, , 1–26 ()
© Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Timing Analysis for Data and Wrap-Around Fill Caches

RANDALL T. WHITE rwhite@cs.fsu.edu
Florida State University, Computer Science Department, Tallahassee, FL 32306-4530

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

CHRIS HEALY AND DAVID WHALLEY [healy,whalley]@cs.fsu.edu
Florida State University, Computer Science Department, Tallahassee, FL 32306-4530, phone: (850) 644-3506, fax:-0058

MARION HARMON harmon@cis.famu.edu
Florida A&M University, Computer & Information Systems Department, Tallahassee, FL 32307-3101, phone: (850) 599-3042, fax: -3221

Editor: Wolfgang A. Halang

Abstract. The contributions of this paper are twofold. First, an automatic tool-based approach is described to bound worst-case *data cache* performance. The 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, and produces results typically within a few seconds. Results obtained by running the system on representative programs are presented and indicate that timing analysis of data cache behavior usually results in significantly tighter worst-case performance predictions.

Second, a method to deal with realistic cache filling approaches, namely wrap-around-filling for cache misses, is presented as an extension to pipeline analysis. Wrap-around fill analysis is more challenging than traditional cache analysis since the words within a program line are loaded into cache in different cycles according to a predetermined sequence, rather than all at the same time. Results indicate that worst-case timing predictions become significantly tighter when wrap-around-fill analysis is performed.

Overall, the contribution of this paper is a comprehensive report on methods and results of worst-case timing analysis for data caches and wrap-around caches. 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 hard real-time systems.

Keywords: Timing Analysis, Data Cache, Wrap-Around Fill Cache, Worst-Case Execution Time

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 [18, 3]. Static

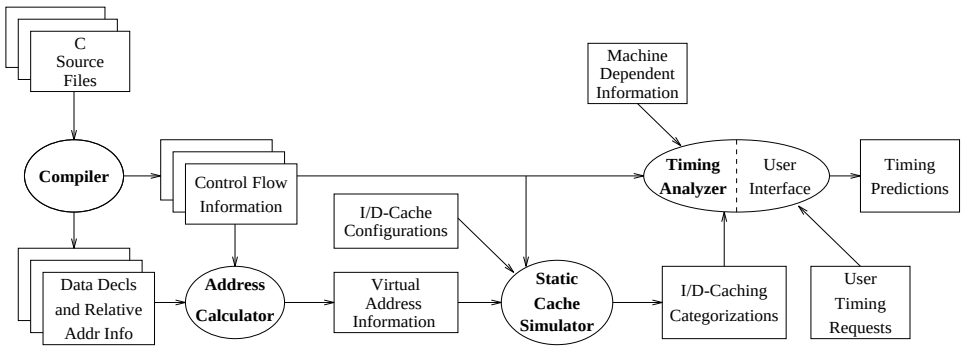


Figure 1. Framework for Timing Predictions

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 a program. 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 wrap-around-fill mechanisms for handling cache misses. Thus, it presents an approach to include common features of contemporary architectures for 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. The *vpo* optimizing compiler [4] 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. This analysis includes the evaluation of architectural characteristics such as pipelining and wrap-around-filling for cache misses. The description of the caching behavior supplied by the static cache simulator is used by the timing analyzer to predict the temporal effect of cache hits and misses overlapped with the temporal behavior of pipelining. The timing analyzer produces WCET predictions for user selected segments of the program or the entire program.

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 [23, 20, 9, 22] to optimized programs on pipelined RISC processors [30, 17, 11], and from uncached architectures to instruction caches [2, 15, 13] and data caches [24, 14, 16]. While there has been some related work in analyzing data caching, there has been no previous work on wrap-around-fill caches in the context of WCET prediction, to our knowledge.

Rawat [24] used a graph coloring technique to bound data caching performance. However, only the live ranges of local scalar variables within a single function were analyzed, which are fairly uncommon references since most local scalar variables are allocated to registers by optimizing compilers.

Kim *et al.* [14] have recently published work about bounding data cache performance for calculated references, which are caused by load and store instructions referencing addressing that can change dynamically. Their technique 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, their approach does not 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.

Li *et al.* [16] 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. However, their ILP approach does facilitate integrating additional user-provided constraints into the analysis.

3. Data Caches

Obtaining tight WCETs in the presence of data caches is quite challenging. Unlike instruction caching, 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. Some 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 simply 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 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 integrated with the pipeline analysis to yield worst-case execution time predictions of program segments.

3.1. Calculation of Relative Addresses

The *vpo* compiler [4] attempts to calculate relative addresses for each data reference associated with load and store instructions after compiler optimizations have been performed (see Figure 1). Compiler optimizations can move instructions between basic blocks and outside of loops so that expansion of registers used in address calculations becomes more difficult. The analysis described here is similar to the data dependence analysis that is performed by vectorizing and parallelizing compilers [5, 6, 7, 21, 28, 29]. However, data dependence analysis is typically performed on a high-level representation. Our analysis had to be performed on a low-level representation after code generation and all optimizations had been applied.

The calculation of relative addresses involves the following steps.

1. The compiler determines for each loop the set of its induction variables, their initial values and strides, and the loop-invariant registers.[1]

2. Expansion of actual parameter information is performed in order to be able to resolve any possible address parameters later.

3. Expansion of addresses used in loads and stores is performed. Expansion is accomplished by examining each preceding instruction represented as a register transfer list (RTL) 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. Expansion of calculated address references (*e.g.* array indexing) requires knowledge of loop induction variables. This approach to expanding addresses provides the ability to handle non-standard induction variables. We are not limited to simple induction variables in simple `for` loops that are updated only at the head of the loop.

Consider the C source code, RTLs and SPARC assembly instructions in Figure 2 for a simple `Initialize` function. The code in `Initialize` goes through elements 1 to 51 in both array A and array B and initializes them to random integers. Note that although delay slots are actually filled by the compiler, they have not been filled when compiling the code for most of the figures in this paper, in order to simplify the examples for the reader.[2]

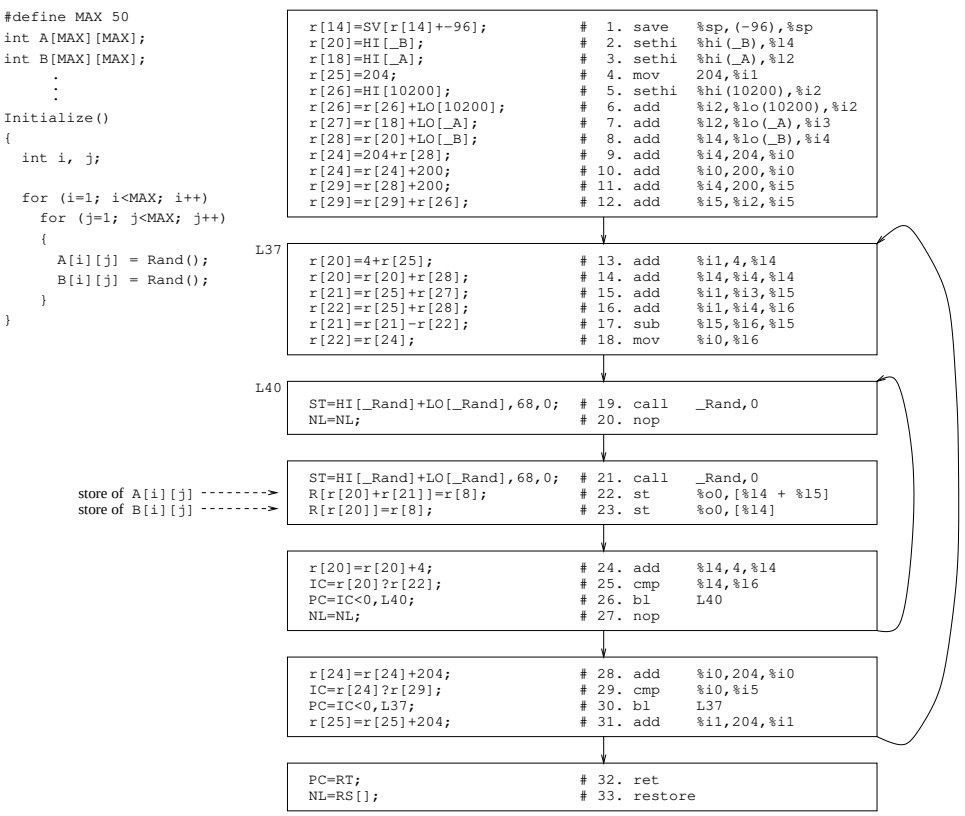


Figure 2. Example C Function, RTLs, and SPARC Assembly for Function Initialize

The first memory address, `r[20] + r[21]` (instruction 22), is for the store of `A[i][j]`. The second memory address, `r[20]` (instruction 23), is for the store of

B[i][j]. Register r[20] is the induction register for the inner loop (instructions 19-27) and thus cannot be expanded. It has an initial value, a stride, and a maximum and minimum number of iterations associated with it, these having been computed and stored earlier in the compilation process.[3] The initial value for r[20] consists of the first element accessed (base address of B plus 4) plus the offset that comes from computing the row location, that of the induction variable for the outer loop, r[25]. The stride is 4 and the minimum and maximum number of iterations are the same, 50. Once the initial value, stride, and number of iterations are available, there is enough information to compute the sequence of addresses that will be accessed by the store of B[i][j]. Knowing that both references have the same stride, the compiler used index reduction to avoid having to use another induction register for the address computation for A[i][j], since it shares the same loop control variables as that for B.

The memory address r[20] + r[21] for A[i][j] includes the address for B (r[20]) plus the difference between the two arrays (r[21]). This can be seen from the following sequence of expansions and simplifications. Remember that register r[20] cannot be immediately expanded since it is an induction register for the inner loop, so the expansion continues with register r[21] as follows. Note that register r[25] will not be expanded either since it is the induction variable for the outer loop (instructions 13-31).

```
1. r[20] + r[21]                                          # load at 22
2. r[20]+(r[21]-r[22])                                    # from 17
3. r[20]+(r[21]-(r[25]+r[28]))                            # from 16
4. r[20]+(r[25]+r[27]-(r[25]+r[28]))                      # from 15
5. r[20]+(r[25]+r[27]-(r[25]+r[20]+LO[_B]))               # from 8
6. r[20]+(r[25]+(r[18]+LO[_A])-(r[25]+r[20]+LO[_B]))      # from 7
7. r[20]+(r[25]+(HI[_A]+LO[_A])-(r[25]+r[20]+LO[_B]))     # from 3
8. r[20]+(r[25]+(HI[_A]+LO[_A])-(r[25]+HI[_B]+LO[_B]))    # from 2
```

The effect of this expansion is simplified in the following steps.

```
9. r[20]+(r[25]+(_A)-(r[25]+_B))   # eliminate HI and LO
10. r[20]+(r[25]+_A-(r[25]+_B))    # remove unnecessary ()'s
11. r[20]+r[25]+_A-r[25]-_B        # remove ()'s and distribute +'s and -'s
12. r[20]+_A-_B                    # remove negating terms
```

Thus, we are left with the induction register r[20] plus the difference between the two arrays. The simplified address expression string is then written to a file containing data declarations and relative address information. When the address calculator attempts to resolve this string to an actual virtual address, it will use the initial value of r[20], which is r[25]+_B+4, and the B's will cancel out (r[25]+_B+4+_A-_B = r[25]+4+_A). Note this gives the initial address of the row in the A array. The range of relative addresses for this example can be depicted algorithmically as shown in Figure 3. For more details on statically determining address information from fully optimized code see [26].

```
for (r[25] = 204; r[25] < 10200; r[25] += 204)
   for (r[20] = r[25]+4+_A; r[20] < r[25]+200+_A; r[20] += 4)
      address{A_{r[25]r[20]}}
```

Figure 3. Algorithmic Range of Relative Addresses for the Load in Figure 2

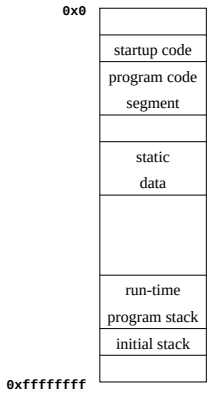


Figure 4. Virtual Address Space (SunOS)

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 also used during 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 4 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 and floating-point constants). In general, the Unix linker (*ld*) places the static data in the same order that the declarations appeared within the assembly files. 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 a 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. Some examples of temporaries include parameters beyond the sixth word passed to a function and memory used to move values between integer and floating-point registers since such movement cannot be accomplished directly on a SPARC. 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 [19].

The iterative algorithm used for static instruction cache simulation [2, 19] is not sufficient for static data 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 have 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

```
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 5. Algorithm to Calculate Data Cache States

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 5 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 [2] 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.

```
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]
```

(a) Detecting Spatial Locality

```
        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.
```

(b) Detecting Temporal Locality across and within Loops

Figure 6. Examples for Spatial and Temporal Locality

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 6(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 6(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 described in other work on instruction

caches [27]. 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.

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 7. The additions to the previously published algorithm [11] to handle calculated references are shown in boldface. Let n be the maximum number of iterations associated with a given 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 selects the longest path for each loop iteration [11, 10]. 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].

```
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 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 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 7. Worst-Case Loop Analysis Algorithm

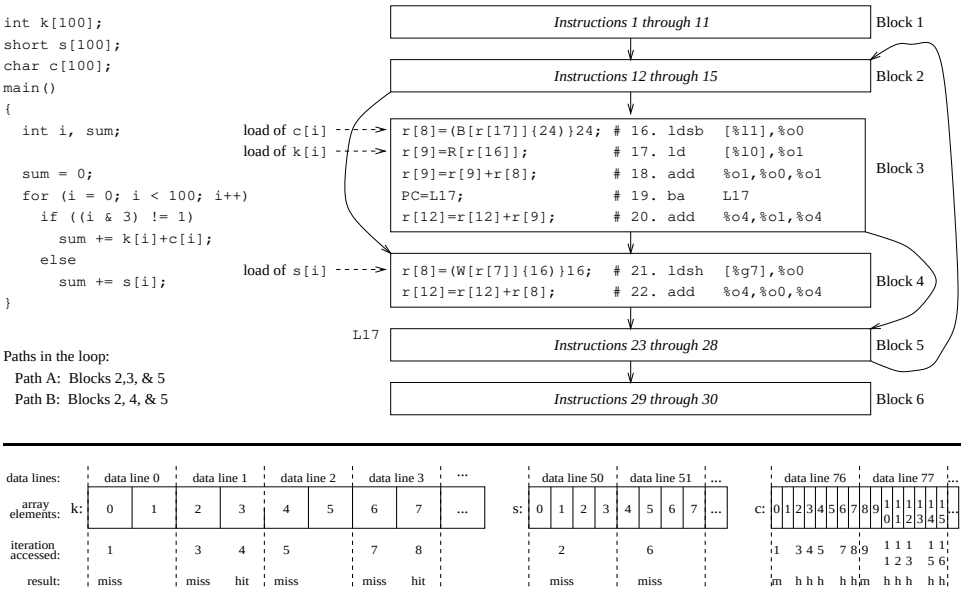
The WHILE loop will subtract one from each calculated reference miss count for the current loop in the longest path chosen on each iteration whenever there are first misses or first hits encountered as misses or hits, respectively. Once no such first misses and first hits are encountered in the longest 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 longest exit path may be shorter than the longest continue path.

A correctness argument about the interpretation of calculated references needs to show that the calculated references are treated as misses the appropriate number of times. The algorithm treats a calculated reference as a miss until its specified number of calculated misses for the loop is exhausted. The IF-THEN portion of the WHILE loop subtracts one from each calculated reference miss count since only a single iteration is analyzed and each calculated reference can only miss once in a given loop iteration. The ELSE-IF-THEN portion of the WHILE loop subtracts the minimum of the misses remaining in any calculated reference for that path and the number of iterations remaining in the loop. The number of iterations analyzed is again the same as the number of misses subtracted for each calculated reference. Since the misses for the calculated references are evaluated before the hits, the interpretation of calculated references will not underestimate the actual number of calculated misses given that the data references have been categorized correctly.

An example is given in Figure 8 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 8. 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 7 to estimate the WCET for the loop in the example shown in Figure 8. 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




```
k[i]: c 50 from [m h m h .. m h ]
s[i]: c 25 from [m h h h m h h h .. m h h h]
c[i]: c 13 from [m h h h h h h h m h h h h h h h .. m h h h h h h h m h h h]
```

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	19	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 8. Example to Illustrate Worst-Case Loop Analysis Algorithm

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, where each function instance is uniquely identified by the sequence of calls resulting in its invocation. 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

Table 1. Timing Analysis Steps for the loop in Figure 8

step	start iter	longest path cycles	min_misses	iters	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

instance, consider the example shown in Figure 8 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, 11].

3.5. Results

Measurements were obtained on code generated for the SPARC architecture by the *vpo* optimizing compiler [4]. The machine-dependent information contained the pipeline characteristics of the MicroSPARC I processor [25]. 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 [25]. 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 last five test programs. The **a** version had the same size arrays that were used in previous studies [2, 11]. 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 run-time 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 3 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 [8], to simulate data cache and pipeline affects and count the number of cycles. The *estimated cycles* were obtained from the timing analyzer discussed in

Table 2. Test Programs for Data Caching

Name	Num Bytes	Description or Emphasis
Des	1346	Encrypts and Decrypts 64 bits
Matcnta	40060	Count and Sum Values in a 100x100 Int Matrix
Matcntb	460	Count and Sum Values in a 10x10 Int Matrix
Matmula	30044	Multiply 2 50x50 Matrices into a 50x50 Int Matrix
Matmulb	344	Multiply 2 5x5 Matrices into a 5x5 Int Matrix
Matsuma	40044	Sum Values in a 100x100 Int Matrix
Matsumb	444	Sum Values in a 10x10 Int Matrix
Sorta	2044	Bubblesort of 500 Int Array
Sortb	444	Bubblesort of 100 Integer Array
Statsa	16200	Calc Sum, Mean, Var., (2 arrays[1000 doubles])
Statsb	600	..., StdDev., & Corr. Coeff. (2 arrays[25 doubles])

Section 3.4. The *estimated ratio* is the quotient of these two values. The *naive ratio* was 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

Table 3. Dynamic Results for Data Caching

Name	Hit Ratio	Observed Cycles	Estimated Cycles	Est/Obs Ratio	Naive Ratio
Des	75.71%	155,340	191,564	**1.23**	1.45
Matcnta	71.86%	1,143,014	1,143,023	**1.00**	1.15
Matcntb	70.73%	12,189	12,189	**1.00**	1.15
Matmula	62.81%	7,245,830	7,952,807	**1.10**	1.24
Matmulb	89.40%	11,396	11,396	**1.00**	1.33
Matsuma	71.86%	1,122,944	1,122,953	**1.00**	1.15
Matsumb	69.98%	11,919	11,919	**1.00**	1.15
Sorta	97.06%	4,768,228	9,826,909	**2.06**	2.88
Sortb	99.40%	188,696	371,977	**1.97**	2.92
Statsa	90.23%	1,237,698	1,447,572	**1.17**	1.29
Statsb	89.21%	32,547	37,246	**1.14**	1.29
average	80.75%	N/A	N/A	**1.24**	1.55

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, 11]. 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 3 was only 2.89 seconds. The overhead of the timing analyzer averages to 1.05 seconds.

4. Wrap-Around-Filling for Instruction Cache Misses

Several timing tools exist that address the hit/miss behavior of an instruction cache. But modern instruction caches often employ various sophisticated approaches to decrease the miss rate or reduce the miss penalty [12]. One approach to reduce the miss penalty in an instruction cache is *wrap-around fill*. A processor employing this feature will load a cache line one word at a time, starting with the instruction

Table 4. Order of Fill When Loading Words of a Cache Line

First Requested Word within Cache Line	Miss Delay for Word							
	0	1	2	3	4	5	6	7
0	7	8	10	11	13	14	16	17
1	8	7	10	11	13	14	16	17
2	16	17	7	8	10	11	13	14
3	16	17	8	7	10	11	13	14
4	13	14	16	17	7	8	10	11
5	13	14	16	17	8	7	10	11
6	10	11	13	14	16	17	7	8
7	10	11	13	14	16	17	8	7

that caused the cache miss. For each word in the program line that is being loaded into the cache, the associated instruction cannot be fetched until its word has been loaded. The motivation for wrap-around fill is to let the CPU proceed with this instruction and allow the pipelined execution to continue while subsequent instructions are loaded into cache. Thus, the benefit is that it is not necessary to wait for the entire cache line to be loaded before proceeding with the execution of the fetched instruction. However, this feature further complicates timing analysis since it can introduce *dead cycles* into the pipeline analysis during those cycles when no instruction is being loaded into cache [25]. Wrap-around fill is used on several recent architectures, including the Alpha AXP 21064, the MIPS R10000 and the IBM 620.

Table 4 shows when words are loaded into cache on the MicroSPARC I processor [25]. In each instruction cache line there are eight words, hence eight instructions. The rows of the table are distinguished by which word w within a cache line was requested when the entire line was not found in cache. The leftmost column shows that any of the words 0-7 can miss and become the first word in its respective program line to be loaded into cache. It takes seven cycles for the requested instruction to reach the instruction cache. During the eighth cycle, the word with which w is paired, either $w + 1$ if w is even or w - 1 if w is odd, gets loaded into cache. After each pair of words is loaded into cache, there is a *dead cycle* during which no word is written. Table 6 indicates that the MicroSPARC I has dead cycles during the ninth, twelfth and fifteenth cycles after a miss occurs. It takes seventeen cycles for an entire program line to be requested from memory and completely loaded into cache. Note that on the MicroSPARC I there is an additional requirement that an entire program line must be *completely* loaded into cache before a different program line can be accessed (whether or not that other program line is already in cache).

For wrap-around-fill analysis, information stored with each path and loop includes the program line number and the cycles during which the words of the loop's first and last program lines are loaded into cache. These cycles are called the *available* times, and the timing analyzer calculates beginning and ending *available* times for each path in a particular loop. For loop analysis, this set of beginning and ending information is propagated along with the worst-case path's pipeline requirements and data hazard information. Keeping track of when the words of a program line

are available in cache is analogous to determining when a particular pipeline stage is last occupied and to detecting when the value of a register is available via hardware forwarding. These *available* times are used to carry out wrap-around-fill analysis of paths and loops. This analysis detects the delays associated with *dead cycles* and cases where these delays can be overlapped with pipeline stalls to produce a more accurate WCET prediction.

4.1. Wrap-Around-Fill Delays Within a Path

During the analysis of a single path of instructions, it is necessary to know when the individual words of a cache line will be loaded with the appropriate instructions. When the timing analyzer processes an instruction that is categorized as a miss, it can automatically determine when each of the instructions in this program line will be loaded into cache, according to the order of fill given in the machine-dependent information. The timing analyzer stores the program line number and the relative word number in that line for every instruction in the program. During the analysis of a path, the timing analyzer can update information about which program line is arriving into cache and when the words of that line are *available* to be fetched without any delay. At the point the timing analyzer is finished examining a path, it will store the information associated with the first and last program lines referenced in this path, including the cycles during which words in these lines become available in cache, plus the amount of delay caused by latencies from the filling of cache lines. Such information will be useful when the path is evaluated in a larger context, namely when the first iteration of a loop or a path in a function is entered or called from another part of the program.

Figure 9 shows the algorithm that is used to determine the number of cycles associated with a instruction fetch while analyzing a path. The cycles when the words become available in the last line fetched are calculated on each miss. In order to demonstrate the correctness of the algorithm, one must show that the required number of cycles are calculated for the wrap-around-fill delay on each instruction fetch. There are three possible cases. The first case is when the instruction being fetched is in the last line fetched, which means the instruction fetch must be a hit. Line 4 in the algorithm uses the arrival time of the associated word containing the instruction to determine if extra cycles are needed for the IF stage. The second and third cases are when the reference was not in the last line fetched and the instruction fetch could be a hit or a miss, respectively. In either case, cycles for the IF stage of the instruction have to include the delay to complete the loading of the last line, which is calculated at line 6. Line 8 calculates the additional cycles for the IF stage required for a miss to load the requested word in the line. Lines 9-10 establish the arrival times of the words in the line when there is a miss. All three cases are handled. Thus, the algorithm is correct given that arrival times of the current line preceding the first instruction in the path are accurate. Techniques to determine the arrival times at the point a path is entered are described in the following sections.

```
// matrix containing information from Table 4
const int waf_delay[WORDS_PER_LINE][WORDS_PER_LINE]

// indicates when each word of the last line fetched become available
int available[WORDS_PER_LINE]

const int max_delay // delay required to load the last word of a line

1:   curr_word_num = inst_word_num % WORDS_PER_LINE.
2:   first_cycle = first vacancy of IF stage.
3:   IF instruction in last_line_fetched THEN
4:      cycles_in_IF = max(0, available[curr_word_num] - first_cycle).
5:   ELSE
6:      cycles_in_IF = previous_line_delay = max(0,last_word_avail - first_cycle).
7:      IF reference was a miss THEN
8:        cycles_in_IF += waf_delay[curr_word_num][curr_word_num].
9:        FOR i = 0 TO WORDS_PER_LINE-1 DO
10:          available[i] = first_cycle + previous_line_delay + waf_delay[curr_word_num][i].
11:       last_word_avail = first_cycle + previous_line_delay + max_delay.
12:       last_line_fetched = line of current instruction.
13:  cycles_in_IF += 1.
```

Figure 9. Algorithm to Calculate WAF Delay within a Path

4.2. Delays Upon Entering A Loop or Function

During path analysis, when the timing analyzer encounters a loop or a function call in that path (child), it determines if the first instruction in the child lies in a different program line than the instruction executed immediately before entering the loop or function. If it does, then the first instruction in the child must be delayed from being fetched if the program line containing the last instruction executed before the child is still loading into cache. If the two instructions lie in the same program line, then it is only necessary to ensure that the instructions belonging to the first program line in the child will be *available* when fetched. Often, these *available* times (and corresponding *dead cycle* delays) have already been calculated by the child. Likewise, the *available* times could have been calculated in some other child encountered earlier in the current path, *i.e.* in the situation where the path calls two functions that share a program line.

Figure 10 shows a small program containing a loop that has ten iterations and comprises instructions 5-9. The first instruction in the loop is instruction 5, and in memory this instruction is located in the same program line (0) as instructions 0-4. At the beginning of the program execution, instruction 0 misses in cache and causes program line 0, containing instructions 0-7, to load into cache. On the first iteration of the loop, the timing analyzer detects that instruction 5 only needs to

spend 1 cycle in the IF stage; there is no *dead cycle* associated with instruction 5 even though program line 0 is in the process of still being fetched into cache during the first iteration.

C Source Code		SPARC Instructions	Word Number	Prog. Line	Categorizations
`main()`		`inst 0: save %sp,-104,%sp`	**0**	**0**	**m**
`{`		`inst 1: mov %g0,%o3`	**1**	**0**	**h**
`  int i, a[10];`		`inst 2: add %sp,.1_a,%o0`	**2**	**0**	**h**
		`inst 3: mov %o0,%o4`	**3**	**0**	**h**
`  for (i = 0; i < 10; ++i)`		`inst 4: add %o0,40,%o5`	**4**	**0**	**h**
`    a[i] = i;`	`L16:`	`inst 5: st %o3,[%o4]`	**5**	**0**	**h**
		`inst 6: add %o4,4,%o4`	**6**	**0**	**h**
`  return 0;`		`inst 7: cmp %o4,%o5`	**7**	**0**	**h**
`}`		`inst 8: bl L16`	**0**	**1**	**fm->fm**
		`inst 9: add %o3,1,%o3`	**1**	**1**	**h**
		`inst 10: ret`	**2**	**1**	**h**
		`inst 11: restore %g0,%g0,%o0`	**3**	**1**	**h**

Pipeline Diagram

| | | cycle | | |

iteration cannot be fetched until cycle 37. This is a situation where a delay due to jumping to a new program line takes place between loop iterations.

4.4. Results

The results of evaluating the same test programs as in section 3.5 are shown in Table 5. The fifth column of the table gives the ratio of the estimated cycles to the observed cycles when the timing analyzer was executed with wrap-around-fill analysis enabled. The sixth column shows a similar ratio of estimated to observed cycles, but this time with wrap-around-fill analysis disabled. In this mode, the timing analyzer assumes a constant penalty of 17 cycles for each instruction cache miss, which is the maximum miss penalty an instruction fetch would incur. The fetch delay actually attains this maximum only when there are consecutive misses, as in the case when there is a call to a function and the instruction located in the delay slot of the call and the first instruction in the function are both misses. In this case, this second miss will incur a 17 cycle penalty because the entire program line containing the delay slot instruction must be completely loaded first. All data cache accesses are assumed to be hits in both the timing analyzer and the simulator for these experiments.

Table 5. Results for the Test Programs

Name	Hit Ratio	Observed Cycles	Estimated Cycles	Ratio with w-a-f Analysis	Ratio without w-a-f Analysis
Des	86.16%	154,791	171,929	1.11	1.38
Matcnta	81.86%	2,158,038	2,161,172	1.00	1.12
Matmula	98.93%	4,544,944	4,547,299	1.00	1.01
Matsuma	93.99%	1,131,964	1,132,178	1.00	1.13
Sorta	76.06%	14,371,854	30,714,661	2.14	2.52
Statsa	88.45%	1,020,769	1,020,810	1.00	1.11
average	87.58%	N/A	N/A	1.21	1.38

The WCET of these programs when wrap-around-fill analysis is enabled is significantly tighter than when wrap-around fill is not considered. *Des* and *Sorta* has overestimations for the same reasons as described in Section 3.5. The small overestimations in the remaining programs primarily result from the timing analyzer's conservative approach to *first miss*-to-*first miss* categorization transitions. These slight overestimations also occurred when the timing analysis assumed a constant miss penalty [11]. Because this situation occurs infrequently, this approach resulted in only small overestimations. The overhead of executing the timing analysis was quite small even with wrap-around-fill analysis. The average time required to produce the WCET of the programs in Table 5 was only 1.27 seconds.

5. Future Work

There are several areas of timing analysis that can be further investigated. More hardware features, such as write buffers and branch target buffers, could be modeled in the timing analysis. Best case timing bounds for various types of caches and other hardware features 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 accuracy of our simulator and 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 presented. 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 technique for WCET prediction for wrap-around-fill caches is presented. When processing a path of instructions, the timing analyzer computes when each instruction in the entire program line will be loaded into cache based on instruction categorizations that indicate which instruction fetches could result in cache misses. The timing analyzer uses this information to determine how much delay, if any, a fetched instruction will suffer due to wrap-around fill. When analyzing larger program constructs such as loops or function instances, the wrap-around-fill information associated with each path is used to detect delays beyond the scope of a single path. The results indicate that WCET bounds are significantly tighter than when the timing analyzer conservatively assumes a constant miss penalty.

Overall, this paper contributes a comprehensive report on methods and results of worst-case timing analysis for data caches and wrap-around caches. 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 hard real-time systems.

Acknowledgments

The authors thank the anonymous referees for their comments that helped improve the quality of this paper and Robert Arnold for providing the timing analysis platform for this research. The research on which this article is based was supported in part by the Office of Naval Research under contract number N00014-94-1-006

and the National Science Foundation under cooperative agreement number HRD-9707076.

Notes

1. A basic loop induction variable only has assignments of the form `v := v` $\pm$ `c`, where `v` is a variable or register and `c` is an integer constant. Non-basic induction variables are also only incremented or decremented by a constant value on each loop iteration, but get their values either directly or indirectly from basic induction variables. A variety of forms of assignment for non-basic induction variables are allowed. Loop invariant values do not change during the execution of a loop. A discussion of how induction variables and loop invariant values are identified can be found elsewhere [1].
2. Annulled branches on the SPARC do not actually access memory (or update registers) for instructions in the delay slot when the branch is not taken. This simple feature causes a host of complications when a load or a store is in the annulled delay slot. However, our approach does correctly handle any such data reference.
3. This earlier computation and expansion of the initial value string of an induction register proceeds in basically the same manner as has already been discussed, except that loop invariant registers are expanded as well.
4. On the MicroSPARC I, a `st` instruction is required to spend two cycles in the MEM stage.

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. N. Audsley, A. Burns, R. Davis, K. Tindell, and A. J. Wellings. Fixed priority pre-emptive scheduling: An historical perspective. *J. of Real-Time Systems*, 8:173–198, 1995.
4. 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.
5. B. Chapman, P. Mehrotra, and H. Zima. Programming in vienna fortran. contractor report 189623, NASA Langley Research Center, 1992.
6. B. Chapman, P. Mehrotra, and H. Zima. High performance fortran without templates. contractor report 191451, NASA Langley Research Center, 1993.
7. B. Chapman, P. Mehrotra, and H. Zima. Extending hpf for advanced data parallel applications. contractor report 194913, NASA Langley Research Center, 1994.
8. 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.
9. 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.
10. C. A. Healy, R. D. Arnold, R. Mueller, D. Whalley, and M. G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1):53–70, January 1999.
11. 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.
12. J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 2nd edition, 1996.
13. Y. Hur, Y. H. Bae, S.-S. Lim, B.-D. Rhee, S. L. Min, C. Y. Park, M. Lee, H. Shin, and C. S. Kim. Worst case timing analysis of RISC processors: R3000/R3010 case study. In *IEEE Real-Time Systems Symposium*, pages 308–319, December 1995.

14. 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.
15. 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.
16. Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *IEEE Real-Time Systems Symposium*, pages 254–263, December 1996.
17. S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, and C. S. Kim. An accurate worst case timing analysis for RISC processors. In *IEEE Real-Time Systems Symposium*, pages 97–108, December 1994.
18. 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.
19. F. Mueller. *Static Cache Simulation and its Applications*. PhD thesis, Dept. of CS, Florida State University, July 1994.
20. C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–61, March 1993.
21. C. D. Polychronopoulos. *Parallel Programming and Compilers*. Kluwer, 1988.
22. P. Puschner. *Zeitanalyse von Echtzeitprogrammen*. PhD thesis, Dept. of CS, Technical University Vienna, December 1993.
23. P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2):159–176, September 1989.
24. J. Rawat. Static analysis of cache analysis for real-time programming. Master's thesis, Iowa State University, 1995.
25. Texas Instruments. *TMS390S10 Integrated SPARC Processor*, February 1993.
26. R. White. *Bounding Worst-Case Data Cache Performance*. PhD thesis, Dept. of Computer Science, Florida State University, April 1997.
27. R. White, F. Mueller, C. Healy, D. Whalley, and M. Harmon. Timing analysis for data caches and set-associative caches. In *IEEE Real-Time Technology and Applications Symposium*, pages 192–202, June 1997.
28. M. Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, 1989.
29. M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.
30. N. Zhang, A. Burns, and M. Nicholson. Pipelined processors and worst case execution times. *Real-Time Systems*, 5(4):319–343, October 1993.