
4:        FOR set := 1 TO n DO
5:           input(P, set):= ϕ;
6:           FOR each immed. predecessor Pred of P DO
7:              input(P, set) ∪= output(Pred, set);
8:        FOR set := 1 TO n DO
9:           diff(set) := ϕ;
10:       FOR set := 1 TO n DO
11:          output(P, set) := input(P, set);
12:          FOREACH line ∈ prog_lines(P) DO
13:             IF set = 1 THEN
14:                output(P, set) ∪= {line};
15:             ELSE
16:                output(P, set) \= {line};
17:                IF map_into_same_line(diff(set - 1), line) \ {line} = ϕ THEN
18:                   CONTINUE;
19:                diff(set - 1) \= {line};
20:                output(P, set) ∪= diff(set - 1);
21:             IF set = 1 OR diff(set - 1) ≠ ϕ THEN
22:                FOREACH conf_line ∈ map_into_-
23:                   same_line(output(P, set) \ {line}, line) DO
24:                   IF set = 1 OR conf_line ∉ diff(set - 1) THEN
25:                      output(P, set) \= {conf_line};
26:                      diff(set) ∪= {conf_line};
```

Figure 7. Calculation of Abstract Cache States

are known be cached prior to the execution of l within the path and the function instance.

The DCS can be used to determine if a program line must be cached due to an earlier reference, regardless of the sequence of execution paths that were taken.

The post-dominator set of a path includes the program lines certain to be reached from the current path, regardless of the taken paths in the control flow. It can also be calculated by DFA and results in a singleton set, even for set-associative caches.

Definition 6. [Post-dominator Set (PDS)] The post-dominator set of a program line l within a path and a function instance is the self-reflexive transitive closure of post-dominating program lines.

This information is commonly used wrt. basic blocks in optimizing compilers. A more detailed discussion of dominators and post-dominators as well as algorithmic details can be found elsewhere [1].

The instruction categories can now be defined with respect to the available data-flow information. Definition 7 formalizes the worst-case instruction categories for each loop level.

Definition 7. [Worst-Case Instruction Categorization]

- Let i_k be an instruction within a path, a loop λ, and a function instance.
- Let n be the degree of associativity of the cache.
- Let $l = i_0..i_{m-1}$ be the program line containing i_k and let i_{first} be the first instruction of l within the path.
- Let s_j be the j-th component of the ACS (n-tuple) for l within the path and let $s = \bigcup_{1 \leq j \leq n} s_j$.
- Let l map into cache line c, denoted by $l \rightarrow c$.
- Let u be the set of program lines in loop λ.
- Let $child(\lambda)$ be the child loop (inner-next loop within nesting) of λ for this path and function instance, if such a child loop exists.
- Let $headers(\lambda)$ be the set of header paths and $preheaders(\lambda)$ be the set of preheader paths of loop λ, respectively.[1]
- Let $s(p)$ be the abstract output cache state of path p.
- Let $linear_j$ be the j-th component of the LCS (n-tuple) for l within the path and let $linear = \bigcup_{1 \leq j \leq n} linear_j$.
- Let dom be the set of dominating program lines (DCS) of path p.
- Let $postdom(p)$ be the set of self-reflexive post-dominating program lines of path p.

Then, $category(i_k, \lambda) =$

$$\begin{cases} \text{always-hit} & \text{if } k \neq first \vee (l \in dom \wedge \\ & [\exists_{1 \leq j \leq n} \; l \in s_j \wedge (\sum_{m \rightarrow c, m \neq l} |m \in s_j| = 0 \vee \sum_{m \rightarrow c, m \neq l} |m \in s| < n)]) \\ \text{first-hit} & \text{if } category(i_k, child(\lambda)) = \text{first-hit} \vee k = first \wedge l \in s \wedge l \in dom \wedge \\ & \forall_{p \in headers(\lambda)} \; l \in postdom(p) \wedge \sum_{m \rightarrow c, m \neq l} |m \in (s \cap u)| \geq n \wedge \\ & \sum_{\substack{m \rightarrow c, m \neq l \\ p \in preheaders(\lambda)}} |m \in ((s(p) \cup linear) \cap u)| < n \wedge \\ \text{first-miss} & \text{if } category(i_k, child(\lambda)) = \text{first-miss} \wedge k = first \wedge l \in s \wedge \\ & \sum_{m \rightarrow c, m \neq l} |m \in s| \geq n \wedge \sum_{m \rightarrow c, m \neq l} |m \in (s \cap u)| < n \\ \text{always-miss} & \text{otherwise} \end{cases}$$

Different loop levels can be distinguished by extracting only the program lines of the data-flow information within the current loop level. Operationally, this can be achieved efficiently by intersecting the set of program lines within the loops with any data-flow set of program lines.

While the definition seems complex, it can be implemented rather efficiently once the data-flow information has been calculated. First, simple operations on sets (that can be implemented as bit vectors) suffice to test the conditions. Second, if one conjunct in a condition fails, the remaining ones are not tested. Third, the implementation orders the conjuncts such that the least likely ones are tested first. To motivate this definition, an informal description of the conditions shall be given.

Always-hit: (on spatial locality within the program line) or (the line of the instruction dominates the path and (the instruction is potentially in some cache cache state) and ((there are no conflicting instructions in this cache state) or (all conflicts fit into the remaining associativity levels))).

First-hit: (the instruction was a first-hit for inner loops) or (it is potentially cached, it dominates the path, it is always executed in the loop, not all conflicts fit into the remaining associativity levels but the conflicts at loop entry plus those inside the loop prior to l at the level for the loop fit into the remaining associativity levels).

First-miss: the instruction was a first-miss for inner loops, it is potentially cached, conflicts do not fit into the remaining associativity levels but the conflicts within the loop do.

Always-miss: This is the conservative assumption for the prediction of worst-case execution time when none of the above conditions apply.

7. Correctness

This section discusses the issue of correctness of the instruction categorization. First, the algorithm in Figure 7 is revisited. Next, Definition 7 is discussed.

The algorithm in Figure 7 depicts a solution to solving data-flow recurrence equations by iterating over the forward control flow of a program (inter-procedurally). Lines 1-3, 5-7, 11-12 and 14 comprise the problem-independent part and have been discussed in the literature [1]. In particular, line 1 represents equation (1), lines 3-7 is due to equation (2), line 11 stems from (4) and line 12 iterates over the program lines according to (3). Line 14 has been adapted to the problem domain according to (5a) by calculating the data flow for program lines. The changes to this original algorithm are twofold: First, each input consists of an n-tuple for each associativity level. Second, upon traversing a path of the control flow, the corresponding program lines are not just added to the data-flow sets. Existing lines may also be removed or shifted within the data-flow sets simulating the replacement policy of the cache, *i.e.*, LRU in this case.

The former property, the n-tuple, allows the distinction of storage space for programming lines within an n-way associative cache. Line 4 ensures that the data-flow

analysis is performed independently for each tuple level (up to line 15). Hence, the correctness of the problem-independent algorithm still holds for each tuple level.

The latter property, the simulation of LRU replacement, encompasses the initialization of a difference tuple *diff* in lines 8-9 and the realization of the LRU aging policy from line 13 onwards. Aging in set-associative caches comprises:

(a) including program lines as they are encountered,

(b) incrementing the age of cached lines and

(c) excluding the least recently used line upon exceeding cache capacity.

To calculate the *output* set from the *input* set, each program *line* is first included (cached) in the *output* set 1 (in line 14), thereby ensuring property (a) and equation (5a). Each conflicting line is excluded from *output* set 1 due to equation (6a) and included in the *diff* set 1 (lines 21-26 and equation (4)), which isolates lines subject to aging according to (b). For subsequent sets i greater than 1, the program *line* is excluded in the *output* set i, which is consistent with (a) since the *line* has just been cached. At the same time, conflicting lines of the previous *diff* set of $i-1$ are included (shifted/aged according to (b), in lines 16-20 due to equation (5b)). Any conflicting lines already residing in *output* set i before inclusion of the previous *diff* are excluded from *output* and included in the next *diff* set i (again consistent with (b), lines 22-26 and equation (6b)). Finally, *diff* set n is dropped, which implements (c) since it contains the least recently used lines (excluded from *output* set n). Should *diff* set i not contain any conflicting lines, then all conflicting lines in *output* sets greater i remain unchanged, reflecting the property (c) that only requires exclusion from cache upon exceeding its capacity (lines 17-18).

Next, Definition 7 is discussed for each category. First, observe that for the worst-case categorization it suffices to ensure the correct categorization of references resulting in a cache hit since cache misses are a safe and conservative assumption for the WCET. Using an indirect proof for a certain category, one assumes that references categorized as cache hits actually result is cache misses and show that this assumption leads to a contradiction.

To prove always-hits correct, assume that an instruction reference was not the first reference of a program line within the current path ($k \neq first$) and assume that this reference was a miss. However, the previous instruction in cache shares the same program line and was referenced right before and is the most recently used item in cache. Hence, the current reference should have been a hit; contradiction. Assume that right part of the disjunct for always-hits holds and the reference results in a miss. If line l is always referenced prior to the current path (dominator information) and it has not been replaced in cache since the most recent reference, then it would have to be in cache (contradiction). We can show that l has not been replaced since either the number of conflicting lines that may have been referenced does not exceed the cache capacity or l resides in some abstract cache state that does not contain any conflicting lines, *i.e.*, the capacity of the cache suffices to hold l.

For first-hits, let us assume that the right part of the disjunct holds. Assume the first reference was a miss. Since l has to be referenced before (dominator), since l

may be cached on any loop entry, and since the number of conflicts wrt. to current loop level is less than the cache capacity (ACS for preheaders plus LCS), l must still be in cache during the first loop iteration. (The union of (a) the ACS for preheaders and (b) the LCS represents the lines that may be cached prior to loop entry (for (a)) and the lines that may be cached after loop entry on the first loop iteration (for (b)) before line l in the path is encountered.) Hence, the first reference must be a hit; contradiction. For subsequent iterations, it is safe to assume misses for the worst-case categorization. Notice that the remaining conditions for first-hits ensure properties used by the timing analyzer (*e.g.*, the condition involving the PDS) or simplify the algorithmic decision process (for the remaining checks).

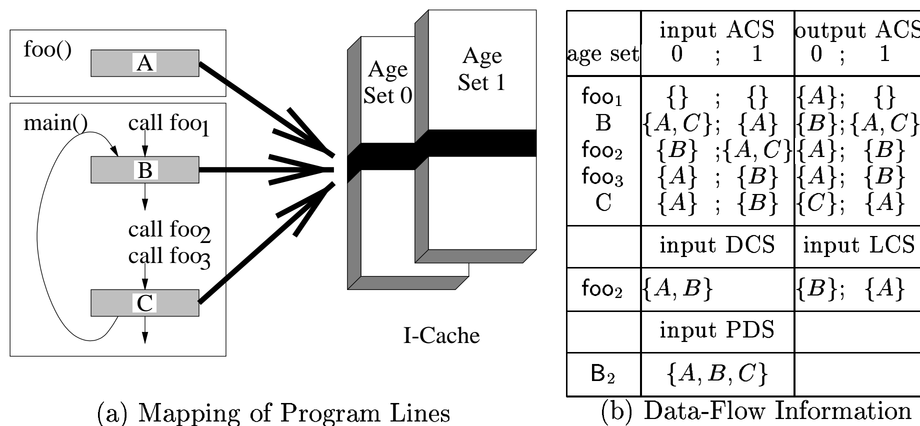
For first-misses, let us assume that the condition in the definition holds and that consecutive references are misses. However, if line l may be cached and the number of conflicting lines inside the loop is less than the cache capacity, then l will be brought into cache after the first reference and remains in cache for consecutive iterations due to the capacity argument. Hence, consecutive references result in hits; contradiction. For the first iteration, it is safe to assume a miss for the worst-case categorization. Notice that the first reference of l may not necessarily occur during the first iteration of the loop if there is conditional control flow. This situation is handled by the timing analyzer, which tracks the first misses that have been referenced along the longest paths within a loop (for the WCET) and is explained later.

Always-misses refer to references that are safely interpreted as misses for the worst-case categorization. This concludes the correctness argument.

8. Examples

Consider the simple example in Figure 8. Program lines A, B, and C map into the same cache line within a 2-way set-associative cache. B and C are executed within a loop. A is executed before the loop (instance foo_1) and twice within the loop (instances foo_2 and foo_3, respectively). For foo_1, A is an always miss since A is not in the input ACS of foo_1. For foo_2, A is a first-hit since it was brought into cache by foo_1, A is in the input DCS and A is in the input PDS of B, which is the header. Furthermore, there are 3 conflicts within the loop (and only 2 cache lines within the set) but only A and B are in the output ACS of foo_1 (the preheader) plus the input LCS of foo_2 wrt. lines of the loop level. For foo_3, A is an always-hit due to temporal locality since it was brought into the input ACS of foo_3 by foo_2 and there are no conflicts in this ACS. Notice that B and C are always-misses since they are not in the input ACS of their basic blocks. However, if there were no calls to foo within the loop, B and C would be first-misses since they would remain in the sets once they are referenced (*i.e.*, brought into cache) and there are only 2 conflicting lines within the loop.

The next example depicts the C code (Figure 9) and the assembly code (Figure 10) of a program calculating the sum of positive elements of an array less the number of non-positive elements. It consists of the functions main and value, the latter being distinguished as function instance value_1 and value_2 called from basic blocks 3 and



age set	input ACS 0 ; 1	output ACS 0 ; 1
foo_1	$\{\}$; $\{\}$	$\{A\}$; $\{\}$
B	$\{A,C\}$; $\{A\}$	$\{B\}$; $\{A,C\}$
foo_2	$\{B\}$; $\{A,C\}$	$\{A\}$; $\{B\}$
foo_3	$\{A\}$; $\{B\}$	$\{A\}$; $\{B\}$
C	$\{A\}$; $\{B\}$	$\{C\}$; $\{A\}$
	input DCS	input LCS
foo_2	$\{A,B\}$	$\{B\}$; $\{A\}$
	input PDS	
B_2	$\{A,B,C\}$	

(b) Data-Flow Information

Figure 8. Example of Instruction Categories

5, respectively. Assuming a 2-way set-associative cache with four instructions per cache line and a total of two cache sets, program lines $\{0, 2, 4, 6\}$ are in conflict such that only two of these lines can be cached at a time (2-way set-associative cache), and program lines $\{1, 3, 5\}$ are also in conflict such that two of these lines can be

```
int a[10] = {1,2,3,4,5,6,7,8,9,10};
int init = 0;

int value(index)
int index;
{
   return a[index];
}

int main() {
   int i, sum, neg;

   sum = neg = init;
   for (i = 0; i < 10; i++) {
      if (value(i) > 0)
         sum = sum + value(i);
      else
         neg++;
   }
   return sum - neg;
}
```

Figure 9. C Code: Summation of Positive Array Elements

cached at the same time. The static cache simulator determined the categories (at the right of each instruction) that could not have been detected by manual inspection of sequential sections of instructions. Static cache simulation does not only handle such spatial locality but also temporal locality, across loops as well as interprocedurally.

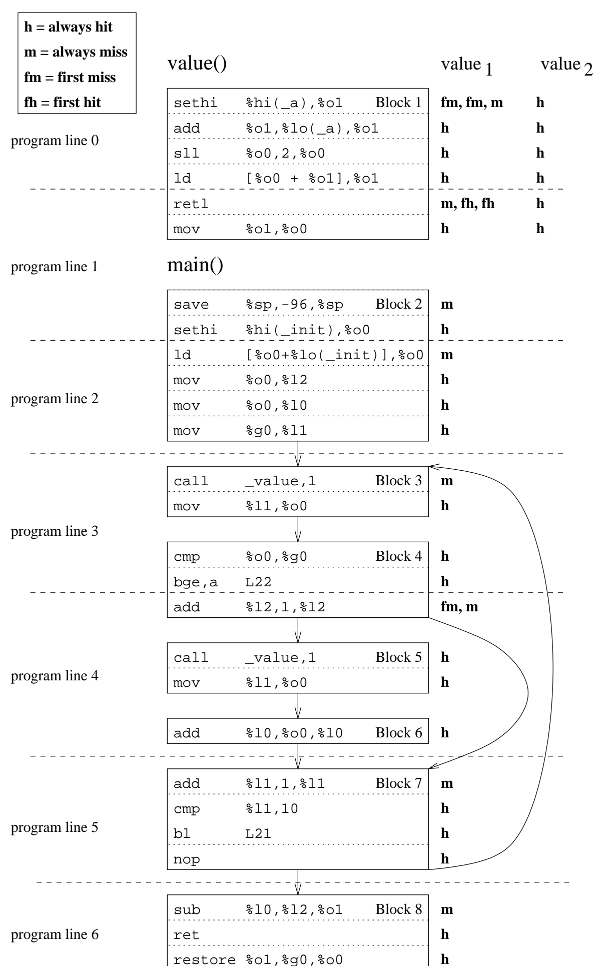


Figure 10. Control Flow, Instructions and their Categories of the Example

For instance, the first instruction of value_1 is a first-miss at the innermost loop level (the function instance of value_1 corresponding to the first call in main) and the next loop level (loop within main). The instruction is an always-miss at the outermost level (function main). The function instance value_1 is called in basic block 3 within the loop in main and program line 0 is uncached when the loop is entered. But this line will remain in cache on subsequent executions of basic block

3 (temporal locality) since there exists only one more conflicting line (program line 4) within the loop and a 2-way set-associative cache can hold both lines. Thus, a first-miss is reported for the function instance and the loop. This also explains the first-miss at the loop level of instruction 3 in basic block 4, which is the first reference to program line 4. In both cases, an always-miss is reported for the outermost level since main, as a function, is only considered to iterate once. A first-miss behaves equivalent to an always-miss on the first iteration. The distinction between categories results from Definition 7 and will be explained later.

Now, consider the first instruction of value_2, an always-hit. The call to value_1 in basic block 3 precedes the execution of basic block 5 calling value_2. Thus, program lines 0 and 4 are cached when value_2 is called. Thus, the first instruction of value_2 is an always-hit due to temporal locality.

Instruction 5 of value_1 belongs to program line 1. This line is in conflict with lines 3 and 5, and all three lines are referenced in each loop iteration. Instruction 5 of value_1 was determined to be an always-miss for the innermost loop level and a first-hit for higher levels. Consider the outermost level first: On the first call to value_1, program line 1 is still in cache since the loop had just been entered from basic block 2 (containing this line) and basic block 3 also brought line 3 in cache, causing a hit for instruction 5. This is equivalent to a first-hit at the outermost loop level (the function level of main).

At the next loop level, the hit remains on loop entry. On subsequent iterations, lines 5 and 3 will replace line 1 when basic blocks 7 and 3 are executed, respectively. This results in misses for instruction 5 of value_1 starting with the second iteration. Thus, the instruction is categorized as a first-hit at this level.

The innermost loop level corresponds to the function level of value_1, where an always-miss is reported. This level has a loop frequency of one iteration since it corresponds to a function. When the timing analyzer determines worst-case predictions for program subranges like this function instance, it has to report the worst case. Since value_1 is executed in a loop and this line was a first-hit within the loop, the worst case for a *single* iteration is a miss. While the static cache simulator supplies the worst-case scenario for each loop level, the timing analyzer decides which values to use according to the (loop) level of analysis.

Instruction 5 of value_2, on the other hand, is an always-hit since line 1 (and line 3) are still cached from the call to value_1. This is an example of temporal locality.

So far, the categorizations have been motivated by informal arguments based on analyzing the execution paths of the program. The static cache simulator, on the other hand, does not performs such path analysis. Instruction categories are instead based on the data-flow information and derived from Definition 7.

Consider instruction 1 of value_1 again. Line 0 is not in the current DCS (see Figure 11). Thus, an always-hit or first-hit can be counted out. But line 0 is in the ACS (in ACS[1]), and lines 2 and 4 are in ACS[0]. When only the lines within the innermost loop (function value) are regarded (ACS intersect lines at loop level), line 0 remains by itself. Thus, the line is categorized as a first-miss at the innermost level. The same holds for the next loop level (the loop within main) since only lines 0 and 4 are in the intersection. At the outermost level (function main), all three

age set						0					;					1					
cache line		0	1	0	1	0	1	0	1	0		0	1	0	1	0	1	0	1	0	
program line		I	I	0	1	2	3	4	5	6		I	I	0	1	2	3	4	5	6	
input ACS($value_1$ path 1)	{					2	3	4			} ; {	I		0	1				5		}
output ACS(main path 2)	{				1	2					} ; {	I	I								}
input LCS($value_1$ path 1)	{					2	3				} ; {	I			1						}
input DCS($value_1$ path 1)	{	I	I		1	2	3				}										
input PDS(main path 3)	{			0	1		3	4	5	6	}										

Figure 11. Data-Flow Sets for Selected Basic Blocks

lines are in the intersection. Thus, an always-miss is reported. This is consistent since a first-miss at the the inner levels corresponds to an always-miss for the first iteration, *i.e.*, the function level of main (with one iteration).

Finally, consider instruction 5 of $value_1$ at the level of the loop within main, categorized as a first-hit. The ACS contains lines 1, 3, and 5. Both lines 1 and 5 are in ACS[1]. Thus, an always-hit can be counted out. Line 1 is in the DCS, in the output ACS of basic block 2 (preheader), and in the post-dominator of basic block 3 (header). Lines 3 and 5 are in the intersection between ACS and lines in the loop, *i.e.*, the number of lines in the intersection equals to the level of associativity ($n = 2$). There are no conflicting lines in the output ACS of basic block 2 (preheader), and line 3 is the only conflicting line in the LCS at this loop level. Thus, the instruction is categorized as a first-hit.

9. Timing Analysis

The timing analyzer calculates the WCET by constructing a timing tree, traversing paths within each loop level, and propagating this timing information bottom-up within the tree. During this traversal, the timing analyzer has to take hardware characteristics into account (*e.g.*, pipelining) and the instruction categorizations have to be interpreted. However, the timing analyzer does not have to take the cache configuration into account. The approach of splitting cache analysis via static cache simulation and timing analysis makes the caching aspects completely transparent to the timing analyzer. Based on the instruction categorizations, the timing analyzer can derive the WCET by propagating timing predictions bottom-up within the timing tree. This derivation process shall be described in more detail.

The timing tree represents the calling structure and the loop structure of the entire program. Functions are distinguished by their calling paths into function instances. This allows a tighter WCET prediction that is mostly due to more accurate caching categorizations associated with the distinguished calling contexts. Each function instance is regarded as a loop level (with one iteration) and is represented as a node

in the timing tree. Regular loops within the program are represented as child nodes of its surrounding function instance (outermost loops) or as child nodes of another loop that they are nested in.

The timing analyzer determines the WCET in a bottom-up traversal of the tree. For any node, all possible paths (sequences of basic blocks) within the current loop level have to be analyzed. When a child node, which represents a nested loop or invoked function, is encountered along a path, its WCET is already calculated and can simply be added to the WCET of the current path after multiplying it with the number of iterations of the current nesting level.

Adjustments are necessary for transitions from (a) first-misses to first-misses and (b) always-misses to first-hits between loop levels since the caching behavior of the instructions with these transitions is not the same for each invocation of the inner loop. In case (a), the WCET of a child node is reduced for each transition by the miss penalty, which is the difference between the overhead for a cache miss and a cache hit. The resulting time of the child node is then multiplied by the number of iterations for the current nesting level before the miss penalty is added once for each transition. This accounts for a miss only on the *first* reference within the loop instead of a miss for *all* references within the loop. In case (b), the time of the child is multiplied with the number of iterations before the miss penalty is subtracted once for each transition. This accounts for a *hit* only on the first reference within the loop instead of a *miss* on the first reference (see [4]).

For a loop with n iterations, a fix-point algorithm is used to determine the cumulative WCET of the loop along a sequence of (possibly different) paths. Once a pattern of longest paths has been established, the remaining iterations can be calculated by a closed formula. In practice, a single path in most loops is the longest path, regardless of changes in caching behavior due to instructions classified as first misses and first hits. Thus, the first iteration is needed to adjust the WCET of child loops along the path, and the second iteration represents the fix-point time for all remaining iterations. The timing analyzer separately evaluates each loop and function within the program, making the analysis very efficient compared to an exhaustive analysis of all permutations of paths within a program. See [4] for a more detailed description of the the timing analyzer.

Consider the example from Figure 10 again. The timing analyzer predicts the WCET by traversing a timing tree, consisting of a node for each loop level (see Figure 12). The leaf nodes correspond to the function instances of value, each with a maximum number of one iteration. The loop within main has a maximum iteration count of 10, and main has an iteration count of one again since it is a function.

The WCET of value_1 is given by a miss and 5 hits (either instruction 5 misses on the first iteration or instruction 1 misses on consecutive iterations). Assuming an overhead of a cache miss of 10 cycles and one cycle for a cache hit, which yields a miss penalty of 9 cycles, the predicted WCET for value_1 is 15 cycles. Since value_2 consists of 6 hits, 6 cycles are predicted.

The WCET at a non-leaf node is calculated by taking the values of the children's nodes, adjusting them if necessary, and then adding the estimates of instructions

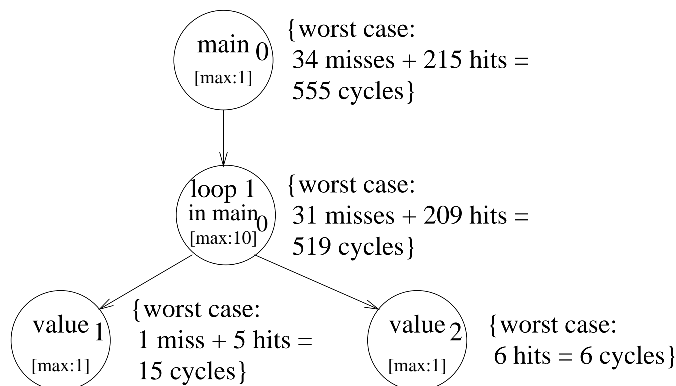


Figure 12. Timing Tree with WCET Prediction

at the current level. Separating the calculation of each node speeds up the process of WCET prediction considerably.

The loop's WCET in main is bounded by executing the longer path (that includes basic blocks 5 and 6) during each iteration. Within the loop, there are 2 always-misses, 9 always-hits and one first-miss over 10 iterations resulting in (2*10+9+1)*10+9=309 cycles. The child nodes are each executed 10 times as well. For value_2 this yield 6*10=60 additional cycles. For value_1 transitions `fm, fm` and `m, fh` require the adjustments described above: The time for the child is reduced by the miss penalty of 9 cycles before multiplied by 10 iterations. Then it is adjusted by adding the penalty of 9 for the transition `fm, fm` and by subtracting the penalty of 9 for `m, fh`, which results in (24-9)*10+9-9=150 cycles. The entire loop node is then predicted to take 309+60+150=519 cycles.

For the level of main, 6 hits and 3 misses are added to result in 555 cycles. This WCET, estimated by static analysis without program execution, is 100% accurate. We confirmed these numbers by measuring the cache behavior of the program's execution with a trace-driven instruction cache simulator on the worst-case input data. The measurements in the next section illustrate that 100% accuracy may not always be achieved and discusses the reasons for these cases.

10. Measurements

Figure 13 illustrates the tools employed within framework of WCET prediction and their interaction. The *vpo* optimizing compiler [6] has been augmented to emit control-flow information and to supply information about the calling structure of functions in addition to regular object code generation. The static cache simulator combines the control-flow information and calling structure to perform interprocedural data-flow analysis (DFA) for caches based on the cache configuration. The simulator determines instruction categorizations, which describe the caching behavior of each instruction reference. The timing analyzer combines these

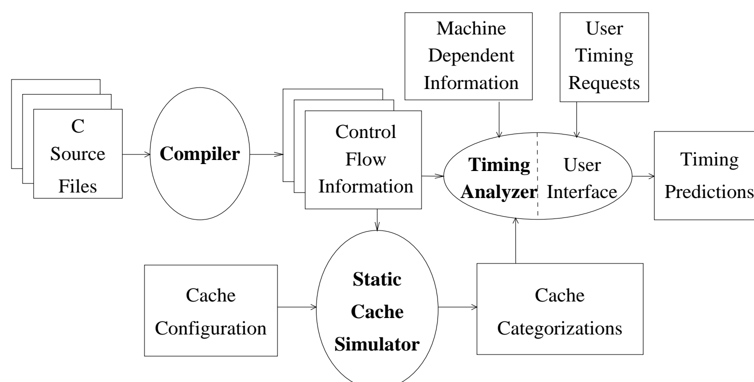


Figure 13. Framework for Timing Predictions

categorizations with the control-flow information to analyze the potential execution paths of the program. The analysis involves a cycle-level simulation of architectural characteristics such as pipelining. The timing analyzer also takes the temporal effect of the instruction categorizations from the static cache simulator into account to overlap memory stalls due to cache misses with pipeline stalls. Finally, WCET predictions are produced by the timing analyzer, either for one or more user selected segments of the program or for the entire program.

The utilization of an optimizing compiler allows us to extract information on the already optimized code, which corresponds to the code deployed for real-time systems. Other frameworks sometimes extract information about unoptimized code that does not correspond to the deployed systems. The framework does not depend on any particular compiler. For example, modifications to the Gnu Ada Translator / the Gnu C Compiler have been studied [21], which illustrates that the tools do not depend on a particular source language. Our framework could also perform timing predictions on object code or the executable if information currently emitted by the compiler was instead derived from object file analysis.

The timing analyzer needs information about the maximum number of iterations for each loop to perform path analysis. This information is provided by the compiler if a fixed loop bound is available. When the number of loop iterations is data dependent, the user is requested to provide a bound manually. The programs of the test set below did not require any user interactions, *i.e.*, the compiler was able to deduce the iteration bound for each loop. Only the path analysis phase of the timing analyzer requires knowledge of loop bounds. The remaining tools, including the static cache analyzer, have no such stipulations. The current implementation of the analysis tools has not yet been augmented to handle bounded recursion but modifications within the analysis methods have been identified [22]. A restriction applying to all tools is the absence of indirect function calls since they prevent the construction of a deterministic call graph where each node represents a function whose name is known statically.

Table 2. Worst-Case Execution Times

Program Name	Description	Observed Cycles	Estimated Cycles	Est/Obs Ratio	Naive Ratio
Des	encrypt and decrypt 64 bits	95,877	109,006	**1.14**	5.58
Matcnt	count+sum int matrix[100x100]	443,754	443,790	**1.00**	9.99
Matmult	multiply 2 to 1 int matrix[50x50]	1,430,538	1,430,538	**1.00**	10.00
Matsum	sum values of int matrix[100x100]	343,628	343,646	**1.00**	9.99
Sort	bubblesort of int array[100]	3,130,692	3,141,718	**1.00**	10.00
Stats	sum, mean, var. of 2 arrays[1000 doubles]	183,491	192,509	**1.05**	9.94
average		N/A	N/A	**1.03**	9.25

In the experiments, static cache simulation and timing analysis were performed for instruction caches for 1/2/4/8-way set-associative caches with 16/8/4/2 lines, respectively, and a line size of 16 bytes. Thus, each cache configuration has the equivalent storage capacity of 256 bytes, which was chosen to model a realistic ratio of program size and cache size (from 2:1 to 9:1). The estimated number of cycles for a program execution was derived from static cache simulation and timing analysis without program execution. This number is compared to the number of observed cycles obtained by a trace-driven cache simulation. In the latter case, the program was executed with its worst-case input data. Pipeline simulation of the timing analyzer was intentionally disabled to isolate the effects of caching. The overhead of a cache miss was assumed to be 10 cycles, a realistic value for contemporary architectures [14].

Table 2 shows the results of WCET prediction for a 4-way associative cache with 4 lines. The other cache configurations mentioned before yield similar results in terms of the ratios and are therefore omitted. The programs are described in column 2. The observed cycles during program execution (column 3) are slightly less than the number of cycles estimated by our tools (column 4). The ratio between estimated and observed cycles (column 5) shows that our method yields tight estimations, sometimes even exact ones. The naive ratio (column 6) simulates a disabled cache, *i.e.*, by assuming that all instruction cache references were misses, and dividing those cycles by the observed cycles. It shows that an overestimation of the WCET of 9.25 times on average for the naive cache is reduced to only a slight overestimation of 1.03 with our approach, *i.e.*, when caches are enabled and included in the WCET prediction. The results for some programs require further explanation.

The timing analysis overestimates program *Des* due to a data dependency that was also previously described. For the programs *Matcnt*, *Matsum* and *Stats*, the number of cycles was slightly overestimated. The programs *Matcnt* and *Matsum* contain conditional control flow and would require exhaustive analysis of all permutations of execution paths to yield more accurate results. Such an approach would result in exponential complexity. Instead, the timing analyzer approximates the execution times conservatively using a fix-point algorithm (see Figure 7). This trade-off between accuracy and feasible time complexity still results in relatively tight but not always precise estimations. The programs *Sort* and *Stats* suffer from

an overly conservative categorization due to a program line crossing a function boundary. Nonetheless, the conservative category results in safe estimates that remain very tight. Notice that the estimation of *Sort* was very tight despite the loop dependencies with a varying number of iterations. The inner loop in the function within *Sort* that sorted the values had a varying number of iterations that depends upon a counter of an outer loop. The tight result was obtained by averaging loop iterations (for details see [12]).

Figure 14 shows the average ratio between estimated and observed cycles for cache associativities between 1 and 8. The estimations remain tight for different levels of associativity. A more detailed analysis can be seen in Figure 15, representing the

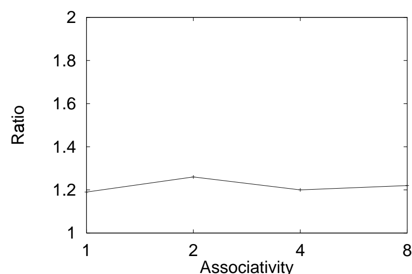


Figure 14. Ratio between Estimated and Observed Cycles

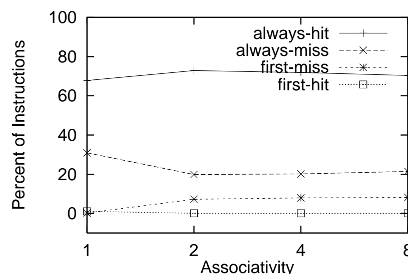


Figure 15. Distribution of Instruction Categories

distribution of the instruction categories, averaged over the test set. The distribution varies only insignificantly for different levels of associativity showing that our method yields tight results regardless of the associativity of caches.

Figure 16 displays the average time measured for static cache simulation on a lightly loaded SPARC 20 (via `gettimeofday`). It shows that the execution time increases with the level of cache associativity. The increase can be attributed to the overhead of operations on bit-vectors implementing the sets of the data-flow equations. The performance overhead for direct-mapped caches is extremely low (about 0.3 sec) and is still respectable (about 1.1 sec) for the largest associativity found in today's processors. Thus, static cache simulation is an adequate and efficient method to model caches for WCET predictions for contemporary architectures.

11. Future Work

We have proposed an extension to this framework of static cache simulation for bounding the WCET of multi-level caches [24] and this work is currently being implemented. We are also working on integrating timing analysis of both instruction caches (described here) and data caches (described in [32]) to obtain timing predictions for a complete machine. Actual machine measurements using a logic analyzer could then be used to gauge the effectiveness of the entire timing analysis environment.

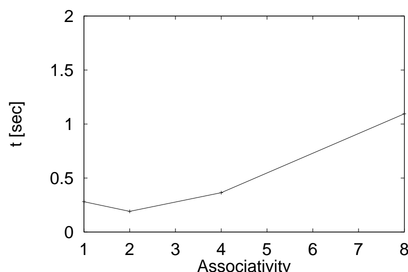


Figure 16. Performance Overhead of Static Cache Simulation

12. Related Work

Recently, a number of research groups have addressed various issues in the area of predicting the WCET of real-time programs. Conventional methods for static analysis have been extended from unoptimized programs on simple CISC processors [28, 26, 10, 27] to optimized programs on pipelined RISC processors [33, 19, 13], and from uncached architectures to instruction caches [4, 17, 15] and data caches [29, 16, 18].

The possibility of an extension of Park's timing schema for set-associative caches is briefly mentioned in [19].

Li *et al.* [18] have described a framework to integrate caching into their integer linear programming (ILP) approach to timing prediction. Their ILP approach has been extended by a set of finite-state automata, one for each cache line with conflicts, that simulate the behavior of set-associative instruction caches and is described by constraints placed at the reference points of program lines in the control flow. The ILP-solver would then take caching effects into account but the cache constraints increased the complexity (and the search space) of the ILP-problem. Their results only estimate the number of cache misses, which makes a comparison with our results difficult since we simulated both cache hits and misses. The ILP approach, when applied to micro-architecture modeling such as caches, does not scale well. For example, the response time for ILP changes from seconds for direct-mapped caches up to hours just for 2-way associative caches while our method exhibits response times of less than one second (direct-mapped) up to a few seconds reported in Figure 16 of this paper, even for 8-way associative caches. Thus, it seems questionable if the ILP approach combined with micro-architecture modeling is feasible when used every day in the software development cycle. It is not clear how their approach scales in general with changing associativity. However, their ILP approach does facilitate the integration of additional user-provided constraints into the analysis.

An alternate formalization [9] based on our instruction categories [22, 3, 23, 32] is presented for set-associative caches via abstract interpretation [2]. Alt *et al.* [2] originally used our notion of abstract cache states but their method distinguishes

a may-analysis (using set unions of cache states at joins in the control flow) and a must-analysis (using set intersections), whereas we infer the latter from the former. For this purpose, we employ additional data-flow information, such as linear cache states, dominator cache state and post-dominator sets. Both approaches distinguish categories at loop levels but in their more recent work [31] they refer to persistent accesses where we use the term first-miss and they dropped first-hits in favor of unclassified references, which may lead to slightly more pessimism. They also distinguish functions according to their invocations (called "virtual inlining" in their papers), handle recursion and perform bounded "virtual unrolling" for loops. Their cache models are more flexible than ours and the timing analysis framework has been extended to handle path analysis via ILP but work on integrating pipeline analysis into the framework is still in progress. A comparison of results is limited by the fact that they collect cache accesses from *dynamic* traces of actual executions (possibly *not* along the worst-case paths) where we rely strictly on *static* analysis techniques of each application being analyzed, which always yields safe estimates.

13. Conclusion

This paper contributes a comprehensive report on timing predictions for set-associative instruction caches. A set-theoretic formal definition through data-flow equations is given and the corresponding operational framework for simulating set-associative caches is described. The data-flow information is then used to determine the cache behavior of instruction references. This method of static cache simulation for set-associative caches is proved correct and shown to yield adequate results in experiments. The work enables tight predictions of the WCET by the timing analyzer, regardless of the degree of cache associativity. The cache simulation overhead scales well with increasing associativity. Overall, this work provides another step toward worst-case execution time prediction of contemporary architectures and its use in schedulability analysis for hard real-time systems.

Acknowledgments

The comments of the reviewers helped to improve the quality of this paper. David Whalley gave advice and support for the work on direct-mapped instruction caches. Robert Arnold provided the pipeline and path analysis tool for this research. Chris Healy extended and improved this tool to its current stage.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques, and Tools.* Addison-Wesley, 1986.
2. M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache behavior prediction by abstract interpretation. In *Static Analysis Symposium*, September 1996.
3. R. Arnold. Bounding instruction cache performance. Master's thesis, Dept. of CS, Florida State University, December 1996.

4. R. Arnold, F. Mueller, D. B. Whalley, and M. Harmon. Bounding worst-case instruction cache performance. In *IEEE Real-Time Systems Symposium*, pages 172–181, December 1994.
5. N. Audsley, A. Burns, R. Davis, K. Tindell, and A. J. Wellings. Fixed priority pre-emptive scheduling: An historical perspective. *J. of Real-Time Systems*, 8:173–198, 1995.
6. M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 329–338, June 1988.
7. Giorgio C. Buttazzo. *Hard Real-Time Computing Systems.* Kluwer, 1997.
8. UC Berkeley CS. CPU info center. http://infopad.eecs.berkeley.edu/CIC/summary/local, March 1999.
9. C. Ferdinand, F. Martin, and R. Wilhelm. Applying compiler techiniques to cache behavior prediction. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pages 37–46, June 1997.
10. M. Harmon, T. P. Baker, and D. B. Whalley. A retargetable technique for predicting execution time. In *IEEE Real-Time Systems Symposium*, pages 68–77, December 1992.
11. C. A. Healy, R. D. Arnold, F. Mueller, D. Whalley, and M. G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1), January 1999.
12. C. A. Healy, M . Sjödin, and D. B. Whalley. Bounding loop iterations for timing analysis. In *IEEE Real-Time Technology and Applications Symposium*, pages 12–21, June 1998.
13. C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *IEEE Real-Time Systems Symposium*, pages 288–297, December 1995.
14. J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 2nd edition, 1996.
15. Y. Hur, Y. H. Bae, S.-S. Lim, B.-D. Rhee, S. L. Min, C. Y. Park, M. Lee, H. Shin, and C. S. Kim. Worst case timing analysis of RISC processors: R3000/R3010 case study. In *IEEE Real-Time Systems Symposium*, pages 308–319, December 1995.
16. S. Kim, S. Min, and R. Ha. Efficient worst case timing analysis of data caching. In *IEEE Real-Time Technology and Applications Symposium*, June 1996.
17. Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *IEEE Real-Time Systems Symposium*, pages 298–397, December 1995.
18. Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *IEEE Real-Time Systems Symposium*, pages 254–263, December 1996.
19. S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, and C. S. Kim. An accurate worst case timing analysis for RISC processors. In *IEEE Real-Time Systems Symposium*, pages 97–108, December 1994.
20. C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. of the Association for Computing Machinery*, 20(1):46–61, January 1973.
21. D. Macos and F. Mueller. Integration Gnat/Gcc into a timing ananlysis environment. In *Work in Progress Session at EuroMicro Workshop on Real-Time Systems*, pages 15–18, June 1998.
22. F. Mueller. *Static Cache Simulation and its Applications.* PhD thesis, Dept. of CS, Florida State University, July 1994.
23. F. Mueller. Generalizing timing predictions to set-associative caches. In *EuroMicro Workshop on Real-Time Systems*, pages 64–71, June 1997.
24. F. Mueller. Timing predictions for multi-level caches. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pages 29–36, June 1997.
25. F. Mueller, D. B. Whalley, and M. Harmon. Predicting instruction cache behavior. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June 1994.
26. C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–61, March 1993.
27. P. Puschner. *Zeitanalyse von Echtzeitprogrammen.* PhD thesis, Dept. of CS, Technical University Vienna, December 1993.

28. P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2):159–176, September 1989.
29. J. Rawat. Static analysis of cache analysis for real-time programming. Master's thesis, Iowa State University, 1995.
30. J. Stankovic, M. Spuri, K. Ramamritham, and G. Buttazzo. *Deadline Scheduling for Real-Time Systems.* Kluwer, 1998.
31. H. Theiling and C. Ferdinand. Combining abstract interpretation and ilp for microarchitecture modelling and program path analysis. In *IEEE Real-Time Systems Symposium*, pages 144–153, December 1998.
32. R. White, F. Mueller, C. Healy, D. Whalley, and M. Harmon. Timing analysis for data caches and set-associative caches. In *IEEE Real-Time Technology and Applications Symposium*, pages 192–202, June 1997.
33. N. Zhang, A. Burns, and M. Nicholson. Pipelined processors and worst case execution times. *Real-Time Systems*, 5(4):319–343, October 1993.