Timing Analysis for Data Caches and Set-Associative Caches *

Randall T. White†, Frank Mueller‡, Christopher A. Healy†,
David B. Whalley†, and Marion G. Harmon §

Abstract

The contributions of this paper are twofold. First, an automatic tool-based approach is described to bound worst-case ***data cache*** *performance. The given approach works on fully optimized code, performs the analysis over the entire control flow of a program, detects and exploits both spatial and temporal locality within data references, produces results typically within a few seconds, and estimates, on average, 30% tighter WCET bounds than can be predicted without analyzing data cache behavior. Results obtained by running the system on representative programs are presented and indicate that timing analysis of data cache behavior can result in significantly tighter worst-case performance predictions. Second, a framework to bound worst-case instruction cache performance for* ***set-associative caches*** *is formally introduced and operationally described. 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 cache simulation overhead scales linearly with increasing associativity.*

1. Introduction

Real-time systems rely on the assumption that the worst-case execution time (WCET) of hard real-time tasks be known to ensure that deadlines of tasks can be met – otherwise the safety of the controlled system is jeopardized. Static analysis of program segments corresponding to tasks provides an analytical approach to determine the WCET for contemporary architectures. The complexity of modern processors requires a tool-based approach since *ad hoc* testing methods may not exhibit the worst-case behavior of the architecture. This paper presents a system of tools that perform timing prediction by statically analyzing optimized code without requiring interaction from the user.

The work presented here addresses the bounding of WCET for data caches and set-associative caches. Thus, it presents an approach to include common features of contemporary architectures within static prediction of WCET. Overall, this work fills another gap between realistic WCET prediction of contemporary architectures and its use in schedulability analysis for hard real-time systems.

The framework of WCET prediction uses a set of tools as depicted in Figure 1. An optimizing compiler has been modified to emit control-flow information, data information, and the calling structure of functions in addition to regular object code generation. A static cache simulator uses the control-flow information and calling structure in conjunction with the cache configuration to produce instruction and data categorizations, which describe the caching behavior of each instruction and data reference, respectively. The timing analyzer uses these categorizations and the control-flow information to perform path analysis of the program. The timing analyzer produces WCET predictions for portions of the program or the entire program, depending on user requests.

2. Related Work

In the past few years, research in the area of predicting the WCET of programs has intensified. Conventional methods for static analysis have been extended from unoptimized programs on simple CISC processors to optimized programs on pipelined RISC processors [12, 6], and from uncached architectures to instruction caches [2, 10, 8] and data caches [9, 11].

Kim *et al.* [9] have recently published work about bounding data cache performance for calculated references, which they refer to as occurring from dynamic load and store instructions. Their approach uses a version of the pigeonhole principle. For each loop they determine the maximum number of references from each dynamic load/store instruction. They also determine the maximum number of distinct locations in memory referenced by these instructions. The difference between these two values is the number of data cache hits for the loop given that there are no conflicting references. This technique efficiently detects temporal locality within loops when all of the data references within a loop fit into cache and the size of each data reference is the same size as a cache line. Their technique at this time does not detect any spatial locality (*i.e.*, when the line size is greater than the size of each data reference and the elements are accessed contiguously) and detects no temporal locality across different loop nests. Furthermore, they cannot currently deal with compiler optimizations that alter the correspondence of assembly instructions to source code. Such compiler optimizations can make calculating ranges of relative addresses significantly more challenging.

*This work was supported in part by the Office of Naval Research under contract number N00014-94-1-0006.

†Florida State University, Computer Science Department, Tallahassee, FL 32306-4019, phone: (904) 644-3506, fax:-0058, e-mail: [rwhite,healy,whalley]@cs.fsu.edu

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

§Florida A&M University, Computer & Information Systems Department, Tallahassee, FL 32307-3101, phone: (904) 599-3042, fax: -3221, e-mail: harmon@cis.famu.edu

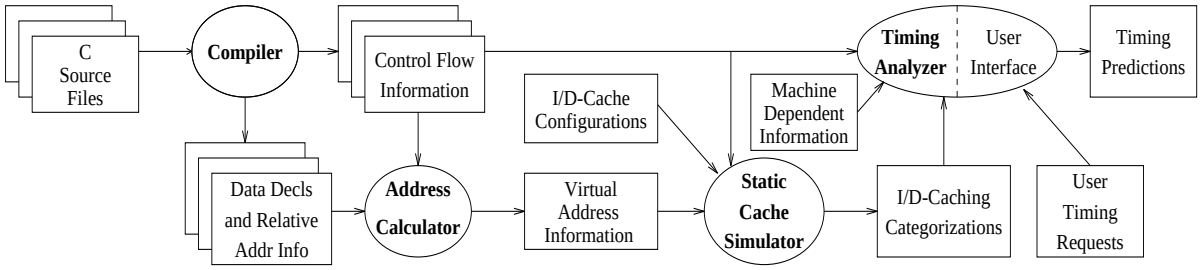


Figure 1: Framework for Timing Predictions

Li *et al.* [11] have described a framework to integrate data caching into their integer linear programming (ILP) approach to timing prediction. Their implementation performs data-flow analysis to find conflicting blocks. However, their linear constraints describing the range of addresses of each data reference currently have to be calculated by hand. They also require a separate constraint for *every* element of a calculated reference causing scalability problems for large arrays. No WCET results on data caches are reported. The ILP approach facilitates integrating additional user-provided constraints into the analysis.

The possibility of an extension of Park's timing schema for set-associative caches is briefly mentioned in [12] However, it has not been formalized nor has an implementation been described or any results been reported. The integer linear programming (ILP) approach [11] has been extended to support timing predictions for set-associative instruction caches. An automaton describes transitions between cache states for each set of conflicting blocks and additional cache constraints describe the problem on the ILP level. The results only estimate the number of cache misses, which makes a comparison with our results difficult since we predicted both cache hits and misses. Their results indicate a much higher overhead of the ILP approach (up to hours) compared to our methods (up to seconds). It is not clear how their approach scales in general with changing associativity.

3. Data Caches

Obtaining tight WCETs in the presence of data caches is quite challenging. Unlike instruction caching, many of the addresses of data references can change during the execution of a program. A reference to an item within an activation record could have different addresses depending on the sequence of calls associated with the invocation of the function. Many data references, such as indexing into an array, are dynamically calculated and can vary each time the data reference occurs. Pointer variables in languages like C may be assigned addresses of different variables or an address that is dynamically calculated from the heap.

Initially, it may appear that obtaining a reasonable bound on worst-case data cache performance is just not feasible. However, this problem is far from hopeless since the addresses for many data references can be statically calculated. Static or global scalar data references do retain the same addresses throughout the execution of a program. Run-time stack scalar data references can often be statically determined as a set of addresses depending upon the sequence of calls associated with an invocation of a function. The pattern of addresses associated with many calculated references, *e.g.* array indexing, can also often be resolved statically.

The prediction of the WCET for programs with data caches is achieved by automatically analyzing the range of addresses of data references, deriving relative and then virtual addresses from these ranges, and categorizing data references according to their cache behavior. The data cache behavior is then included in the pipeline analysis to yield worst-case execution time predictions of program segments.

3.1. Calculation of Relative Addresses

The *vpo* compiler [3] attempts to calculate relative addresses for each data reference associated with load and store instructions after compiler optimizations have been performed (see Figure 1), which may not be trivial, especially for non-scalar data references. Compiler optimizations can move instructions between basic blocks and outside of loops so that expansion of registers used in the address calculation becomes more difficult. Our approach calculates relative data address information automatically after all compiler optimizations have been performed.

First, the compiler determines for each loop the set of its induction variables, their initial values and strides, and the loop-invariant registers. Next, expansion of actual parameter information is performed in order to be able to resolve any possible address parameters later. Then, expansion of addresses used in loads and stores is performed. Expansion is accomplished by examining each preceding instruction represented as an RTL (register transfer list) and replacing registers used as source values in the address with the source of the RTL setting that register. Induction variables associated with a loop are not expanded. Loop invariant values are expanded by proceeding to the end of the preheader block of that loop. Expansion of the addresses of scalar references to the run-time stack (e.g. local variables) is trivial. Expansion of references to static data (e.g. global variables) often requires expanding loop invariant registers since these addresses are constructed with instructions that may be moved out of a loop. Although, there is not room in this paper to give examples of calculating relative addresses of data memory references, we did find many instances where traditional techniques for calculating ranges of relative addresses were not adequate due to interference from various compiler optimizations. For more details on statically determining address information

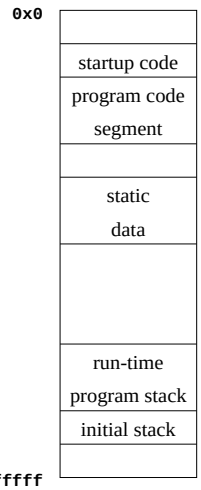


Figure 2: Virtual Address Space (SunOS)

```
WHILE any change DO
  FOR each basic block instance B DO
    IF B == top THEN
      input_state(B) = calc_input_state(B) = all invalid lines
    ELSE
      input_state(B) = calc_input_state(B) = NULL
    FOR each immed pred P of B DO
      input_state(B) += output_state(P)
      calc_input_state(B) += output_state(P) + calc_output_state(P)
      IF P is in another loop THEN
        input_state(B) += calc_output_state(P) + data_lines(remaining in that loop)
    output_state(B) = input_state(B)
    FOR each data reference D in B DO
      IF D is scalar reference THEN
        output_state(B) += data_line(D)
        output_state(B) -= data_lines(D conflicts with)
        calc_output_state(B) += data_line(D)
        calc_output_state(B) -= data_lines(conflicts with)
      ELSE
        output_state(B) -= data_lines(D could conflict with)
        calc_output_state(B) += data_lines(D could access)
        calc_output_state(B) -= data_lines(D could conflict with)
```

Figure 3: Algorithm to Calculate Data Cache States

from fully optimized code see [16].

3.2. Calculation of Virtual Addresses

Calculating addresses that are relative to the beginning of a global variable or an activation record is accomplished within the compiler since much of the data flow information required for this analysis is readily available due to its use in compiler optimizations. However, calculating virtual addresses cannot be done in the compiler since the analysis of the call graph and data declarations across multiple files is required. Thus, an address calculator (see Figure 1) uses the relative address information in conjunction with control-flow information to obtain virtual addresses.

Figure 2 shows the general organization of the virtual address space of a process executing under SunOS. There is some startup code preceding the instructions associated with the compiled program. Following the program code segment is the static data, which is aligned on a page boundary. The run-time stack starts at high addresses and grows toward low addresses. Part of the memory between the run-time stack and the static data is the heap, which is not depicted in the figure since addresses in the heap could not be calculated statically by our environment.

Static data consists of global variables, static variables, and non-scalar constants (e.g. strings). In general, the Unix linker (*ld*) places the static data in the same order that the declarations appeared within an assembly file. Also, static data within one file will precede static data in another file specified later in the list of files to be linked. (There are some exceptions to these rules depending upon how such data is statically initialized.) In addition, padding between variables sometimes occurs. For instance, variables declared as `int` and `double` on the SPARC are aligned on word and double-word boundaries, respectively. In addition, the first static or global variable declared in each of the source files comprising the program is aligned on a double-word boundary.

Run-time stack data includes temporaries and local variables not allocated to registers. The address of the activation record for a function can vary depending upon the actual sequence of calls associated with its activation. The virtual address of an activation record containing a local variable is determined as the sum of the sizes of the activation records associated with the sequence of calls along with the initial run-time stack address. The address calculator (along with the static simulator and timing analyzer) distinguishes between different function instances and evaluates each instance separately. Once the static data names and activation records of functions are associated with virtual addresses, the relative address ranges can be converted into virtual address ranges.

Only virtual addresses have been calculated so far. There is no guarantee that a virtual address will be the same as the actual physical address, which is used to access cache memory on most machines. In this paper we assume that the system page size is an integer multiple of the data cache size, which is often the case. For instance, the MicroSPARC I has a 4KB page size and a 2KB data cache. Thus, both a virtual and corresponding physical address would have the same relative offset within a page and would map to the same line within the data cache.

3.3. Static Simulation to Produce Data Reference Categorizations

The method of static cache simulation is used to statically categorize the caching behavior of each data reference in a program for a specified cache configuration (see Figure 1). A program control-flow graph is constructed that includes the control flow within each function and a function instance graph, which uniquely identifies each function instance by the sequence of call sites required for its invocation. This program control-flow graph is analyzed to determine the possible data lines that can be in the data cache at the entry and exit of each basic block within the program [13].

The iterative algorithm used for static instruction cache simulation [2, 13] is not sufficient for static data

(a) Detecting Spacial Locality

```
int a[100][100];                          int a[100][100];
main() { /* row order sum */              main() { /* column order sum */
   int i, j, sum = 0;                        int i, j, sum = 0;
   for (i = 0; i < 100; i++)                 for (j = 0; j < 100; j++)
      for (j = 0; j < 100; j++)                 for (i = 0; i < 100; i++)
         sum += a[i][j];                           sum += a[i][j];
}                                         }
```

```
row order:  c 25 2500  from [m h h h m h h h m h h h ...  m h h h]
col order:  m          from [m m m m m m m m m m m m ...  m m m m]
```

(b) Detecting Temporal Locality across and within Loops

```
   int i, j, sum = 0, same = 0, a[50], b[50];
   ...
   for (i = 0; i < 50; i++)
      sum += a[i];                  /* a[i] is ref 1 */
   for (i = 0; i < 50; i++)
      for (j = 0; j < 50; j++)
         if (a[i] == b[j])          /* a[i] is ref 2  and  b[i] is ref 3 */
            same++;
```

```
ref 1:  c 13        from [m h m h h h m h h h m h h h ...  m h h h]
ref 2:  h           from [h h ...  h h] due to temporal locality across loops.
ref 3:  c 13 13     from [m h h m h h h ...  m h] on first execution of inner loop,
                      and [h h h h ...  h] on all successive executions of it.
```

Figure 4: Examples for Spacial and Temporal Locality

cache simulation. The problem is that the calculated references can access a range of possible addresses. At the point that the data access occurs, the data lines associated with these addresses may or may not be brought in cache, depending upon how many iterations of the loop has been performed at that point. To deal with this problem a new state was created to indicate whether or not a particular data line could potentially be in the data cache due to calculated references. When an immediate predecessor block is in a different loop (the transition from the predecessor block to the current block exits a loop), the data lines associated with calculated references in that loop that are guaranteed to still be in cache are unioned into the input cache state of that block. The iterative algorithm in Figure 3 is used to calculate the input and output cache states for each basic block in the program control flow.

Once these cache state vectors have been produced, they are used to determine whether or not each of the memory references within the bounded virtual address range associated with a data reference will be in cache. The static cache simulator needs to produce a categorization of each data reference in the program. The four worst-case categories of caching behavior used in the past for static instruction cache simulation were as follows. (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. These categorizations are still used for scalar data references.

To obtain the most accuracy, a worst-case categorization of a calculated data reference for each iteration of a loop could be determined. For example, some categorizations for a data reference in a loop with 20 iterations might be as follows:

`m h h h m h h h m h h h m h h h m h h h`

With such detailed information the timing analyzer could then accurately determine the worst-case path on each iteration of the loop. However, consider a loop with 100,000 iterations. Such an approach would be very inefficient in space (storing all of the categorizations) and time (analyzing each loop iteration separately). The authors decided to use a new categorization called **Calculated (c)** that would also indicate the maximum number of data cache misses that could occur at each loop level in which the data reference is nested. The previous data reference categorization string would be represented as follows (since there is only one loop level involved): `c 5`

The order of access and the cache state vectors are used to detect cache hits within calculated references due to **spatial locality**. Consider the two code segments in Figure 4(a) that sum the elements of a two dimensional array. The two code segments are equivalent, except that the left code segment accesses the array in row order and the right code segment uses column order (*i.e.*, the `for` statements are reversed). Assume that the scalar variables (`i`, `j`, `sum`, and `same`) are allocated to registers. Also, assume the size of the direct-mapped data cache is 256 bytes with 16 cache lines containing 16 bytes each. Thus, a single row of the array `a` requiring 400 bytes cannot fit into cache. The static cache simulator was able to detect that the load of the array element in the left code segment had at most one miss for each of the array elements that are part of the same data line. This was accomplished by inspecting the order in which the array was accessed and detecting that no conflicting lines were accessed in these loops. The categorizations for the load data reference in the two segments are given in the same figure. Note in this case that the array happens to be aligned

on a line boundary. The specification of a single categorization for a calculated reference is accomplished in two steps for each loop level after the cache states are calculated. First, the number of references (iterations) performed in the loop is retrieved. Next, the maximum number of misses that could occur for this reference in the loop is determined. For instance, at most 25 misses will occur in the innermost loop for the left code segment. The static cache simulator determined that all of the loads for the right code segment would result in cache misses. Its data caching behavior can simply be viewed as an always miss. Thus, the range of 10,000 different addresses referenced by the load are collapsed into a single categorization of `c 25 2500` (calculated reference with 25 misses at the innermost level and 2500 misses at the outer level) for the left code segment and an `m` (always miss) for the right code segment.

Likewise, cache hits from calculated references due to **temporal locality** both across and within loops are also detected. Consider the code segment in Figure 4(b). Assume a cache configuration with 32 16-byte lines (512 byte cache) so that both arrays `a` and `b` requiring 400 bytes total (200 each) fit into cache. Also assume the scalar variables are allocated to registers. The accesses to the elements of array `a` after the first loop were categorized as an `h` (always hit) by the static simulator since all of the data lines associated with array `a` will be in the cache state once the first loop is exited. This shows the detection of temporal locality *across* loops. After the first complete execution of the inner loop, all the elements of `b` will be in cache, so then all references to it on the remaining executions of the inner loop are also categorized as hits. Thus, the categorization of `c 13 13` is given. Relative to the innermost loop, 13 misses are due to bringing `b` into cache during the first complete execution of the inner loop. There are also only 13 misses relative to the outermost loop since `b` will be completely in cache on each iteration after the first. Thus, temporal locality is also detected *within* loops.

The current implementation of the static data cache simulator (and timing analyzer) imposes some restrictions. First, only direct-mapped data caches are supported. Obtaining categorizations for set-associative data caches can be accomplished in a manner similar to that for instruction caches described in the next section. Second, recursive calls are not allowed since it would complicate the generation of unique function instances. Third, indirect calls are not allowed since an explicit call graph must be generated statically.

3.4. Timing Analysis

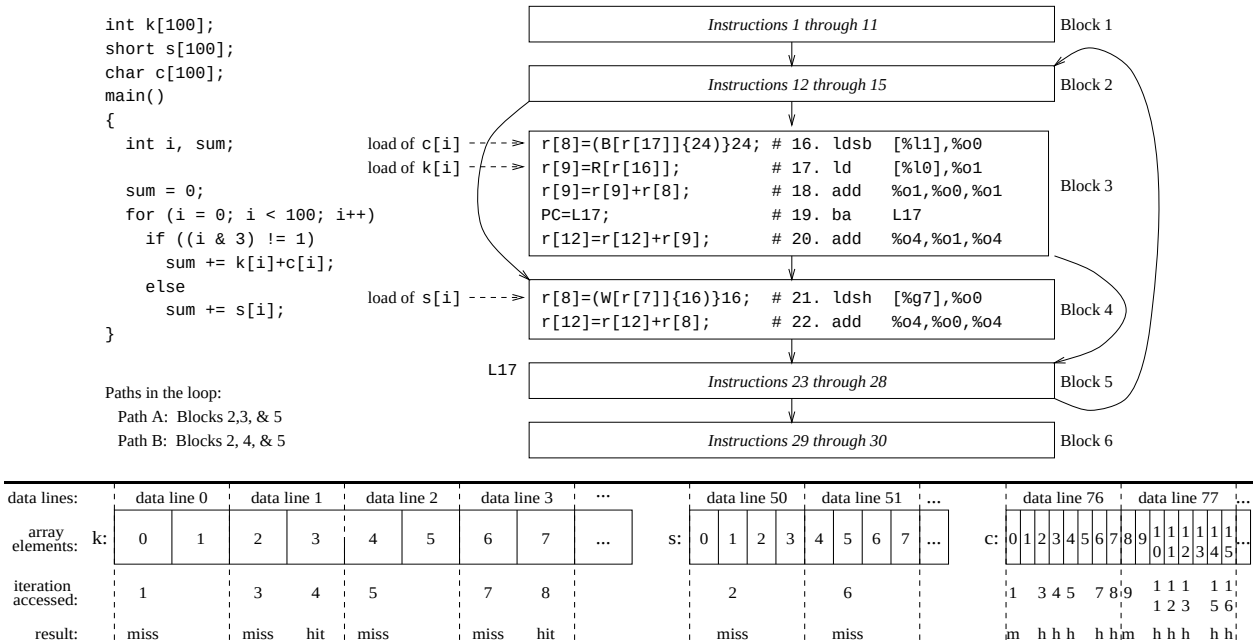
The timing analyzer (see Figure 1) utilizes pipeline path analysis to estimate the WCET of a sequence of instructions representing paths through loops or functions. Pipeline information about each instruction type is obtained from the machine-dependent data file. Information about the specific instructions in a path is obtained from the control-flow information files. As each instruction is added separately to the pipeline state information, the timing analyzer uses the data caching categorizations to determine whether the MEM (data memory access) stage should be treated as a cache hit or a miss.

```
total_cycles = curr_iter = 0.
pipeline_info = first_misses_encountered
    = first_hits_encountered = NULL.
WHILE curr_iter != n - 1 DO
    Find the longest continue path.
    first_misses_encountered += first misses that
        were misses in this path.
    first_hits_encountered += first hits that
        were hits in this path.
    IF first miss or first hit encountered in this path THEN
        curr_iter += 1.
        Subtract 1 from the remaining misses of each
            calculated reference in this path.
        Concatenate pipeline_info with the union of the
            info for all paths.
        total_cycles += additional cycles required by union.
    ELSE IF a calculated reference was encountered
            in this path as a miss THEN
        min_misses = the minimum of the number
            of remaining misses of each calculated
            reference in this path that is nonzero.
        min_misses = min(min_misses, n - 1 - curr_iter).
        curr_iter += min_misses.
        Subtract min_misses from the remaining
            misses of each calc ref in this path
        Concatenate pipeline_info with the union of the
            info for all paths min_misses times.
        total_cycles += min_misses
            * (additional cycles required by union).
    ELSE
        break.
Concatenate pipeline_info with the union of the pipeline info
    for all paths (n - 1 - curr_iter) times.
total_cycles += (n - 1 - curr_iter)
    * (additional cycles required by union).
FOR each set of exit paths that have a transition to a
    unique exit block DO
    Find the longest exit path in the set.
    first_misses_encountered +=
        first misses that were misses in this path.
    first_hits_encountered +=
        first hits that were hits in this path.
    Concatenate pipeline_info with the union of the info
        for all exit paths in the set.
    total_cycles += additional cycles required by exit union.
    Store this information with the exit block for the loop.
```

Figure 5: Worst-Case Loop Analysis Algorithm

The worst-case loop analysis algorithm was modified to appropriately handle calculated data reference categorizations. The timing analyzer will conservatively assume that each of the misses for the current loop level of a calculated reference has to occur before any of its hits at that level. In addition, the timing analyzer is unable to assume that the penalty for these misses will overlap with other long running instructions since the analyzer may not evaluate these misses in the exact iterations in which they occur. Thus, each calculated reference miss is always viewed as a hit within the pipeline path analysis and the maximum number of cycles associated with a data cache miss penalty is added to the total time of the path. This strategy permits an efficient loop analysis algorithm with some potentially small overestimations when a data cache miss penalty could be overlapped with other stalls.

The worst-case loop analysis algorithm is given in Figure 5. The additions to the previously published



Pipeline Diagram for Path A: Instructions 12-20 and 23-28 (blocks 2,3,5)

stage	cycle 1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
IF	12	13	14	15	16	17	18	19	19	20	23	24	25	26	27	28				
ID		12	13	14	15	16	17	18	18	19	20	23	24	25	26	27	28			
EX			12	13		15	16	17		18		20	23	24	25	26		28		
MEM				12	13		15	16	17		18		20	23	24	25	26		28	
WB					12	13		15	16	17		18		20	23	24	25	26		28

Pipeline Diagram for Path B: Instructions 12-15 and 21-28 (blocks 2,4,5)

stage	cycle 1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
IF	12	13	14	15	21	22	23	23	24	25	26	27	28				
ID		12	13	14	15	21	22	22	23	24	25	26	27	28			
EX			12	13		15	21		22	23	24	25	26		28		
MEM				12	13		15	21		22	23	24	25	26		28	
WB					12	13		15	21		22	23	24	25	26		28

Figure 6: Example to Illustrate Worst-Case Loop Analysis Algorithm

algorithm [6] to handle calculated references are shown in boldface. Let n be the maximum number of iterations associated with a loop. The WHILE loop terminates when the number of processed iterations reaches n - 1 or no more first misses, first hits, or calculated references are encountered as misses, hits, and misses, respectively. This WHILE loop will iterate no more than the minimum of $(n - 1)$ or $(p + r)$ times, where p is the number of paths and r is the number of calculated reference load instructions in the loop.

The algorithm attempts to select the longest path for each loop iteration. In order to demonstrate the correctness of the algorithm, one must show that no other path for a given iteration of the loop will produce a longer time than that calculated by the algorithm. Since the pipeline effects of each of the paths are unioned, it only remains to be shown that the caching effects are treated properly. All categorizations are treated identically on repeated references, except for first misses, first hits, and calculated references. Assuming that the data references have been categorized correctly for each loop and the pipeline analysis was correct, it remains to be shown that first misses, first hits, and calculated references are interpreted appropriately for each loop iteration. A correctness argument about the interpretation of first hits and first misses is given in [2].

The WHILE loop will subtract one from each calculated reference miss count for the current loop in the longest path chosen on each iteration whenever there are first misses or first hits encountered as misses or hits, respectively. Once no such first misses and first hits are encountered in the longest path, the same path will remain the longest path as long as its set of calculated references that were encountered as misses continue to be encountered as misses since the caching behavior of all of the references will be treated the same. Thus, the pipeline effects of this longest path are efficiently replicated for the number of iterations associated with the minimum number of remaining misses of the calculated references that are nonzero within the longest path. After the WHILE loop, all of the first misses, first hits, and calculated references in the longest path will be encountered as hits, misses, and hits, respectively. The unioned pipeline effects after the WHILE loop will not change since the caching behavior of the references will be treated the same. Thus, the pipeline effects of this path are efficiently replicated for all but one of the remaining iterations. The last iteration of the loop is treated separately since the

step	starting iteration	longest path cycles	min_misses	iterations handled	additional cycles	total cycles
1	1	20+18=38	min(13,50)=13	13	20+((20-4)*12)+(18*13)=446	446
2	14	20+9=29	min(37)=37	37	((20-4)*37)+(9*37)=925	1371
3	51	17+9=26	min(25)=25	25	((17-4)*25)+(9*25)=550	1921
4	76	20+0=20	N/A	24	(20-4)*24=384	2305
5	100	20+0=20	N/A	1	20-4=16	2321

Table 1: Timing Analysis Steps for the loop in Figure 6

longest exit path may be shorter than the longest continue path.

An example is given in Figure 6 to illustrate the algorithm. The `if` statement condition was contrived to force the worst-case paths to be taken when executed. Assume a data cache line size of 8 bytes and enough lines to hold all three arrays in cache. The figure also shows the iterations when each element of each of the three arrays will be referenced and whether or not each of these references will be a hit or a miss. Two different paths can be taken through the loop on each iteration as shown in the integer pipeline diagram of Figure 6. Note that the pipeline diagrams reflect that the loads of the array elements were found in cache. The miss penalty from calculated reference misses is simply added to the total cycles of the path and is not directly reflected in the pipeline information since these misses may not occur in the same exact iterations as assumed by the timing analyzer.

Table 1 shows the steps the timing analyzer uses from the algorithm given in Figure 5 to estimate the WCET for the loop in the example shown in Figure 6. The longest path detected in the first step is Path A, which contains references to `k[i]` and `c[i]`. The pipeline time required 20 cycles and the misses for the two calculated references (`k[i]` and `c[i]`) required 18 cycles. Note that each miss penalty was assumed to require 9 cycles. Since there were no first misses, the timing analyzer determines that the minimum number of remaining misses from the two calculated references is 13. Thus, the path is replicated an additional 12 times. The overlap between iterations is determined to be 4 cycles. Note that 4 is not subtracted from the first iteration since any overlap for it would be calculated when determining the worst-case execution time of the path through the `main` function. The total time for the first 13 iterations will be 446. The longest path detected in step 2 is also Path A. But this time all references to `c[i]` are hits. There are 37 remaining misses to `k[i]`. The total time for iterations 14 through 50 is 925 cycles. The longest path detected in step 3 is Path B, which has 25 remaining misses to `s[i]`. This results in 550 additional cycles for iterations 51 through 75. After step 3 the worst-case loop analysis has exited the WHILE loop in the algorithm. Step 4 calculates 384 cycles for the next 24 iterations (76-99). Step 5 calculates the last iteration to require 16 cycles. The timing analyzer calculates the last iteration separately since the longest exit path may be shorter than other paths in the loop. The total number of cycles calculated by the timing analyzer for this example was identical to the number obtained by execution simulation.

A timing analysis tree is constructed to predict the worst-case performance. Each node of the tree represents either a loop or a function in the function instance graph. The nodes representing the outer level of function instances are treated as loops that will iterate only once. The worst-case time for a node is not calculated until the time for all of its immediate child nodes are known. For instance, consider the example shown in Figure 6 and Table 1. The timing analyzer would calculate the worst-case time for the loop and use this information to next calculate the time for the path in `main` that contains the loop (block 1, loop, block 6). The construction and processing of the timing analysis tree occurs in a similar manner as described in [2, 6].

3.5. Results

Measurements were obtained on code generated for the SPARC architecture by the *vpo* optimizing compiler [3]. The machine-dependent information contained the pipeline characteristics of the MicroSPARC I processor [15]. A direct-mapped data cache containing 16 lines of 32 bytes for a total of 512 bytes was used. The MicroSPARC I uses write-through/no-allocate data caching [15]. While the static simulator was able to categorize store data references, these categorizations were ignored by the timing analyzer since stores always accessed memory and a hit or miss associated with a store data reference had the same effect on performance. Instruction fetches were assumed to be all hits in order to isolate the effects of data caching from instruction caching.

Table 2 shows the test programs used to assess the timing analyzer's effectiveness of bounding worst-case data cache performance. Note that these programs were restricted to specific classes of data references that did not include any dynamic allocation from the heap. Two versions were used for each of the first five test programs. The **a** version had the same size arrays that were used in previous studies [2, 6]. The **b** version of each program used smaller arrays that would totally fit into a 512 byte cache. The number of bytes reported in the table is the total number of bytes of the variables in the program. Note that some of these bytes will be in the static data area while others will be in the runtime stack. The amount of data is not changed for the program *Des* since its encryption algorithm is based on using large static arrays with preinitialized values.

Table 2 also depicts the dynamic results from executing the test programs. The *hit ratios* were obtained from the data cache execution simulation. Only *Sort* had very high data cache hit ratios due to many repeated references to the same array elements. The *observed cycles* were obtained using an execution simulator, modified from [5], to simulate data cache and pipeline affects and count the number of cycles. The *estimated cycles* were obtained from the timing analyzer discussed in Section 3.4. The *estimated ratio* is the quotient of these two values. The *naive ratio* was

Name	Num Bytes	Description or Emphasis	Hit Ratio	Observed Cycles	Estimated Cycles	Est/Obs Ratio	Naive Ratio
Des	1346	Encrypts and Decrypts 64 bits	75.71%	155,340	191,564	**1.23**	1.45
Matcnta	40060	Count and Sum Values in a 100x100 Int Matrix	71.86%	1,143,014	1,143,023	**1.00**	1.15
Matcntb	460	Count and Sum Values in a 10x10 Int Matrix	70.73%	12,189	12,189	**1.00**	1.15
Matmula	30044	Multiply 2 50x50 Matrices into a 50x50 Int Matrix	62.81%	7,245,830	7,952,807	**1.10**	1.24
Matmulb	344	Multiply 2 5x5 Matrices into a 5x5 Int Matrix	89.40%	11,396	11,396	**1.00**	1.33
Matsuma	40044	Sum Values in a 100x100 Int Matrix	71.86%	1,122,944	1,122,953	**1.00**	1.15
Matsumb	444	Sum Values in a 10x10 Int Matrix	69.98%	11,919	11,919	**1.00**	1.15
Sorta	2044	Bubblesort of 500 Int Array	97.06%	4,768,228	9,826,909	**2.06**	2.88
Sortb	444	Bubblesort of 100 Integer Array	99.40%	188,696	371,977	**1.97**	2.92
Statsa	16200	Calc Sum, Mean, Var., (2 arrays[1000 doubles])	90.23%	1,237,698	1,447,572	**1.17**	1.29
Statsb	600	..., StdDev., & Corr. Coeff. (2 arrays[25 doubles])	89.21%	32,547	37,246	**1.14**	1.29

Table 2: Dynamic Results for Data Caching

calculated by assuming all data cache references to be misses and dividing those cycles by the observed cycles.

The timing analyzer was able to tightly predict the worst-case number of cycles required for pipelining and data caching for most of the test programs. In fact, for five of them, the prediction was exact or over by less that one-tenth of one percent. 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 number of iterations performed was overrepresented on average by about a factor of two for this inner loop. The strategy of treating a calculated reference miss as a hit in the pipeline and adding the maximum number of cycles associated with the miss penalty to the total time of the path caused overestimations with the *Statsa* and *Statsb* programs, which were the only floating-point intensive programs in the test set. Often delays due to long-running floating-point operations could have been overlapped with data cache miss penalty cycles. *Matmula* had an overestimation of about 10% whereas the smaller data version *Matmulb* was exact. The *Matmul* program has repeated references to the same elements of three different arrays. These references would miss the first time they were encountered, but would be in cache for the smaller *Matmulb* when they were accessed again since the arrays fit entirely in cache. When all the arrays fit into cache there is no interference between them. However, when they do not fit into cache the static simulator conservatively assumes that any *possible* interference must result in a cache miss. Therefore, the categorizations are more conservative and the overestimation is larger. Finally, the *Des* program has several references where an element of a statically initialized array is used as an index into another array. There is no simple method to determine which value from it will be used as the index. Therefore, we must assume that *any* element of the array may be accessed any time the data reference occurs in the program. This forces all conflicting lines to be deleted from the cache state and the resulting categorizations to be more conservative. The *Des* program also has overestimations due to data dependencies. A longer path deemed feasible by the timing analyzer could not be taken in a function due to the value of a variable. Despite the relatively small overestimations detailed above, the results show that with certain restrictions it is possible to tightly predict much of the data caching behavior of many programs.

The difference between the *naive* and *estimated* ratios shows the benefits for performing data cache analysis when predicting worst-case execution times. The benefit of worst-case performance from data caching is not as significant as the benefit obtained from instruction caching [2, 6]. An instruction fetch occurs for each instruction executed. The performance benefit from a write-through/no-allocate data cache only occurs when the data reference from a load instruction is determined by the timing analyzer to be in cache. Load instructions only comprised on average 14.28% of the total executed instructions for these test programs. However, the results do show that performing data cache analysis for predicting worst-case execution time does still result in substantially tighter predictions. In fact, for the programs in the test set the prediction improvement averages over 30%.

The performance overhead associated with predicting WCETs for data caching using this method comes primarily from that of the static cache simulation. The time required for the static simulation increases linearly with the size of the data. However, even with large arrays as in the given test programs this time is rather small. The average time for the static simulation to produce data reference categorizations for the 11 programs given in Table 2 was only 2.89 seconds. The overhead of the timing analyzer averages to only 1.05 seconds.

4. Set-Associative Instruction Caches

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 [4].

The method of static cache simulation provides the means to predict the caching behavior of instruction and data references. This section formalizes the handling of set-associative instruction caches. An instruction is assigned a category for each loop level (*i.e.*, always-hit, always-miss, first-hit or first-miss, as discussed previously in the context of data caches). The analysis for set-associative instruction caches is based on the following formal framework:

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 block.*

The traversal of every possible sequence of blocks leads to an exponential explosion. To avoid this overhead, we restrict the analysis to abstract cache states:

Definition 2 (Abstract Cache State (ACS)) *The abstract cache state of a program line l within a block and a function instance is the set of program lines (for each level of associativity) that can potentially be cached prior to the execution of l within the block 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.

Given the control-flow information of a program and a cache configuration, the ACSs for each block have to be calculated. Using data-flow analysis (DFA), each block has an input state and an output state, corresponding to the ACS before and after the execution of the block, respectively. An iterative algorithm for the calculation of ACS' via DFA is given in [14]. Our 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]. Additional DFA is required to determine the linear cache state and the post-dominator set for each block before a definition for instruction categories can be given.

Definition 3 (Linear Cache State (LCS)) *The linear cache state of a program line l within a block and a function instance is the set of program lines (for each level of associativity) that can potentially be cached in the forward control-flow graph prior to the execution of l within the block and the function instance.*

The forward control-flow graph is the acyclic graph resulting from the removal of back edges (backwards edges forming loops [1]) in the regular control-flow graph. 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.

Definition 4 (Post-dominator Set) *The post-dominator set of a program line l within a block and a function instance is the self-reflexive transitive closure of post-dominating program lines.*

Informally, the post-dominator set describes the program lines certain to be reached from the current block, regardless of the taken paths in the control flow (see [1] for more details).

The instruction categories can now be formally defined (see Definition 5, at the top of the next page) and implemented rather efficiently once DFA has been performed. 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 (see [14] for algorithmic details).

Always-hit: (on spatial locality within the program line) or ((the instruction is in cache in the absence of loops) and ((there are no conflicting instructions in the 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, even without loops and even for all loop preheaders, it is always executed in the loop, not all conflicts fit into the remaining associativity levels but conflicts within the loop fit into the remaining associativity levels for the loop headers, even when disregarding loops).

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.

4.1. Example

Consider the example in Figure 7. 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 1 of foo) and twice within the loop (instances 2 and 3 of foo). For foo(1), A is an always miss since A is not in the ACS of foo(1). For foo(2), A is a first-hit since it was brought into cache by foo(1), there are 3 conflicts within the loop (and only 2 cache lines within the set) but B is the only conflict in the output ACS and the LCS within the loop. For foo(3), A is an always-hit due to temporal locality since it was brought into the ACS 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 ACS of their 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 and brought into cache and there are only 2 conflicting lines within the loop.

4.2. 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

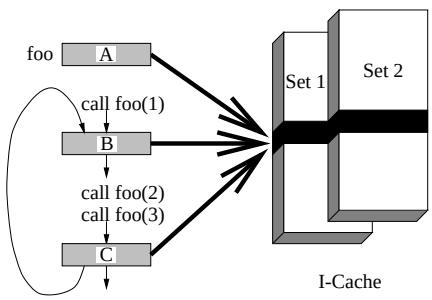


Figure 7: Example of Instruction Categories

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 $s = \bigcup_{1 \leq j \leq n} s_j$.
- 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 $header(\lambda)$ be the set of header paths and $preheader(\lambda)$ be the set of preheader paths of loop λ, respectively.
- 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 $postdom(p)$ be the set of self-reflexive post-dominating programming lines of path p.

Then,

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

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. The miss penalty was assumed to be 9 cycles [7]. (For the numbers reported here, pipeline simulation of the timing analyzer was intentionally disabled to isolate the effects of caching.)

Table 3 shows the results of WCET prediction for a 4-way associative cache with 8 lines. The other cache configurations mentioned before yield similar results in terms of the ratios and are therefore omitted. The programs are described in Table 2. 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 naive ratio (column 5) simulates a disabled cache and was calculated by assuming that all data 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 1.32 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 *Sorta* due to a loop with a varying number of iterations as described in Section 3.5. Likewise, *Des* causes an overestimation due to a data dependency that was also previously described. For the programs *Matcnta*, *Matsuma* and *Statsa*, the number of cycles was slightly overestimated. The programs *Matcnta* and *Matsuma* 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 [14]). This trade-off between accuracy and feasible time complexity still results in relatively tight but not always precise estimations. The program *Statsa* suffers 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.

Program	Observed Cycles	Estimated Cycles	Est./Obs. Ratio	Naive Ratio
Des	95,877	109,069	**1.14**	5.58
Matcnta	443,754	443,790	**1.00**	9.99
Matmulta	1,430,538	1,430,538	**1.00**	10.00
Matsuma	343,628	343,646	**1.00**	9.99
Sorta	3,130,692	6,249,474	**2.00**	10.00
Statsa	183,491	192,518	**1.05**	9.94

Table 3: Worst-Case Execution Times

We also measured the average ratio between estimated and observed cycles for cache associativities between 1 and 8 and observed that this ratio is independent of the level of associativity. Furthermore, the distribution of the instruction categories, averaged over the test set, varied only insignificantly for different levels of associativity. Thus, the presented method for WCET predictions yields tight results regardless of the associativity of caches. Finally, we observed that the

overhead of performing static cache analysis increases linearly with the level of cache associativity. The increase can be attributed to the overhead of bit-vector operations implementing the DFA. The performance overhead for direct-mapped caches is extremely low (about 200 ms) and is still respectable (about 1.7 sec) for $n = 8$, the largest associativity found in today's processors [14]. Thus, static cache simulation is an adequate method to model caches for WCET predictions for contemporary architectures efficiently.

5. Future Work

There are several areas of timing analysis that can be further investigated. The effect of wrap-around fill data caches can be analyzed. We currently assume that each load requires a constant miss penalty for accessing memory. However, cache lines are filled from memory one word at a time, and analyzing the wrap-around fill behavior can tighten the predicted WCET. Timing predictions for set-associative data caches can be produced in a manner similar to that for instruction caches described in this paper. Best case timing bounds for both data and set-associative caches may also be investigated. An eventual goal of this research is to integrate the timing analysis of both instruction and data caches 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.

6. Conclusion

There are two general contributions of this paper. First, an approach for bounding the worst-case data caching performance is introduced. It uses data flow analysis within a compiler to determine a bounded range of relative addresses for each data reference. An address calculator converts these relative ranges to virtual address ranges by examining the order of data declarations and the call graph of the program. Categorizations of the data references are produced by a static simulator. A timing analyzer uses the categorizations when performing pipeline path analysis to predict the worst-case performance for each loop and function in the program. The results so far indicate that the approach is valid and can result in significantly tighter worst-case performance predictions.

Second, a report on an implementation of timing predictions for set-associative caches is given. A formal method and the corresponding operational framework for simulating set-associative caches is described. This method of static cache simulation for set-associative caches 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 cache simulation overhead scales linearly with increasing associativity.

Overall, this paper contributes a comprehensive report on methods and results of worst-case timing analysis for data caches and set-associative caches. The analysis occurred on code generated with all compiler optimizations enabled and requires no user-specified information. The approach taken is unique and provides a considerable step toward realistic worst-case execution time prediction of contemporary architectures and its use in schedulability analysis for real-time systems.

7. Acknowledgements

The authors thank Robert Arnold for providing the timing analysis platform for this research. The anonymous referees also provided helpful suggestions that improved the quality of this paper.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] 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.

[3] 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.

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

[5] J. W. Davidson and D. B. Whalley. A design environment for addressing architecture and compiler interactions. *Microprocessors and Microsystems*, 15(9):459–472, November 1991.

[6] 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.

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

[8] Y. Hur, Y. H. Bea, S.-S. Lim, B.-D. Rhee, S. L. Min, Y. C. 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] F. Mueller. *Static Cache Simulation and its Applications.* PhD dissertation, Dept. of Computer Science, Florida State University, July 1994.

[14] F. Mueller. Generalizing timing predictions to set-associative caches. In *EuroMicro Real-Time Workshop*, June 1997.

[15] Texas Instruments. *TMS390S10 Integrated SPARC Processor*, February 1993.

[16] R. White. *Bounding Worst-Case Data Cache Performance.* PhD dissertation, Dept. of Computer Science, Florida State University, April 1997.