Integrating the Timing Analysis of Pipelining and Instruction Caching*

Christopher A. Healy, David B. Whalley
Computer Science Dept., Florida State Univ.
Tallahassee, FL 32306-4019
e-mail: whalley@cs.fsu.edu, phone: (904) 644-3506

Marion G. Harmon
Comp. & Info. Sys. Dept., Florida A&M Univ.
Tallahassee, FL 32307-3101
e-mail: harmon@cis.famu.edu, phone: (904) 599-3042

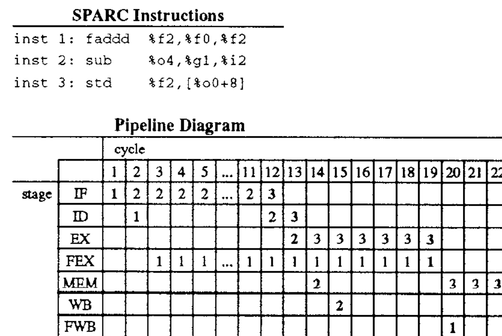
Abstract

Recently designed machines contain pipelines and caches. While both features provide significant performance advantages, they also pose problems for predicting execution time of code segments in real-time systems. Pipeline hazards may result in multicycle delays. Instruction or data memory references may not be found in cache and these misses typically require several cycles to resolve. Whether an instruction will stall due to a pipeline hazard or a cache miss depends on the dynamic sequence of previous instructions executed and memory references performed. Furthermore, these penalties are not independent since delays due to pipeline stalls and cache miss penalties may overlap. This paper describes an approach for bounding the worst-case performance of large code segments on machines that exploit both pipelining and instruction caching. First, a method is used to analyze a program's control flow to statically categorize the caching behavior of each instruction. Next, these categorizations are used in the pipeline analysis of sequences of instructions representing paths within the program. A timing analyzer uses the pipeline path analysis to estimate the worst-case execution performance of each loop and function in the program. Finally, a graphical user interface is invoked that allows a user to request timing predictions on portions of the program.

1. Introduction

Many architectural features, such as pipelines and caches, in recent processors present a dilemma for architects of real-time systems. Use of these architectural features can result in significant performance improvements. In order to exploit these performance improvements in a real-time system, the WCET (Worst Case Execution Time) must be determined statically. Yet these same features introduce a potentially high level of unpredictability. Dependencies between instructions can cause pipeline hazards that may delay the completion of instructions. Instruction or data cache misses can also require several cycles to resolve. Predicting the caching behavior of an instruction is even more difficult since it may be affected by memory references that occurred long before the instruction was executed.

Unfortunately, the timing analysis of these features is exacerbated since pipelining and caching behavior are not independent. For instance, consider the code segment and pipeline diagram in Figure 1 consisting of three SPARC instructions.[1] Each number within the pipeline diagram represents that the specified instruction is currently in the pipeline stage shown to the left and is in that stage during the cycle indicated above. The first instruction performs a floating-point addition and requires a total of 20 cycles. Fetching the second instruction results in a cache miss, which is assumed to have a miss penalty of nine additional cycles. The third instruction has a data dependency with the first instruction and the execution of its MEM stage is delayed until the floating-point addition is calculated.[2] The miss penalty associated with the access to main memory to fetch the second instruction is completely overlapped with the execution of the floating-point addition in the first instruction. If the pipeline analysis and cache miss penalty were treated independently, then the number of estimated cycles associated with these



SPARC Instructions

```
inst 1: faddd  %f2,%f0,%f2
inst 2: sub    %o4,%g1,%i2
inst 3: std    %f2,[%o0+8]
```

Pipeline Diagram

stage \ cycle	1	2	3	4	5	...	11	12	13	14	15	16	17	18	19	20	21	22
IF	1	2	2	2	2	...	2	3										
ID		1						2	3									
EX									2	3	3	3	3	3	3			
FEX			1	1	1	...	1	1	1	1	1	1	1	1	1			
MEM										2						3	3	3
WB											2							
FWB																1		

Figure 1. Overlapping pipeline stages with a cache miss.

[1] The pipeline cycles and stages represent the execution of these instructions on a MicroSPARC I processor [1].

[2] A `std` instruction has no write back stage since a store instruction only updates memory and not a register. The `std` instruction also requires three cycles to complete the MEM stage on the MicroSPARC I.

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

1052-8725/95 $04.00 © 1995 IEEE

instructions would be increased from 22 to 31 (i.e. by the cache miss penalty).

This paper describes an approach for integrating the timing analysis of pipelining and instruction caching behavior. Let a task be the portion of code executed between two scheduling points (context switches) in a system with a non-preemptive scheduling paradigm. When a task starts execution, the cache memory is assumed to be invalidated. During task execution, instructions are brought into cache and often result in many hits and misses that can be predicted statically. These predictions can be integrated with pipeline analysis to estimate tight WCET bounds.

Figure 2 depicts an overview of the approach described in this paper for bounding the worst-case performance of large code segments on machines with pipelines and instruction caches. Control-flow information, which could have been obtained by analyzing assembly or object files, is stored as the side effect of the compilation. The control-flow information is passed to a static cache simulator. It constructs the control-flow graph of the program that consists of the call graph and the control flow of each function. The program control-flow graph is then analyzed for a given cache configuration and a categorization of each instruction's potential caching behavior is produced. The timing analyzer uses these categorizations to determine whether an instruction fetch should be treated as a hit or a miss during the pipeline analysis. It produces a worst-case estimate for each loop and function within the program. Finally, user interface windows are displayed allowing one to request the timing bounds for portions of the program.

2. Instruction caching categorization

The method of static cache simulation is used to statically categorize the caching behavior of each instruction using a specific cache configuration in a given program.[3] The static simulation consists of three phases. First, a program control-flow graph of the entire program is constructed. This includes the control flow within each function and a function instance graph. A function instance graph is simply a call graph where each function is uniquely identified by the sequence of call sites required for its invocation. Thus, a directed acyclic call graph (without recursion) is transformed into a tree of function instances.

Next, this program control-flow graph is analyzed to determine the possible program lines that can be in cache at the entry and exit of each basic block within the program. The iterative algorithm in Figure 3 is used to calculate an input and output cache state for each basic block in the function instance graph. A cache state is simply the subset of all program lines that can potentially be cached at that point in the control flow.

```
input_state(top) = all invalid lines
WHILE any change DO
  FOR each basic block instance B DO
    input_state(B) = NULL
    FOR each immed pred P of B DO
      input_state(B) += output_state(P)
    output_state(B) =
      (input_state(B) + prog_lines(B))
      - conf_lines(B)
```

Figure 3. Algorithm to calculate cache states.

Finally, the input state for each basic block is used to categorize the caching behavior of each instruction within the block. An instruction's caching behavior is assigned to one of four categories for each loop level in which an instruction is contained. Note that each function is treated as a loop that executes for a single iteration. The four categories of caching behavior are:

Always Miss. The instruction is not guaranteed to be in cache when it is referenced.

Always Hit. The instruction is guaranteed to always be in cache when it is referenced.

First Miss. The instruction is not guaranteed to be in cache on its first reference each time the loop is entered, but is guaranteed

[3] Static cache simulation is only briefly introduced in this section. It is described in more detail elsewhere [2], [3], [4], [5].

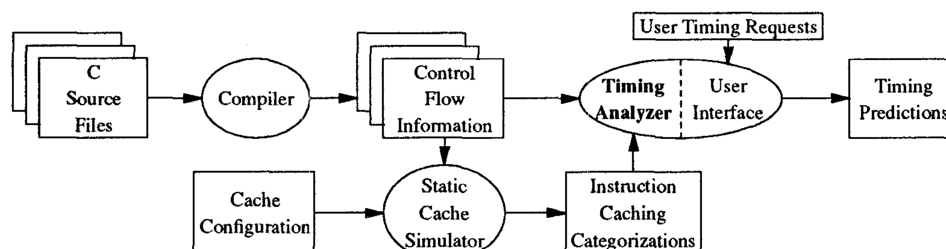


Figure 2. Overview of bounding pipeline and cache performance.

to be in cache on subsequent references.

First Hit. The instruction is guaranteed to be in cache on its first reference each time the loop is executed, but is not guaranteed to be in cache on subsequent references.

3. Pipeline path analysis

This section describes how the analysis of the pipeline performance of a sequence of instructions is accomplished. First, information about each type of instruction is read from a machine-dependent data file. This pipeline information for each type of instruction includes the worst-case number of cycles required by each stage of the pipeline for its execution.[4] The analyzer also stores the latest stage each source operand of the instruction can receive its value via forwarding without delay and the earliest stage in which the result of the instruction type can be forwarded. Finally, information about the specific instructions in the sequence is obtained. This information includes the actual registers associated with the source and destination operands, which is obtained from the control-flow information generated by the compiler, and the instruction caching categorization of each instruction, which is produced by the static cache simulator.

During the analysis of a path, the analyzer keeps track of the total number of cycles required by the path and a set of pipeline information. This information includes when each pipeline stage was first and last used within the path for avoiding structural hazards.[5] It is represented as the number of cycles from the beginning and end of the path for each pipeline stage. In addition, information indicating when each register was first and last used in the path is also maintained to avoid data hazards.[6] Again, this information is represented as the number of cycles from the beginning and end of the path for each register. The set of pipeline information for avoiding hazards after the three instructions in Figure 1 have been analyzed is shown in Tables 1 and 2. Table 1 represents the information for avoiding structural hazards. Only the numbers shown in bold are required to be stored. These values represent when each stage was first used from the beginning of the path and last used from the end. The values in the table correspond to the information associated with the instruction numbers that are represented in bold in Figure 1.

[4] The number of cycles required for some floating-point instructions on processors can vary depending upon the values of its operands.

[5] A structural hazard indicates that a stage of an instruction cannot be executed earlier due to the pipeline stage already being used.

[6] A data hazard indicates that a particular stage of an instruction cannot be executed earlier due to the pipeline stage using a source register that matches the destination register not yet updated by a pipeline stage of another instruction.

Table 2 represents the information for avoiding data hazards. Only the information for the registers referenced in the instructions in Figure 1 are shown.

Stage	IF	ID	EX	FEX	MEM	WB	FWB
Beg Inst	1	1	2	1	2	2	1
Cycles from Beg	**0**	**1**	**12**	**2**	**13**	**14**	**19**
End Inst	3	3	3	1	3	2	1
Cycles from End	**10**	**9**	**3**	**3**	**0**	**7**	**2**

Table 1. Structural hazard information for the instructions in Figure 1.

Register	%g1	%o0	%o4	%i2	%f0	%f2
first needed	**12**	**13**	**12**	N/A	**2**	**2**
last produced	N/A	N/A	N/A	**9**	N/A	**3**

Table 2. Data hazard information for the instructions in Figure 1.

This set of pipeline information is created by processing one instruction at a time from the sequence of instructions that comprise a path. Each instruction can be represented by the same form of pipeline information that is shown in Tables 1 and 2 for a path. This information is modified if it is found that the instruction's caching categorization indicates that the instruction fetch was a miss. The miss penalty is used to increment the total number of cycles and the cycles from the beginning (structural hazard information) for all other stages besides the IF stage and the first needed registers (data hazard information) for that instruction. The addition of an instruction to the pipeline information for a path will not only update the total number of cycles and the information associated with the end of the pipeline, but also the beginning of the pipeline if a referenced stage or register in the instruction had not been previously used.

Retaining this set of pipeline information allows additions to the beginning or end of a path. Since both the pipeline requirements for a path and a single instruction can be represented with this set of pipeline information, concatenating two paths together can be accomplished in the same manner as concatenating an instruction onto the end of a path. The concatenation is accomplished one stage at a time. A stage from the second set of pipeline information is moved to the earliest cycle that does not violate any of the following conditions.

(1) There is no structural hazard with another instruction. For instance, the beginning of the IF stage of instruction 2 in Figure 1 could not be placed in cycle 1 since that stage was already occupied.

(2) There is no data hazard due to a previous instruction producing a result that is needed by a source operand of the current instruction in that stage. For example, the

beginning of the MEM stage for instruction 3 in Figure 1 could not be moved past the FEX stage of instruction 1 at cycle 19 due to the data hazard between the `faddd` and `std` instructions.

(3) The placement of the instruction does not violate its own pipeline requirements. For instance, the ID stage of instruction 2 has to occur at least 10 cycles after the beginning of its IF stage in Figure 1.

Other information associated with the pipeline analysis of a path need not be stored. For instance, it does not matter when instruction 2 entered the ID stage after the pipeline information has been calculated for all three instructions in Figure 1. No instruction being added to either the beginning or end of the pipeline could possibly have a structural hazard with the ID stage of instruction 2 since it would first have a structural hazard with the ID stage of instruction 1 or instruction 3, respectively. Thus, the amount of pipeline information associated with a path is dramatically reduced as opposed to storing how each stage is used during every cycle. Furthermore, no limit need be imposed on the amount of potential overlap when concatenating the analysis of two paths.

4. Loop analysis

In order to predict the worst-case execution time of a loop, the timing analyzer has to predict the execution time of each possible path within the loop. The timing analyzer will reserve either one cycle or the number of cycles associated with a cache miss for the IF (instruction fetch) stage for each instruction categorized as an always hit or always miss, respectively. If an instruction is categorized as a first miss, then the timing analyzer will treat the instruction fetch as a miss if the program line has not yet been encountered as a first miss in the timing of the loop. If the program line has been encountered, then the instruction fetch will be treated as a hit instead. Likewise, if an instruction is categorized as a first hit, then the timing analyzer will treat the instruction fetch as a cache hit on the first reference and a cache miss thereafter.

With pipelining it is possible that the combination of a set of paths may produce a longer execution time than just selecting the longest path. For instance, consider a loop with two paths that take about the same number of cycles to execute. One path has a floating-point addition near the beginning of the path and the other path has a floating-point addition near the end. Alternating between the paths will produce the worst case execution time since there will be a structural hazard between the two floating-point additions.

To avoid the problem of calculating all combinations of paths, which would be the only method for obtaining perfectly accurate estimations, it was decided to union the pipeline effects of the paths for an iteration of a loop together. This unioning of pipeline information simplified the algorithm and also did not cause a noticeable overestimation. All paths through a loop start with the same loop header block. Thus, the beginning pipeline information (stages and registers) is rarely affected. Paths through a loop often end with the same block of instructions. In addition, one path may be longer than the others, so the ending pipeline information is often not affected.

Figure 4 shows a toy function and its corresponding SPARC assembly code.[7] There are two possible paths through an iteration of the loop in the program, <7,8,12,13,14,15,16> and <7,8,9,10,11,13,14,15,16>. Figure 5 shows the instructions and corresponding pipeline diagrams for the two paths within the loop.[8] To simplify the example, it is assumed that the loop has already been executed and all of the instructions and data are in cache (i.e. there are no instruction fetch or data memory misses). Table 3 shows the structural hazard information for the two paths in Figure 5 and how the information in path 1 has to be adjusted before it would be unioned. The union of the number of cycles from the beginning and end of the paths for a given stage will simply be the minimum number encountered. The structural hazard information indicating the number of cycles from the end of path 1 has to be adjusted since its total number of cycles is 13 less than the cycles required by path 2. The resulting union of the structural hazard information of the two paths would be identical to the structural hazard information for path 2. Note that the data hazard information would change slightly since instruction 12 references register `%o0` as a source operand and `%o1` as both a source and destination. Yet, representing access to these registers would not likely have an effect when the timing analysis is performed between this path and its predecessor and successor paths since the EX stage is used before and after cycle 6, which is when instruction 12 enters the EX stage.

---

[7] Note that the generated assembly code has been optimized by the compiler. The local variables `i`, `count`, and `dcount` have been allocated to registers `%o2`, `%o1`, and `%f2`, respectively. The instruction following each transfer of control takes effect before the transfer of control is taken since the SPARC has delayed branches. The `cmp` comparison preceding the `bge` branch (instruction 7) has been moved to both immediately precede the loop and in the delay slot (instruction 16) of the `bl` branch (instruction 15). Branches with a "`,a`" represent that the result of the instruction within the delay slot will be annulled if the branch is not taken.

[8] Note instructions 7, 10, and 15 are transfers of control. The actual transfer of control (i.e. updating the program counter) occurs in the ID stage. Thus, there are no additional pipeline stages associated with these instructions. Also note the one cycle stall between instructions 8 and 12 in the EX stage of path 1 due to a load hazard. Finally, the `ldd` load (instruction 9) requires two cycles to complete the MEM stage [1].

```
C Source Code
-----------------------
main()
{
   int i, cnt = 0;
   double dcnt = 0.0;
   extern int incr;
   extern double dincr;

   for (i=0; i < 10;
        i++)
     if (i < 5)
        dcnt += dincr;
     else
        cnt += incr;
}
```

```
Inst      Assembly Code
----      -------------------------
 0        mov    %g0,%o1
 1        sethi  %hi(L01),%o0
 2        ldd    [%o0+%lo(L01)],%f2
 3        mov    %g0,%o2
 4        sethi  %hi(_dincr),%o3
 5        sethi  %hi(_incr),%o4
 6        cmp    %o2,5
 7  L8:   bge,a  L9
 8        ld     [%o4+%lo(_incr)],%o0
 9        ldd    [%o3+%lo(_dincr)],%f0
10        ba     L6
11        faddd  %f2,%f0,%f2
12  L9:   add    %o1,%o0,%o1
13  L6:   add    %o2,1,%o2
14        cmp    %o2,10
15        bl,a   L8
16        cmp    %o2,5
17        retl
18        nop
```

Figure 4. Example C source code and corresponding SPARC instructions.

Path 1 Instructions

```
inst  7: bge,a L19
inst  8: ld    [%o4+%lo(_incr)],%o0
inst 12: add   %o1,%o0,%o1
inst 13: add   %o2,1,%o2
inst 14: cmp   %o2,10
inst 15: bl,a  L18
inst 16: cmp   %o2,5
```

Path 1 Pipeline Diagram

stage \ cycle	1	2	3	4	5	6	7	8	9	10	11	12
IF	7	8	12	13	13	14	15	16				
ID		7	8	12	12	13	14	15	16			
EX				8		12	13	14		16		
FEX												
MEM					8		12	13	14		16	
WB						8		12	13	14		16
FWB												

Path 2 Instructions

```
inst  7: bge,a L19
inst  8: ld    [%o4+%lo(_incr)],%o0
inst  9: ldd   [%o3+%lo(_dincr)],%f0
inst 10: ba    L16
inst 11: faddd %f2,%f0,%f2
inst 13: add   %o2,1,%o2
inst 14: cmp   %o2,10
inst 15: bl,a  L18
inst 16: cmp   %o2,5
```

Path 2 Pipeline Diagram

stage \ cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...	24	25
IF	7	8	9	10	11	13	13	14	15	16							
ID		7	8	9	10	11	11	13	14	15	16						
EX				8	9				13	14		16					
FEX								11	11	11	11	11	11	11	...	11	
MEM					8	9	9			13	14		16				
WB						8					13	14		16			
FWB								9									11

Figure 5. Pipeline diagrams for the two paths through the loop in Figure 4.

Path 1 Info	IF	ID	EX	FEX	MEM	WB	FWB
Cycles from Beg	0	1	3	N/A	4	5	N/A
Cycles from End	4	3	2	N/A	1	0	N/A
Adj End Cycles	17	16	15	N/A	14	13	N/A
Path 2 Info	**IF**	**ID**	**EX**	**FEX**	**MEM**	**WB**	**FWB**
Cycles from Beg	0	1	3	7	4	5	7
Cycles from End	15	14	13	1	12	11	0

Table 3. Structural hazard information for the paths in Figure 5.

Let n be the maximum number of iterations associated with a loop. The algorithm for estimating the worst-case time for a loop is shown in Figure 6. The WHILE loop in the algorithm terminates when the number of calculated iterations reaches n - 1 or no more first misses (first hits) are encountered as misses (hits). Thus, the WHILE loop will either iterate (n - 1) or (m + 1), where m is the number of paths in the loop since a first miss (first hit) can miss (hit) at most once during the loop execution.

The algorithm selects the longest path on each iteration of the loop. 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 worst-case time than that calculated by the algorithm. Since the pipeline effects of each of the paths within the loop are unioned, it only remains to be shown that the caching effects are treated properly. The instruction fetch time used for each instruction depends on whether it is assumed to be a hit or miss, which depends on its categorization. The cache hit time is one cycle on most machines. The cache miss time is the cache hit time plus the miss penalty, which is the time required to access main memory. All categorizations are treated identically on repeated references, except for first misses and first hits. Assuming that the instructions have been categorized correctly for each loop and the pipeline analysis was correct, it remains to be shown that first misses and first

```
pipeline_information = NULL.
first_misses_encountered = NULL.
first_hits_encountered = NULL.
curr_iter = 0.
WHILE curr_iter != n - 1 DO
   curr_iter += 1.
   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.
   Concatenate pipeline_information with the
      union of the information for all paths.
   IF no new first misses or first hits
      are encountered in the path THEN
      break.
Concatenate pipeline_information with the union
   of the pipeline information for all paths
   (n - 1 - curr_iter) times.
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_information with the
      union of the information for all exit
      paths in the set.
   Store this information with the exit block
      for the loop.
```

Figure 6. Loop analysis algorithm.

hits are interpreted appropriately for a given iteration of the loop. A correctness argument about the interpretation of first hits and first misses is given in previous work [4].

Once no more first hit or first miss instructions are encountered that hit or miss respectively, the pipeline effects associated with the path chosen will not change since the caching behavior of the instructions within a path will always be treated the same. The pipeline effects of the last path are efficiently replicated for all but one of the remaining iterations. The last iteration of the loop is treated separately. The longest exit path for a loop may be shorter than the longest continue path. By examining the exit paths separately, a tighter estimate can be obtained. Thus, the algorithm estimates a bound that is at least as great as the actual worst-case bound.

5. Program analysis

A timing analysis tree is constructed to predict the worst-case times of code segments containing nested loops and function calls. Each node of the tree represents either a loop or a function in the function instance graph. Each node is considered a natural loop.[9] The nodes representing the outer level of function instances are treated as loops that will iterate only once when entered.

The loops in the timing analysis tree are processed in a bottom-up manner. In other words, the worst-case time for a loop is not calculated until the times for all of its immediate child loops are known. The algorithm given in the previous section described how a loop containing no other loops would be analyzed. The timing of a non-leaf loop is accomplished using this algorithm and the pipeline information and total times from its immediate child loops. Associated with each loop is a set of exit blocks, which indicates the possible blocks outside the loop that can be reached from the last block in each exit path. A unique set of timing information is stored for the child loop with each of these exit blocks. If a path within a loop enters a child loop, then the pipeline information and total time from the appropriate exit block are used at that point during the analysis of the path.[10]

The transition of an instruction categorization from the child loop level to the current loop level will be used to determine if any adjustment to the the child loop time is required. These transitions between categorizations requiring adjustments are described in Table 4. The *fm=>fm* adjustment is necessary since there should be only one miss associated with the instruction and a miss should only occur the first time the child loop is entered. The *m=>fh* adjustment is necessary since the first reference to the instruction in the outer loop will be a hit.

Child => Parent	Action to Adjust Child Loop Time
fm => fm	Use the child loop time for the first iteration. For all remaining iterations subtract the miss penalty from the child loop time.
m => fh	For the first iteration subtract the miss penalty from the child loop time. For all remaining iterations use the child loop time directly.

Table 4. Use of child loop times.

Making these adjustments when pipelining is involved resulted in some slight overestimations. The problem is that the caching behavior of a particular instruction depends on the loop level being analyzed. When an adjustment at an outer level would be needed for an instruction, the authors conservatively added the maximum number of cycles associated with a cache miss

[9] A natural loop is a loop with a single entry block. While the static simulator can process unnatural loops, the timing analyzer is restricted to only analyzing natural loops since it would be difficult for both the timing analyzer and the user to determine the set of possible blocks associated with a single iteration in an unnatural loop. It should be noted that unnatural loops occur quite infrequently.

[10] The timing analysis across loop levels is only briefly introduced in this section. It is described in more detail elsewhere [2], [4].

penalty to the total time of the path containing the instruction and treated the instruction fetch as a cache hit within the path pipeline analysis. When the instruction fetch should be viewed as a cache hit at an outer loop level, the previously added miss penalty cycles were subtracted from the loop's time. This strategy permitted a single pipeline analysis of each loop, yet adjustments could still be made at outer levels of the program. An overestimation occurs when the instruction fetch is treated as a miss and the cache miss penalty could be overlapped with the execution of other instructions or stalls (as shown in Figure 1). Fortunately, these adjustments are not that common. Results indicated that only about 4.5% of the instructions within the function instance graph were classified as first misses or first hits and many of these did not require adjustments. Thus, these adjustments resulted in only small and relatively infrequent overestimations.

6. Results

Measurements were obtained on code generated for the SPARC architecture by the *vpo* optimizing compiler [6]. Six simple programs described in Table 5 were used to assess the effectiveness of the timing analyzer. A direct-mapped instruction cache configuration containing 8 lines of 16 bytes was used. Thus, the cache contained 128 bytes of instructions. The programs were 4 to 17 times larger than the cache as shown in column 2 of Table 5. Column 3 shows the hit ratio of each program. Only *Matmul* had a very high ratio due to three tightly nested loops in a single function to perform the matrix multiplication. Each program was highly modularized to test the handling of timing predictions across function calls.

Name	Num Bytes	Hit Ratio	Description or Emphasis
Des	2,240	81.41%	Encrypts and Decrypts 64 Bits
Matcnt	812	81.81%	Counts and Sums Values in a 100x100 Matrix
Matmul	768	99.24%	Multiplies 2 50x50 Matrices
Matsum	644	88.22%	Sums Values in a 100x100 Matrix
Sort	556	83.99%	Bubblesort of 500 Numbers
Stats	1,428	88.41%	Calcs Sum, Mean, Var., StdDev., & Linear Corr. Coeff.

Table 5. Test programs.

The results of evaluating these programs are shown in Table 6. The observed cycles for these measurements were obtained by enhancing a traditional cache simulator [7]. The simulator produced the *pipeline only observed* cycles and the timing analyzer produced the *pipeline only estimated* cycles by assuming that all instruction fetches (IF stages) were cache hits and only required a single cycle. The *pipeline only naive* cycles were obtained by assuming that only a single pipeline stage could be executing at one time (i.e. no overlap). The *caching only observed* cycles and *caching only estimated* cycles were obtained with the assumption that the pipeline had only a single stage (an IF), a cache hit required a single cycle, and a cache miss required an additional miss penalty of nine cycles. The *naive caching only* cycles were calculated by assuming every instruction fetch resulted in a cache miss. The *pipeline and caching estimated* cycles were produced by the techniques that were described in this paper for integrating the analysis of pipelining and instruction caching behavior. All data cache references were assumed to be hits in the three sets of measurements.

Pipeline Only	Observed Cycles	Estimated Cycles	Estim. Ratio	Naive Ratio
Des	66,594	68,254	1.02	3.82
Matcnt	1,063,572	1,063,572	1.00	2.38
Matmul	4,347,806	4,347,806	1.00	2.13
Matsum	933,540	933,540	1.00	2.28
Sort	3,380,660	6,748,925	2.00	8.13
Stats	900,231	900,231	1.00	1.70
Caching Only	**Observed Cycles**	**Estimated Cycles**	**Estim. Ratio**	**Naive Ratio**
Des	142,956	163,015	1.14	3.86
Matcnt	1,169,055	1,259,055	1.08	3.79
Matmul	1,527,648	1,527,648	1.00	9.36
Matsum	707,219	707,219	1.00	4.85
Sort	7,639,611	15,253,902	2.00	8.17
Stats	372,410	372,410	1.00	4.90
Pipeline & Caching	**Observed Cycles**	**Estimated Cycles**	**Estim. Ratio**	**Naive Ratio**
Des	149,706	169,613	1.13	5.02
Matcnt	1,769,321	1,859,323	1.05	3.69
Matmul	4,444,911	4,445,413	1.00	4.98
Matsum	1,277,465	1,277,477	1.00	4.08
Sort	7,765,125	15,504,172	2.00	10.78
Stats	1,016,048	1,016,145	1.00	3.12

Table 6. Dynamic results for the test programs.

The *pipelining only* timing analysis had exact predictions for all programs except *Des* and *Sort*. The analysis of these two programs depicts problems faced by all timing analyzers. The timing analyzer did not accurately determine the worst-case paths in a function within *Des* primarily due to data dependencies. A longer path deemed feasible by the timing analyzer could not be taken in a function due to a variable's value in an `if` statement. The *Sort* program contains an inner loop whose number of iterations depends on the counter of an outer loop. At this point the timing tool either automatically receives the maximum loop iterations from the control-flow information produced by the compiler or requests a maximum number of iterations from the user. Yet, the tool would need a sequence of values representing the number of

iterations for each invocation of the inner loop. The number of iterations performed was overrepresented on average by a factor of two for this specific loop. Note that both of these problems are encountered by other timing tools and are not directly related to the pipeline analysis.

As reported previously [4], the *caching only* timing analysis results were also quite accurate. This analysis had exact predictions for *Matmul*, *Matsum*, and *Stats* since there were few conditional constructs except to exit loops. The *Matcnt* program used an `if-then-else` construct to either add a nonnegative value to a sum and increment a counter for the number of nonnegative elements or just increment a counter for the negative elements. The adding of the nonnegative value to a sum was accomplished in a separate function, which was purposely placed in a location that would *conflict* with the program line containing the code to increment a counter for the negative elements. Multiple executions of the `then` path, which includes the call to the function to perform the addition, still required more cycles than alternating between the two paths. Yet, the algorithm for estimating the worst-case instruction caching performance assumes that the first reference to a program line within a path would always be a miss if there were accesses to any other conflicting program lines within the same loop. This assumption simplified the algorithm since the effect of all combinations of paths need not be calculated. Thus, one reference was counted repeatedly as a miss instead of a hit. This path was executed 10,000 times and accounted for a 90,000 cycle [10,000*miss penalty] or an 8% overestimation. Note that the execution of this single path accounted for 40.61% of the total instructions referenced during the program execution. The programs *Des* and *Sort* had overestimations caused by the same problems described previously for the *pipeline only* measurements. The naive ratio was lower than initially anticipated by the authors. These test programs contained many long running instructions (floating-point operations and integer multiply and divides) that were frequently executed and often resulted in stalls. In addition, transfers of control were also quite frequent and were only considered to require two pipeline stages in our analysis.

The integrated *pipeline and caching* analysis also resulted in quite tight predictions. Again the predictions for the programs *Matmul*, *Matsum*, and *Stats* were very accurate. Note that the estimated cycles were slightly greater than the observed cycles for these programs. This overestimation was due to the problem of an instruction's caching behavior changing between loop levels. These changes require an adjustment as shown in Table 4. The approach used by the authors was to treat such an instruction as a hit in the pipeline analysis and simply add the miss penalty to the total time. When the instruction should be viewed as a hit at an outer level, then this miss penalty was simply subtracted and an accurate estimation is obtained. However, in these three programs the potential overlap between a miss penalty and a stall due to a hazard were not always detected.[11] The *Des*, *Matcnt*, and *Sort* programs had its usual overestimations due to data dependencies, a cache conflict, and an inaccurate number of estimated loop iterations, respectively. The naive ratio indicates that much tighter WCET bounds can be obtained when the benefits of pipelining and instruction caching are analyzed.

7. User interface

Once the initial timing analysis has been completed, a graphical user interface is invoked. This interface allows the user to quickly request timing predictions for functions, loops, paths, or subpaths via mouse clicks and reports the best and worst-case timing estimations. Whenever a different construct is selected, the highlighted lines in windows containing the source and assembly code are automatically updated and scrolled to the appropriate position. Thus, the user can quickly observe the relationship between timing constraints associated with the source code and sequences of machine instructions. This interface is described in more detail elsewhere [8].

8. Comparison with previous work

There has been much work on the issue of predicting execution time of programs. Most approaches in the past have not dealt with the effects of pipelining and instruction caching [9], [10], [11]. There have also been some recent studies on predicting pipeline performance by Harmon *et. al.* [12] and Narasimhan and Nilsen [13]. Yet, these studies did not address caching issues.[12] Furthermore, the former study was limited to nonnested functions and the latter study required the sequence of executed instructions to be known. Finally, there has been some recent work on predicting instruction caching performance. Arnold *et. al.* [4] implemented a timing analysis system to tightly bound instruction cache performance. However, this approach did not address pipelining issues.

There has been only one previous study that attempted to address the issue of predicting the WCET of programs on machines with both pipelining and an instruction

[11] For instance, the 502 cycle overestimation in *Matmul* occurred from 50 miss penalties completely overlapping with stalls from an integer multiply instruction and 52 misses overlapping with one cycle load hazards.

[12] Harmon assumed the entire code segment would fit into cache. Thus, he assumed at most one miss for each cache reference.

cache. Lim *et. al.* [14] described an method based on an extension of a previous timing tool [15]. Lim's method differs quite significantly from our approach described in this paper, which instead builds on flow analysis techniques found in optimizing compilers. Lim's method uses a timing schema associated with each source-level language program construct. They stored information about a predetermined number of cycles at the head and tail of a reservation table produced as a result of the pipeline analysis on the instructions associated with a program construct. In addition, this method stored information about the set of memory blocks whose first reference depends upon the cache contents prior to the execution of the construct. Lim also stored the set of memory blocks known to remain in cache after the execution of the construct. Eventually, this timing information is concatenated with another construct that would be executed immediately before the current construct. Their timing analyzer attempted to overlap the head of the reservation table of the current construct with the tail of the reservation table of the other construct as much as possible. Likewise, the list of memory blocks known to be in cache after executing the other construct is used to adjust the time of the current construct by comparing this list to the list of first reference blocks in the current construct. This method stored multiple paths for conditional constructs, such as an `if-then-else`. They pruned or eliminated a particular path when it was found that the worst-case execution time of the path was faster than the best-case execution time of another path within the same construct.

There are some limitations with Lim's method. The accuracy of their results is limited by the length of the head and tail of the reservation table stored with the program constructs. They concluded that the length of this head and tail only had to be large enough to contain information for five cycles. This conclusion was based on experiments indicating that their timing analysis results did not change significantly when the length was increased further. However, there are some instructions that require many cycles. For instance, a floating-point division on the MicroSPARC I can require up to 56 cycles to complete [1]. If such an instruction were at the end of a construct, then many more than five integer instructions at the head of a following construct could be overlapped with the floating-point division. In addition, their method stores information about each stage for every cycle in the head and tail of the reservation table. In contrast, our method requires much less information and imposes no limit on the length of the potential pipeline overlap. Only the relative distance from the beginning and end of the path has to be stored for each stage for the structural hazard pipeline information as shown by the numbers represented in boldface in Table 1.

The approach that Lim *et. al.* used to analyze caching behavior limits the accuracy of the analysis. They used a single bottom-up pass when performing the timing analysis of a program. The caching behavior of a large percentage of the instruction fetches within a construct would be unknown until many of the surrounding constructs were processed. Their approach was to treat the instruction fetch as a hit within the pipeline and add the cycles associated with a cache miss penalty to the total time of the construct. When it was later found that an instruction reference was a hit, they would subtract the miss penalty from the total time. However, an overestimation may result when the instruction is not found in cache. As shown in Figure 1, the instruction fetch miss penalty of one instruction (instruction 2) can be completely hidden by a stall with a long running instruction (data hazard stall on instruction 3). Whether the fetch of instruction 2 was a hit or a miss would have no effect on the total number of cycles. The Lim method would rarely detect instruction fetches that would always be misses until the surrounding constructs are analyzed, which is after the pipeline analysis of a construct has already occurred. Our approach of categorizing the caching behavior of each instruction before starting the timing analysis allows the detection of such situations. For instance, we found that the *pipeline and caching* estimated ratio for the six test programs increased on average by about 3% when the complete miss penalty was always added for each predicted miss.

9. Future work

We are working on several enhancements to the timing analyzer. An algorithm that predicts best-case pipelining and instruction cache performance is currently being implemented. We plan to automate the detection of many data dependencies using existing compiler optimization techniques to obtain tighter performance estimations [16]. The retargetability of the timing analyzer will also be enhanced by isolating any remaining machine dependent information in data files.

We are exploring methods to predict the timing of other architectural features associated with RISC processors. Work is currently ongoing to verify that our technique accurately predicts performance for the MicroSPARC I by using a logic analyzer. This will require predicting the performance of other features, such as wrap-around filling of cache lines. The effect of data caching is also an area that we are pursuing. Unlike instruction caching, many of the addresses of references to data can change during the execution of a program. Thus, obtaining reasonably tight bounds for worst-case and best-case data cache performance is significantly more challenging. However, many of the data references

are known. For instance, static or global data references retain the same addresses during the execution of a program. Due to the analysis of a function instance tree (no recursion allowed), addresses of run-time stack references can be statically determined even when the addresses may differ for different invocations of the same function. Compiler flow analysis can be used to detect the pattern of many calculated references, such as indexing through an array. While the benefits of using a data cache for real-time systems will probably not be as significant as using an instruction cache, its effect on performance should still be substantial.

10. Conclusions

This paper has presented a technique for predicting the worst-case execution time of programs on machines with pipelining and instruction caches. First, a static cache simulator analyzes the control flow of a program to statically categorize the caching behavior of each instruction within the program. Second, a timing analyzer uses these instruction categorizations when analyzing the pipeline performance of a path of instructions. Third, the timing analyzer uses a concise representation of the pipeline information to concatenate the performance of paths in an efficient manner when predicting the performance of loops. Fourth, a timing analysis tree is used to predict the performance of an entire program. Finally, a graphical user interface has been implemented that allows users to obtain timing predictions of portions of the program. The results indicate that the timing analyzer can quickly obtain tight predictions of WCET.

11. Acknowledgements

The authors thank Robert Arnold and Frank Mueller for providing advice and the platform for this research. The current timing analyzer, which includes pipeline analysis, was a direct extension of the previous timing analyzer implemented by Robert Arnold, which bounded instruction cache performance. The static simulator implemented by Frank Mueller was also used in this project. Lo Ko and Emily Ratliff implemented the user interface. The anonymous reviewers also provided helpful suggestions that improved the quality of the paper.

12. References

[1] Texas Instruments, Inc., *Product Preview of the TMS390S10 Integrated SPARC Processor*, 1993.

[2] F. Mueller, *Static Cache Simulation and Its Applications*, PhD Dissertation, Florida State University, Tallahassee, FL (August 1994).

[3] F. Mueller and D. Whalley, "Efficient On-the-fly Analysis of Program Behavior and Static Cache Simulation," *Static Analysis Symposium*, pp. 101-115 (September 1994).

[4] R. Arnold, F. Mueller, D. Whalley, and M. Harmon, "Bounding Worst-Case Instruction Cache Performance," *Proceedings of the Fifteenth IEEE Real-Time Systems Symposium*, pp. 172-181 (December 1994).

[5] F. Mueller and D. B. Whalley, "Fast Instruction Cache Analysis via Static Cache Simulation," *Proceedings of the 28th Annual Simulation Symposium*, pp. 105-114 (April 1995).

[6] M. E. Benitez and J. W. Davidson, "A Portable Global Optimizer and Linker," *Proceedings of the SIGPLAN '88 Symposium on Programming Language Design and Implementation*, pp. 329-338 (June 1988).

[7] J. W. Davidson and D. B. Whalley, "A Design Environment for Addressing Architecture and Compiler Interactions," *Microprocessors and Microsystems* **15**(9) pp. 459-472 (November 1991).

[8] L. Ko, D. B. Whalley, and M. G. Harmon, "Supporting User-Friendly Analysis of Timing Constraints," *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pp. 107-115 (June 1995).

[9] C. Y. Park, "Predicting Program Execution Times by Analyzing Static and Dynamic Program Paths," *Real-Time Systems* **5**(1) pp. 31-61 (March 1993).

[10] D. Niehaus, "Program Representation and Translation for Predictable Real-Time Systems," *Proceedings of the Twelfth IEEE Real-Time Systems Symposium*, pp. 53-63 (December 1991).

[11] P. Puschner and C. Koza, "Calculating the Maximum Execution Time of Real-Time Programs," *Real-Time Systems* **1**(2) pp. 159-176 (September 1989).

[12] M. G. Harmon, T. P. Baker, and D. B. Whalley, "A Retargetable Technique for Predicting Execution Time," *Proceedings of the Thirteenth IEEE Real-Time Systems Symposium*, pp. 68-77 (December 1992).

[13] K. Narasimhan and K. D. Nilsen, "Portable Execution Time Analysis for RISC Processors," *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, (June 1994).

[14] S. S. Lim, Y. H. Bae, G. T. Jang, B. D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Kim, "An Accurate Worst Case Timing Analysis Technique for RISC Processors," *Proceedings of the Fifteenth IEEE Real-Time Systems Symposium*, pp. 97-108 (December 1994).

[15] A. C. Shaw, "Reasoning about Time in Higher-Level Language Software," *IEEE Transactions on Software Engineering* **15**(7) pp. 875-889 (July 1989).

[16] F. Mueller and D. B. Whalley, "Avoiding Conditional Branches by Code Replication," *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pp. 56-66 (June 1995).