Generalizing Timing Predictions to Set-Associative Caches

Frank Mueller, Humboldt-Universität zu Berlin, Institut f. Informatik, 10099 Berlin (Germany)
e-mail: mueller@informatik.hu-berlin.de phone: (+49) (30) 20181-276 fax:-280

Abstract

Recently, research on the static prediction of worst-case execution time (WCET) of programs has been extended from simple CISC to pipelined RISC processors, and from uncached architectures to direct-mapped instruction caches. This work goes one step further by introducing a framework to handle WCET prediction for set-associative caches*. Generalizing the work of static cache simulation of direct-mapped caches to set-associative caches, a formalization of the new method is given and the operational characteristics are presented and discussed by example. WCET predictions for several programs are presented by combining the static cache analysis for set-associative caches with a timing analysis tool. This approach has the advantage that cache configuration details are handled by static cache simulation but remain transparent to the timing analyzer. It is shown that static cache analysis for set-associative caches results in just as tight timing predictions as reported for direct-mapped caches. Overall, this work fills another gap between realistic WCET prediction of contemporary architectures and its use in schedulability analysis for hard real-time systems.*

1. Introduction

Hard real-time systems rely on the assumption that the deadlines of tasks can be met – otherwise the safety of the controlled system is jeopardized. Several scheduling paradigms have been developed to support the analysis of a task set and determine if a schedule is feasible, *e.g.,* rate-monotone analysis [10]. These scheduling paradigms rely on the assumption that the worst-case execution time (WCET) of hard real-time tasks be known *a-priori*. If the WCET of all tasks is known, it can be determined if a schedule is feasible, *i.e.,* if the tasks are guaranteed to meet their deadlines.

Determining the WCET is a prerequisite for off-line schedulability analysis, but most practitioners use *ad-hoc* methods to measure the execution time of a program with some worst-case input. However, such an approach may yield incorrect (overly optimistic) results in the context of modern processors with pipelines, caches, or even instruction-level parallelism. For example, while a longer execution path typically contributes to the WCET in uncached systems, a shorter path with frequent cache misses may result in longer execution in a cached system. An analytical approach is needed, supported by tools, to determine the WCET for contemporary architectures if real-time systems want to exploit the speed of such processors.

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 and data cache are kept separately. The level of associativity for such caches ranges between 1 and 8 (see Table 1) [4]. This work introduces a framework to handle WCET prediction for *set-associative caches*.

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

Table 1: Cache Associativity for various Processors

2. Related Work

The task of bounding the WCET of programs is (due to the undecidability of the halting problem) generally constraint by a set of assumptions about the use of programming language constructs and about the underlying operating system. For a static estimate of the WCET, an upper bound on the number of loop iterations has to be known, indirect calls should not be used, and memory should not be allocated dynamically. Often, recursive functions are also not allowed, although there exist outlines on treating bounded recursion similar to bounded loops [11].

Early work in the field of WCET prediction used a timing schema to propagate execution times of programming structures along the control-flow graph of functions and call graph of a program. Park analyzed programs at the source level (disregarding compiler optimizations) [12] while Harmon *et. al.* performed instruction-level timing augmented by the execution times of each instruction for a given architecture [5]. Recent advances in computer architecture forced researchers to extend these methods. Pipelined processors were handled by simulating the execution stages of instructions and overlapping the stages of adjacent instructions along execution paths [14, 9, 6].

To model the effect of cache memories, several approaches were taken. The first approach uses an extension of the timing schema by Park and updates caching information associated with paths during the traversal of the timing graph. Distinct cache states of multiple paths may have to be considered during the analysis before shorter paths are pruned until only the longest, worst-case path remains [9]. An an extension of the timing schema for set-associative caches is briefly mentioned but neither has it been formalized nor has an implementation or even results been reported. In another approach, integer linear programming (ILP) was used to describe constraints on the execution paths to derive the WCET from these constraints, first by Puschner and most recently by Li *et. al.* [13, 8]. Li *et. al.* enhanced this method 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. Since the ILP approach is subject to long response times (up to hours compared to up to seconds with our approach), it seems questionable if this approach is feasible when used every day in the software development cycle. It is not clear how their approach scales in general with changing associativity.

An alternate formalization based on our instruction categories [11] is presented for set-associative caches via abstract interpretation [2] but no results have been reported. They 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 imply the latter from the former. For this purpose, we employ additional data-flow information, such as linear cache states and post-dominator sets. Furthermore, our additional information is used to distinguish categories at loop levels, a distinction that they do not make. Thus, our approach should result in tighter bounds. Their approach is also likely to impose more time overhead for the analysis due to their abstract interpretation of a complex model *vs.* our direct implementation of the model in the static cache simulator.

3. Framework for Timing Prediction

The framework of WCET prediction discussed here includes a modified compiler and a set of tools to replace hand-calculated or externally timed estimations with analytically derived, reliable timings. The programmer does not need to know the worst-case input since the framework statically determines execution paths leading to WCET predictions by means of path analysis. Figure 1 gives an overview of the tools within the framework.

An optimizing Compiler accepts the source code of a program (currently for the language C). The compiler produces object code and separately emits control-flow information and the calling structure of functions. The static cache simulator uses the control-flow information and calling structure in conjunction with the cache configuration to produce instruction categorizations describing the caching behavior of each instruction. The timing analyzer combines these categorizations with the control-flow information to perform a path analysis of the program. This may include the simulation of architectural characteristics (*e.g.,* pipelining) but the caching behavior can be inferred from instruction categories, *i.e.* the process of cache simulation is entirely

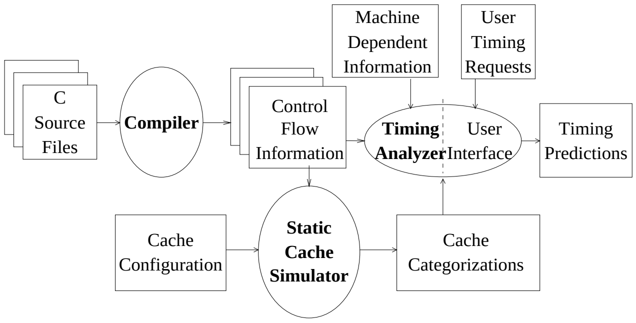

Figure 1: Framework for Timing Predictions

separated and transparent to the timing analyzer. The timing analyzer produces WCET predictions for portions of the program or the entire program, depending on user requests.

4. Simulation of Set-Associative Caches

Static cache simulation provides the means to predict the caching behavior of the instructions of a program/task. The predicted caching behavior is distinguished by the following categories:

Always-hit: The instruction results in a cache hit on each reference.

Always-miss: The instruction results in a cache miss on each reference.

First-hit: The instruction results in a cache hit on the first reference and a cache miss for any subsequent references.

First-miss: The instruction results in a cache miss on the first reference and a cache hit for any subsequent references.

A program may consist of a number of loops, possibly nested and distributed over several functions. For each loop level, an instruction receives a distinct categorization. The timing analyzer can then derive tight bounds of execution time by inspecting the categorizations for each loop level.

Since instruction categorizations have to be determined by inter-procedural analysis of the entire program, the call graph of the program has to be analyzed. The method of static cache analysis traces the origin of calls within the call graph by distinguishing *function instances.* Since instruction categorizations for a function are specified for each function instance, the timing analyzer can interpret different caching behaviors depending on the calling sequence to yield tighter WCET predictions. More detailed explanations of the operational characteristics of the timing analyzer, as well as examples, will be given later.

4.1. Abstract Cache State

The static cache simulator determines the categories of an instruction based on a novel view of cache memories, using a variation of iterative inter-procedural data-flow analysis (DFA). We first introduce the formal framework to reason about the caching behavior.

Definition 1 (Potentially Cached) *A program line l can potentially be 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 path.*

The process of determining if a line is potentially cached may be performed by a path traversal. However, the combinatorial problem of traversing every possible sequence of paths leads to an exponential explosion in the search space wrt. the branching factor, *i.e.* nodes with conditional branches that have two successors in the control flow. Static cache simulation counters this complexity problem via inter-procedural DFA, modified for caching purposes. 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 2 (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 are employed to support the operational framework of DFA simulating the cache invalidation protocol of set-associative caches.

4.2. Operational Framework for DFA

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. Such conflicting lines are subject to the replacement policy. For an n-way set-associative cache, the least-recently used (LRU) line is generally replaced. We assume the LRU policy for the remainder of the paper since it is the most-commonly used policy in practice. 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.

Algorithm 1 depicts the calculation of ACS for an n-way associative cache. The first path of function `main` 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` vectors): 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` vectors upon encountering conflicting lines within a single path.

This DFA requires a time overhead comparable to that of inter-procedural 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].

Algorithm 1 (Calc. of Abstract Cache States)
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 \=.

```
input(main, 1):= all invalid lines;
WHILE any change DO
    FOR each instance of a path P in the program DO
        FOR set := 1 TO n DO
            input(P, set):= φ;
            FOR each immed. predecessor Pred of P DO
                input(P, set) ∪= output(Pred, set);
        FOR set := 1 TO n DO
            diff(set) := φ;
        FOR set := 1 TO n DO
            output(P, set) := input(P, set);
            FOREACH line ∈ prog_lines(P) DO
                IF set = 1 THEN
                    output(P, set) ∪= {line};
                ELSE
                    output(P, set) \= {line};
                    IF map_into_same_line(diff(set - 1),
                        line) \ {line} = φ THEN
                        CONTINUE;
                    diff(set - 1) \= {line};
                    output(P, set) ∪= diff(set - 1);
                IF set = 1 OR diff(set - 1) ≠ φ THEN
                    FOREACH conf_line ∈ map_into_-
                        same_line(output(P, set) \ {line},
                        line) DO
                        IF set = 1 OR conf_line ∉
                            diff(set - 1) THEN
                            output(P, set) \= {conf_line};
                            diff(set) ∪= {conf_line};
```

4.3. 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 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 3 (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. 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. As a result, the LCS is an n-tuple of sets of program lines, assuming an n-way set associative cache.

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 4 (Post-dominator Set) *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 post dominators can be found elsewhere [1].

The instruction categories can now be defined with respect to the available data-flow information. Definition 5 (see next page) formalizes the worst-case instruction categories for each loop level. 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 set operations on 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: If the first instruction of the line has been categorized, then any other instruction of the same line is an always-hit (follows from spatial locality). Otherwise, there are enough associativity levels to cache (at most) $n-1$ conflicting lines, *i.e.*

- if the line is in the LCS,
- if the line is in some j-th ACS,
- if there is no conflicting line in the j-th ACS or there are enough associativity levels to hold conflicting lines in the ACS.

First-hit: If the instruction was a first-hit at the inner-next loop level or if the line was guaranteed to be cached upon entering the loop but may have been replaced in cache after iterating within the loop, *i.e.*

- if the instruction is first within the line,
- if the line is in the ACS,
- if the line is in the LCS,
- if the line is in the output ACS for all preheaders of the loop, [1]
- if the line is guaranteed to be executed when the loop is entered (post dominator),
- if there are more conflicting lines (within the loop) than available associativity levels,
- if there are enough associativity levels to hold conflicting lines (within the loop) in the output ACS of the preheaders,
- if there are enough associativity levels to hold conflicting lines (within the loop) in the output LCS of the preheaders.

First-miss: If the instruction was a first-miss at the inner-next loop level or if the line may not be in cache when the loop is entered but is guaranteed to be brought into cache after one loop iteration, *i.e.*

- if the instruction is first within the line,
- if the line is in the ACS,
- if there are more conflicting lines than available associativity levels,
- if there are enough associativity levels to hold conflicting lines (within the loop) in the ACS.

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

4.4. Example

The example in Figure 2 shows the assembly code 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 called from two different places within a loop in main. Assuming a two-way set-associative cache with four instructions per cache line and a total of two 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 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 inter-procedurally.

For instance, the first instruction of `value (a)` is a first-miss at the inner-most loop level (`value (a)`) and the next loop level (loop within `main`). The instruction is an always-miss at the outer-most level (function

[1] The common notion of "natural loops" defines a single loop header preceded by a single preheader outside the loop [1]. This work extends this notion to handle more general control flow with unstructured loops. Multiple loop headers occur only for unstructured loops, which are handled by the simulator. Multiple loop preheaders occur when the loop can be entered from more than one path outside the loop, which can occur even for natural loops.

Definition 5 (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 $s = \bigcup_{1 \le j \le n} s_j$.
- 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 $header(\lambda)$ be the set of header paths and $preheader(\lambda)$ 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 $linear = \bigcup_{1 \le j \le n} linear_j$.
- Let $postdom(p)$ be the set of self-reflexive post-dominating programming lines of path p.

Then,

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

`main`). The function instance `value (a)` is called in 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 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 block 4, which is the first reference to program line 4. In both cases, an always-miss is reported for the outer-most 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 5 and will be explained later.

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

Instruction 5 of `value (a)` 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 (a)` was determined to be an always-miss for the inner-most loop level and a first-hit for higher levels. Consider the outer-most level first: On the first call to `value (a)`, program line 1 is still in cache since the loop had just been entered from block 2 (containing this line) and block 3 also brought line 3 in cache, causing a hit for instruction 5. This is equivalent to a first-hit at the outer-most 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 blocks 7 and 3 are executed, respectively. This results in misses for instruction 5 of `value (a)` starting with the second iteration. Thus, the instruction is categorized as a first-hit at this level.

The inner-most loop level corresponds to the function level of `value (a)`, 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 (a)` 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 (b)`, on the other hand, is an always-hit since line 1 (and line 3) are still cached from the call to `value (a)`. 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 5.

Consider instruction 1 of `value (a)` again. Line 0 is not in the current LCS (see Figure 3). 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 inner-most 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 inner-most level. The same

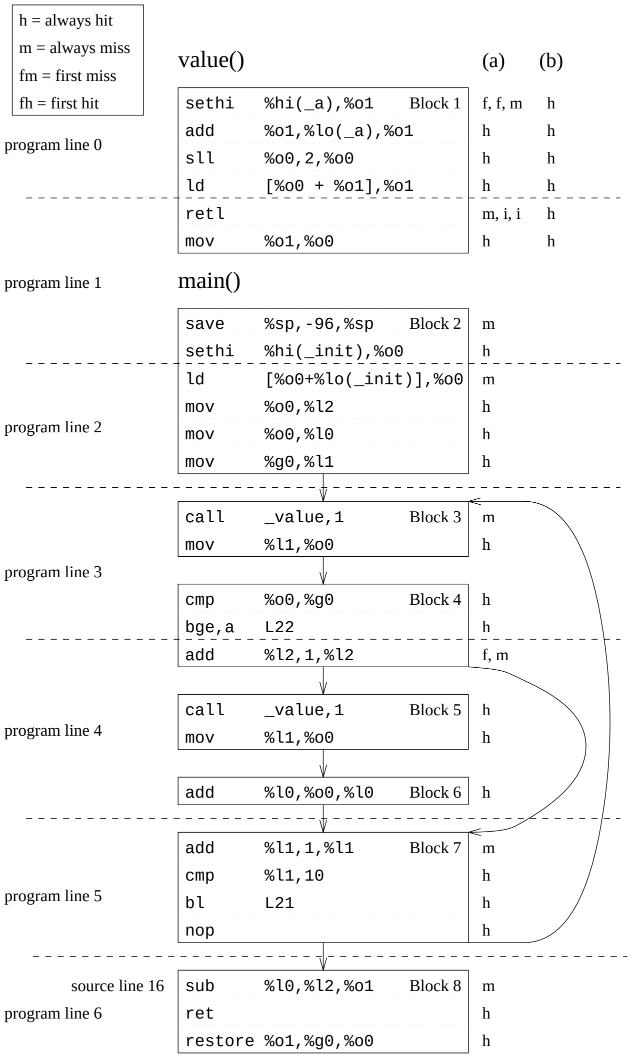

Figure 2: Program Structure and Instruction Categorizations

holds for the next loop level (the loop within `main`) since only lines 0 and 4 are in the intersection. At the outer-most level (function `main`), all three 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 (a)` 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 LCS, in the output ACS of block 2 (preheader), and in the post-dominator of 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 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.

```
                    cache line 0 1 0 1 0 1 0 1 0
                  program line I I 0 1 2 3 4 5 6
Abstract Cache State (ACS)
    value(a).1.in [0]=[         2 3 4     ]
    value(a).2.in [1]=[I   0 1         5  ]
        main.2.out[0]=[      1 2          ]
        main.2.out[1]=[I I                ]
Linear Cache State (LCS)
    value(a).1.in [0]=[         2 3       ]
    value(a).1.in [1]=[I     1            ]
Post-Dominator
        main.3.in    =[    0 1   3 4 5 6 ]
```

Figure 3: Data-Flow Sets for Selected Blocks

5. 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. Solemnly based on the instruction categorizations, the timing analyzer can derive the WCET by propagating timing predictions bottom-up within the timing tree. This interpretation of instruction categories shall be described in more detail.

The timing tree represents the calling structure and the loop structure of the entire program. As seen in the context of the static cache simulator, functions are distinguished by their calling paths into function instances. This allows a tighter prediction of the WCET due to the enhanced information about the calling context. 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 (outer-most 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 is encountered along a path, its WCET is already calculated and can simply be added to the WCET of the current path, sometimes with small adjustments. Adjustments are necessary for transitions from first-misses to first-misses and always-misses to first-hits between loop levels [3]. 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, most loops have one longest path. 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 scope of the WCET anal-

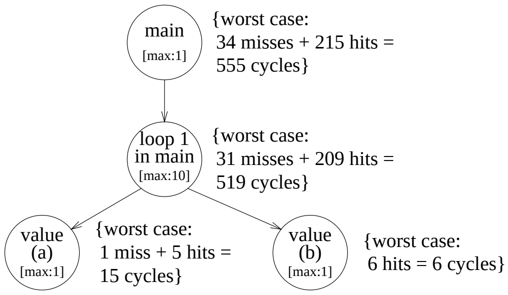

Figure 4: Timing Tree with WCET Prediction

ysis can such be limited to one loop level at a time, making timing analysis very efficient compared to an exhaustive analysis of all permutations of paths within a program. See [3] for a more detailed description of the the timing analyzer.

Consider the example from Figure 2 again. The timing analyzer predicts the WCET by traversing a timing tree, consisting of a node for each loop level (see Figure 4). 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 (a)` is given by a miss and 5 hits (either instruction 5 misses on the first iteration or instruction 1 misses on consecutive iterations). Assuming a miss penalty of 10 cycles, the predicted WCET for `value (a)` is 15 cycles. Since `value (b)` consists of 6 hits, 6 cycles are predicted. The loop's WCET in `main` is bounded by executing the longer path (blocks 5 and 6) during each iteration. There are 3 misses at this level: 2 always-misses within the loop and one between the first-hit and first-miss in `value (a)`. There are 20 hits: 9 always-hits in the loop, 10 always-hits in the instances of `value` and one more hit between the first-miss and first-hit in `value (a)`. Each of these hits and misses occur during each of the 10 loop iterations. In addition, there is a first-miss, counted as 9 hits and 1 miss. Thus, there is a total of 3*10+1=31 misses and 20*10+9=209 hits, *i.e.* 31*10+209=519 cycles.

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

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 cache simulator on the worst-case input data.

6. Measurements

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: The test programs were 2 to 9 times larger than the cache and included a data encryption program (`des`), matrix operations such as multiplication (`matmult`), summation (`matsum`) and counting of non-negative elements (`matcnt`), as well as the bubble-sort algorithm (`sort`) and a program calculating statistical functions of two arrays of numbers (`stats`). The estimated number of cycles for a program execution was derived from static cache simulation and timing analysis without program execution. Pipeline simulation of the timing analyzer was intentionally disabled to isolate the effects of caching. 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 it's worst-case input data. The miss penalty was assumed to be 10 cycles, a realistic value for contemporary architectures[7].

Program	Observed Cycles	Estimated Cycles	Ratio
Des	95877	109069	1.14
Matcnt	443754	443790	1.00
Matmult	1430538	1430538	1.00
Matsum	343628	343646	1.00
Sort	3130692	6249474	2.00
Stats	183491	192518	1.05
average	937996	1461505	1.20

Table 2: Worst-Case Times for a 256B, 4-way Set-Associative Cache

Table 2 shows the results of WCET prediction for a 4-way associative cache with 8 lines. The observed cycles during program execution (column 2) are slightly less than the number of cycles estimated by our tools (column 3). The ratio between estimated and observed cycles (column 4) shows that our method yields tight estimations, sometimes even exact ones. The results for some programs require further explanation. The program `sort` contains an inner loop whose termination depends on the iteration count of the outer loop. The static bound on the maximum iterations of the inner loop, however, is presented as a constant to the timing analyzer. Thus, the timing analyzer overestimates the number of cycles by a factor of 2 due to a lack of information. Tighter estimations would be possible by requesting a more detailed analysis of the loop structure and providing distinct bounds on the inner loop for each iteration of the outer loop. The program `des` has a similar data dependency preventing tighter estimates. However, these problems are not caused by the cache analysis approach.

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 the fix-point algorithm described earlier. This trade-off between accuracy and feasible time complexity still results in relatively tight but not always precise estimations. The program `stats` suffers from an overly pessimistic cat-

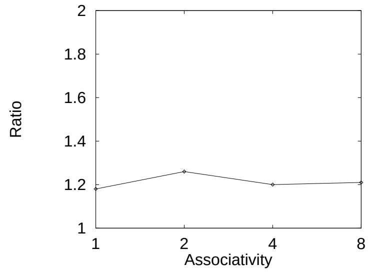

Figure 5: Ratio between Estimated and Observed Cycles

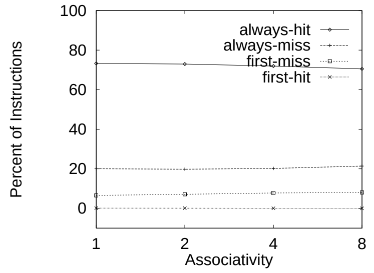

Figure 6: Distribution of Instruction Categories

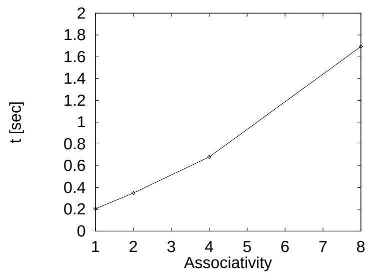

Figure 7: Performance Overhead of Static Cache Simulation

egorization due to a program line crossing a function boundary. However, the pessimistic category reported results in safe estimates that are still very tight.

Figure 5 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 6, representing the 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 7 displays the average time measured for static cache simulation on a lightly loaded SPARC 20 (via `gettimeofday`). It shows that the execution time increases linearly with the level of cache associativity. The increase can be attributed to the overhead of bit-vector operations implementing the data-flow equations. The performance overhead for direct-mapped caches is extremely low (about 200 ms) and is still respectable (about 1.7 sec) for the largest associativity found in today's processors. Thus, static cache simulation is an adequate method to model caches for WCET predictions for contemporary architectures efficiently.

7. Conclusion

This paper describes a formal method and the corresponding operational framework for simulating set-associative caches and yielding worst-case execution time (WCET) predictions of real-time programs. The method of static cache simulation is generalized to model set-associative caches by means of data-flow analysis. Instructions are categorized to describe their caching behavior for each loop level of the program. This information is used by a timing analyzer to bound the WCET. The isolated handling of caching issues in the static cache simulator allows caching aspects to remain transparent to the timing analyzer. The method of static cache simulation is shown to yield adequate results to enable tight predictions of the WCET by the timing analyzer, regardless of the degree of cache associativity. The performance of the static cache simulator is remarkably good for direct-mapped caches and increases linearly with the associativity to a moderate level even for the highest degree of associativity found in practice. By efficiently handling set-associative caches in the context of timing analysis, this work fills another gap between realistic WCET prediction of contemporary architectures and its use in schedulability analysis for hard real-time systems.

Acknowledgments

David Whalley and Christian Ferdinand pointed out mistakes in the manuscript. Chris Healy provided the timing analyzer including some modification and corrections for this work.

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, 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.

[4] UC Berkeley CS. CPU info center. http://infopad.eecs.berkeley.edu/CIC/summary/-local, March 1997.

[5] 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.

[6] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the timing analysis of pipelining and caching. In *IEEE Real-Time Systems Symposium*, pages 288–297, December 1995.

[7] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 2nd edition, 1996.

[8] 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*, December 1996.

[9] S.-S. Lim, Y. H. Bea, G. T. Jang, B.-D. Rhee, S. L. Min, Y. C. 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.

[10] C.L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.

[11] F. Mueller. *Static Cache Simulation and its Applications.* PhD thesis, Dept. of CS, Florida State University, July 1994.

[12] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–61, March 1993.

[13] P. Puschner. *Zeitanalyse von Echtzeitprogrammen.* PhD thesis, Dept. of CS, Technical University Vienna, December 1993.

[14] N. Zhang, A. Burns, and M. Nicholson. Pipelined processors and worst case execution times. *Real-Time Systems*, 5(4):319–343, October 1993.