THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

PREDICTING PIPELINE AND INSTRUCTION CACHE PERFORMANCE

By

Christopher A. Healy

A Thesis submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Master of Science

Degree Awarded:
Fall Semester, 1995

The members of the Committee approve the thesis of Christopher A. Healy defended on October 17, 1995.

David B. Whalley
Professor Directing Thesis

Theodore P. Baker
Committee Member

Charles J. Kacmar
Committee Member

Gregory A. Riccardi
Committee Member

Approved:

R. C. Lacher, Chair, Department of Computer Science

CHAPTER 1

INTRODUCTION

Users of real-time systems are not only interested in obtaining correct computations from their programs, but timely responses as well. A program which gives a useful result past a deadline is not acceptable. Therefore, it is necessary to determine a program's execution time statically. It is unrealistic to attempt to predict a precise execution time for every real-time program since the execution time often depends upon input values whose influence on the program's control flow is unknown until the program executes. In addition, floating-point instructions usually vary in execution time based on the values of their operands. Consequently, instead of trying to derive a single execution time, a more pragmatic approach is to calculate upper (worst-case) and lower (best-case) bounds on the execution time. Real-time programmers tend to be more interested in the worst-case execution time (WCET) because of the notion of real-time deadlines. In other words, a task that completes too early is not as much of a concern as a task that finishes too late.

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. Yet, these same features introduce a potentially high level of unpredictability when it comes to establishing bounds on a program's execution time. Dependencies between instructions can cause pipeline

hazards that may delay the completion of instructions. While there has been much work analyzing the execution of a sequence of instructions within a basic block, the analysis of pipeline performance across basic blocks is more problematic. Instruction or data cache misses can also require several cycles to resolve. Predicting 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. In addition, caching and pipeline behavior are not independent, exacerbating the problem of timing analysis. Without the ability to predict instruction cache and pipeline performance simultaneously when calculating a WCET, it has been customary to be pessimistic, assuming that all instruction cache accesses would be misses and that pipeline data hazards would always give rise to additional execution delay. As an illustration, consider the following code segment and pipeline diagram in Fig. 1 consisting of three SPARC instructions. The pipeline cycles and stages represent the execution on a MicroSPARC I processor [1]. Each number within the pipeline diagram represents an instruction that is currently in the pipeline stage shown above it and occupies that stage during the cycle indicated to the left. Instruction 0 performs a floating-point addition that requires a total of twenty cycles. Fetching instruction 1 results in a cache miss, which is assumed to have a miss penalty of nine additional cycles.[1] Instruction 2 has a data dependency with instruction 0 and the execution of its CA stage is delayed until the floating-point addition is calculated.[2] The miss penalty associated with the access to main memory to fetch instruction 1 completely overlaps with the execution of the floating-point addition in

[1] The MicroSPARC I employs wrap-around filling upon a cache miss, so that the miss penalty actually depends on which word within the cache line the instruction belongs. See Chapter 7 for a discussion of this feature.

[2] Note that 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 CA stage on the MicroSPARC I.

SPARC Instructions

```
inst 0: faddd  %f2,%f0,%f2
inst 1: sub    %o4,%g1,%i2
inst 2: std    %f2,[%o0+8]
```

Pipeline Stage Abbreviations

IF	Instruction Fetch
ID	Instruction Decode
EX	integer EXecute
FEX	Floating-point EXecute
CA	data Cache Access
WB	integer Write Back
FWB	Floating-point Write Back

Pipeline Diagram

		stage						
		IF	ID	EX	FEX	CA	WB	FWB
cycle	1	**0**						
	2	1	**0**					
	3	1			**0**			
	4	1			0			
	5	1			0			
	...	...			...			
	11	1			0			
	12	**2**	1		0			
	13		**2**	**1**	0			
	14			2	0	**1**		
	15			2	0		**1**	
	16			2	0			
	17			2	0			
	18			2	0			
	19			**2**	**0**			
	20					2		**0**
	21					2		
	22					**2**		

Figure 1: Example of Overlapping Pipeline Stages with a Cache Miss

instruction 0. If the pipeline analysis and cache miss penalty were treated independently, then the number of estimated cycles associated with these instructions would be increased from 22 to 31 (i.e. by the cache miss penalty).

The remainder of the thesis will proceed as follows. Chapter 2 presents the context in which the timing analyzer operates with respect to its input/output and ancillary software. Chapter 3 explicates the algorithm for obtaining best-case and worst-case performance. Chapter 4 reports how well the timing analyzer predicts the performance of six benchmark programs. Chapter 5 discusses the role of the graphical user interface in communicating with the user. Chapter 6 examines related work in the area of predicting execution time. Chapter 7 describes future improvements planned for the timing analyzer, and Chapter 8 presents the conclusions.

CHAPTER 2

PREVIOUS WORK

The timing analyzer described in this thesis is part of a software package that has been under development by several researchers over the past few years. This package consists of an optimizing compiler called *vpo* [2], a static instruction cache simulator and a timing analyzer with a graphical user interface. Fig. 2 depicts an overview of the approach for predicting performance of large code segments on machines with pipelines and instruction caches.

Control-flow information, which could have also been obtained by analyzing assembly or object files, is stored as the side effect of *vpo*'s compilation of one or more C source files. This control-flow information is passed to the static cache simulator, which ultimately categorizes each instruction's potential caching behavior based on a

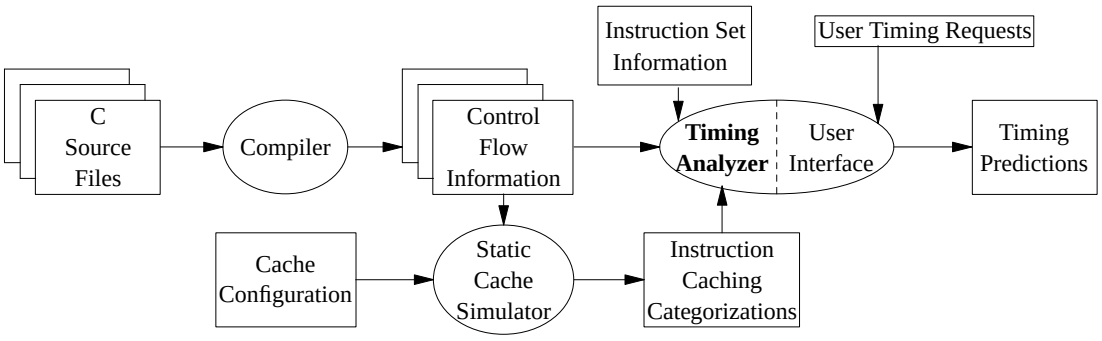

Figure 2: Overview of Bounding Pipeline and Cache Performance

given cache configuration. The caching behavior of an instruction is assigned one of four categories, described in Tables 1 and 2, for each loop level in which an instruction is contained. The theory and implementation of static cache simulation is described in more detail elsewhere [3, 4, 5, 6]. The timing analyzer uses the instruction caching

Table 1: Definitions of Worst-Case Instruction Categories

Instruction Category	Definition According to Behavior in Instruction Cache
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 executed, but is guaranteed 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.

Table 2: Definitions of Best-Case Instruction Categories

Instruction Category	Definition According to Behavior in Instruction Cache
always miss	The instruction is guaranteed to not be in cache when it is referenced.
always hit	It is possible that the instruction is in cache every time it is referenced.
first miss	The instruction is guaranteed to not be in cache on its first reference each time the loop is executed, but may be in cache on subsequent references.
first hit	The instruction may be in cache on its first reference each time the loop is executed, but is guaranteed to not be in cache on subsequent references.

categorizations to determine whether an instruction fetch should be treated as a hit or a miss during the pipeline analysis of a path. The timing analyzer also reads a file that specifies the hardware's instruction set pipeline constraints in order to detect structural and data hazards between instructions. Given a program's control-flow information and instruction caching categorizations along with the processor's instruction set information, the timing analyzer then derives best-case and worst-case estimates for each loop and function within the program. This version of the timing analyzer is an extension of an earlier timing tool [5, 7] which bounded instruction cache performance. Although most machines that have an instruction cache also have a data cache, the timing analyzer does not as yet predict data cache performance. When the timing analyzer has completed its analysis, it invokes a graphical user interface [8] allowing the user to request timing bounds for portions of the program. Excerpts of this thesis, including a concise description of the algorithm and worst-case results, can be found in [9].

CHAPTER 3

TIMING ANALYSIS

Several steps are necessary to obtain the timing predictions of a program. The optimizing compiler *vpo* determines control-flow information. Next, the static cache simulator predicts the caching behavior of each assembly instruction according to the program's control-flow. The timing analyzer also uses the control-flow information to determine the set of paths through each loop and function [5]. Once this information has been computed, the timing analyzer turns its attention to predicting the BCET and WCET.

The timing analyzer determines execution time for programs by first analyzing the innermost loops and functions, and proceeding to higher level loops and functions until it reaches `main()`. For example, consider the skeleton program in Fig. 3. The timing analyzer will establish best- and worst-case time bounds of `fun2()`, `loop_2`, `loop_1`, `fun1()`, `fun2()` and finally `main()`. Note that `fun2()` needs to be analyzed twice since it is called from two different places. The pipeline and caching behavior of the two invocations of `fun2()` are likely to differ. For example, if an instruction *i* in `fun2()` contains an assembly instruction that maps to the same cache line as an instruction involving the `++j` operation in `loop_2`, instruction *i* will always be a miss in cache as long as `fun2()` is invoked from inside `loop_2`. On the other hand, instruction *i* may still be a hit when `fun2()` is called from `main()`. The timing

```
void fun1()
{
  int i, j;

  for (i = 0; i < 100; ++i)       /* loop_1 : outer loop */
    for (j = 0; j < 100; ++j) {   /* loop_2 : inner loop */
      fun2();
    }
}
void fun2()
{
  /* body of function */
}
main()
{
  fun1();
  fun2();
}
```

Figure 3: A Skeleton Program

analyzer treats a function as a loop that only executes for a single iteration. Hereafter, a *loop* being analyzed will refer to either a *loop*[3] or a *function* within the program.

3.1 Analyzing A Single Path of Instructions

Before the timing analyzer examines the program, it reads information from a machine-dependent file concerning the pipeline requirements of each instruction in the processor's instruction set. This information includes how many cycles each instruction spends in each pipeline stage. For floating-point instructions, the number of cycles spent in the FEX can vary significantly depending upon the values of the register operands. For instance, the double-precision divide instruction `fdivd` (which is distinct from the single precision divide instruction `fdivs`) can take as few as six cycles in the FEX or as

[3] In this thesis, *loops* will be restricted to natural loops. 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 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.

many as 56 cycles. Thus, a floating-point intensive program tends to have a wider difference between best-case and worst-case execution times than a program with only integer instructions. The timing analyzer also obtains from this file the latest pipeline stage in which the values of its register operands are required via forwarding for each instruction to proceed, and in which stage the value of the destination register is available via forwarding.

The control-flow information that *vpo* provides also identifies the register operands of each instruction in the program. As mentioned before, the static cache simulator categorizes each instruction's expected caching behavior. Based on an instruction's categorization, the timing analyzer can decide whether the instruction will be treated as a hit or miss in the pipeline. When an instruction is a hit in the pipeline, it will spend one cycle in the IF stage, possibly more if it cannot immediately proceed to the ID due to a stall. When an instruction is treated as a miss in the pipeline, it will spend the duration of the miss penalty in the IF stage in addition to the single cycle it would have occupied the IF had the instruction been a hit. Even if it is a miss in cache, the instruction may spend more than ten cycles in the IF stage if there is a stall. For example, a double-precision floating-point divide instruction `fdivd` may spend up to 56 cycles in the FEX stage. If the `fdivd` is followed by two instructions, the first of which being another floating-point instruction that is a cache hit and the second of which that is cache miss, then there will be a structural hazard between the `fdivd` instruction and the floating-point instruction following it. As a result, while the `fdivd` instruction is occupying the FEX for 56 cycles, the second instruction after it will spend the same 56 cycles in the IF stage. In this case, the cache miss is overlapped in time with the structural hazard.

A path of instructions consists of all the instructions that can be executed during a

single iteration of a loop (or in the case of a function, all the instructions that are executed in one invocation of the function). If the loop has no conditional (e.g. `if` or `switch`) statements, then there will be only one path associated with this loop. As an example, consider the function `Square()` in Fig. 4. This function contains seven instructions, numbered from 0 through 6, that comprise one path. Instructions 0 and 1 are classified as *always misses*, and for this reason they must each spend ten cycles in the IF stage before proceeding to the other pipeline stages. Instruction 1 is a store instruction `st`, which must spend two cycles in the CA stage. This pipeline requirement results in a structural hazard since instruction 2 is ready to enter the CA stage in cycle 24 but cannot do so until instruction 1 vacates it. Thus, instruction 1 causes instructions 2, 3 and 4 to stall in the EX, ID and IF stages, respectively, during cycle 24. A similar structural hazard occurs during cycle 26 when instruction 2, another store instruction, occupies the CA stage for two cycles. Later, during cycles 27 and 28, a data hazard takes place between instructions 3 and 4. Instruction 3 loads the value of register `%f2` which is instruction 4's source operand. This means that instruction 4 cannot enter the FEX stage until instruction 3 leaves the CA stage. Finally, instruction 4 must spend seven cycles in the FEX stage, not due to any cache miss or pipeline hazard, but because of the hardware's pipeline requirement of the `fmultd` instruction.

The analyzer examines the instructions sequentially. It keeps track of the number of cycles required to execute the path up to the instruction currently being processed, plus pipeline information regarding the beginning and ending behavior of the path. Tables 3 and 4 depict how this pipeline information is gradually modified as the analyzer processes each instruction in `Square()`. The first row of each table shows the pipeline information after only instruction 0 has been processed, the second row shows the

C Source Code

```
double Square(x)
double x;
{
  return x * x;
}
```

SPARC Instructions

```
inst 0: save   %sp,(-72),%sp
inst 1: st     %i0,[%sp+.4_x]
inst 2: st     %i1,[%sp+(.4_x+4)]
inst 3: ldd    [%sp+.4_x],%f2
inst 4: fmuld  %f2,%f2,%f0
inst 5: ret
inst 6: restore
```

Pipeline Diagram

		stage						
		IF	ID	EX	FEX	CA	WB	FWB
cycle	1	**0**						
	...	...						
	10	0						
	11	1	**0**					
	12	1		**0**				
	13	1				**0**		
	14	1					**0**	
	15	1						
	...	...						
	20	1						
	21	2	1					
	22	3	2	1				
	23	4	3	2		1		
	24	4	3	2		1		
	25	5	4	3		2		
	26	5	4	3		2		
	27	5	4			3		
	28	5	4			3		
	29	**6**	5		**4**			**3**
	30		**6**		4			
	31			**6**	4			
	32				4	**6**		
	33				4		**6**	
	34				4			
	35				**4**			
	36							**4**

Figure 4: Path through Function `Square()`

pipeline information taking into account instructions 0 and 1, and so on. The last row depicts the the pipeline behavior of the entire path. The values in the rows labeled Cycles from Beg in Table 3 represent how many cycles *after cycle 1* that particular stage is first occupied. The values in the rows labeled Cycles from End in Table 4 represent how many cycles *before the last cycle* (which is given in the rightmost column) that stage is last occupied. To determine during which cycle an instruction completed its occupation of a particular stage, one subtracts the Cycles from End value from the total

Table 3: Creating Beginning Pipeline Information for `Square()`

Inst	Stage	IF	ID	EX	FEX	CA	WB	FWB
0	Cycles from Beg Beg Occupant	0 0	10 0	11 0	N/A N/A	12 0	13 0	N/A N/A
1	Cycles from Beg Beg Occupant	0 0	10 0	11 0	N/A N/A	12 0	13 0	N/A N/A
2	Cycles from Beg Beg Occupant	0 0	10 0	11 0	N/A N/A	12 0	13 0	N/A N/A
3	Cycles from Beg Beg Occupant	0 0	10 0	11 0	N/A N/A	12 0	13 0	28 3
4	Cycles from Beg Beg Occupant	0 0	10 0	11 0	28 4	12 0	13 0	28 3
5	Cycles from Beg Beg Occupant	0 0	10 0	11 0	28 4	12 0	13 0	28 3
6	Cycles from Beg Beg Occupant	0 0	10 0	11 0	28 4	12 0	13 0	28 3

Table 4: Creating Ending Pipeline Information for `Square()`

Inst	Stage	IF	ID	EX	FEX	CA	WB	FWB	total cycles
0	Cycles from End End Occupant	4 0	3 0	2 0	N/A N/A	1 0	0 0	N/A N/A	14
1	Cycles from End End Occupant	4 1	3 1	2 1	N/A N/A	0 1	10 0	N/A N/A	24
2	Cycles from End End Occupant	5 2	4 2	2 2	N/A N/A	0 2	12 0	N/A N/A	26
3	Cycles from End End Occupant	7 3	5 3	3 3	N/A N/A	1 3	14 0	0 3	29
4	Cycles from End End Occupant	12 4	8 4	10 3	1 4	8 3	22 0	0 4	36
5	Cycles from End End Occupant	8 5	7 5	10 5	1 4	8 3	22 0	0 4	36
6	Cycles from End End Occupant	7 6	6 6	5 6	1 4	4 6	3 6	0 4	36

cycles value in the same row. For example, the first row of Table 4 says that if the path consisted solely of instruction 0, the total cycle time for the path would be fourteen cycles, according to the rightmost column. It also states that instruction 0 finishes the IF stage four cycles before the end of the path. Since 14 - 4 = 10, instruction 0 finishes its

occupation of the IF stage during cycle 10. The second row of Table 4 refers to the path if it only consisted of instructions 0 and 1. In this case the path's total time is 24 cycles, as given in the rightmost column, and the WB stage is last occupied 10 cycles before the final stage, as given in the third column from the right. Subtracting these two figures gives 24 - 10 = 14, meaning that the WB stage is last occupied during cycle 14. Table 4 indicates further that the last occupant of the WB stage was instruction 0, which agrees with the pipeline diagram in Fig. 4.

The beginning pipeline information, as given in Table 3, is not immediately relevant for the timing analysis of the function `Square()`. Its role comes into play when the timing analyzer proceeds to the analysis of an entire *loop*, as described in the next section. For path analysis, the ending pipeline information is necessary for the avoidance of structural hazards. The beginning and ending occupants of the stages are not needed for the timing analysis, but are provided here for clarity. Table 5 shows information about the register operands whose values are needed and/or set by the instructions. This register information is needed to detect data hazards. Figures in the rows labeled first needed show how many cycles *after cycle 1* that particular register's value is required as a source operand. Figures in the rows labeled last produced count how many cycles *before the last cycle* that register's value is available.

Retaining this set of pipeline information allows additions to the beginning or end of a path. Since the pipeline requirements for a path and for a single instruction can both 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 of two sets of pipeline information is accomplished one stage

Table 5: Data Hazard Information for the Instructions in `Square()`

Inst	Register	...	%o6	...	%i0	%i1	...	%f0	...	%f2	...
0	first needed	N/A	10	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	last produced	N/A	2	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
1	first needed	N/A	10	N/A	21	N/A	N/A	N/A	N/A	N/A	N/A
	last produced	N/A	12	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
2	first needed	N/A	10	N/A	21	22	N/A	N/A	N/A	N/A	N/A
	last produced	N/A	14	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
3	first needed	N/A	10	N/A	21	22	N/A	N/A	N/A	N/A	N/A
	last produced	N/A	17	N/A	N/A	N/A	N/A	N/A	N/A	1	N/A
4	first needed	N/A	10	N/A	21	22	N/A	N/A	N/A	27	N/A
	last produced	N/A	24	N/A	N/A	N/A	N/A	1	N/A	8	N/A
5	first needed	N/A	10	N/A	21	22	N/A	N/A	N/A	27	N/A
	last produced	N/A	24	N/A	N/A	N/A	N/A	1	N/A	8	N/A
6	first needed	N/A	10	N/A	21	22	N/A	N/A	N/A	27	N/A
	last produced	N/A	24	N/A	N/A	N/A	N/A	1	N/A	8	N/A

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 Fig. 4 could not be placed in cycle 20 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 instruction in that stage. For example, the beginning of the FEX stage for instruction 4 in Fig. 4 must take place after instruction 3 finishes its CA stage due to the data hazard between the `ldd` and `fmuld` instructions.

(3) The placement of the instruction does not violate its own pipeline requirements. For instance, in Fig. 4 the ID stage of instruction 1 has to occur at least eleven cycles after the beginning of its IF stage.

Data and structural hazards can also occur upon entering and leaving a child loop. For instance, if `Square()` in Fig. 4 is invoked from another function, and the instruction that is executed after returning from `Square()` has `%f0` as a source operand, then it will have a data dependency with instruction 4 of `Square()`. The timing analyzer can

detect this potential hazard in much the same manner as though `Square()` were a single instruction in the calling function's path.

After the beginning and ending pipeline behavior of a path has been determined, 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 seven instructions in Fig. 4. 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 0 or instruction 6, 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.

3.2 Loop Analysis

To find the BCET and WCET for a loop, the timing analyzer must first evaluate all of the possible paths through the loop.

3.2.1 The Union Concept

With pipelining it is possible that the combination of a set of paths may produce a longer execution time than just repeatedly selecting the longest path. For instance, consider a loop with two paths that take about the same number of cycles to execute. Path 1 has a `fdivd` instruction near its beginning and path 2 has a `fdivd` instruction near its end. Alternating between the paths will produce the WCET since there will be a structural hazard between the two instructions when path 1's `fdivd` occurs shortly after path 2's

`fdivd`.

To avoid the problem of calculating all combinations of paths, which would be the only method for obtaining perfectly accurate estimations, the timing analyzer determines the *union* of possible pipeline effects of the paths for an iteration of a loop. This simplifies the algorithm and also does not cause any noticeable overestimation or underestimation. Since all paths through a loop must begin with the same header block, the beginning pipeline information among the various paths is usually the same. Also, paths often end with the same block of instructions, so that ending pipeline information is unaffected by the process of uniting the pipeline information. However, beginning and ending pipeline information can significantly differ when one path consists exclusively of integer instructions while another contains floating-point instructions. This situation occurs in a simple program depicted in Figs. 5 and 6.

The generated assembly code has been optimized by *vpo*. The local variables `i`, `count` and `fcount` have been allocated to registers `%o3`, `%o2` and `%f1` respectively. Since the SPARC has delayed branches, the instruction following each transfer of control takes effect before the branch is taken. The loop in this program consists of instructions 10 through 27. *Vpo* has replicated instruction 9, the comparison, to also appear in the delay slot at the end of the loop, instruction 27. A branch instruction ending in "`,a`" is an annulled branch, meaning that the result of the instruction in the delay slot will be annulled if the branch is not taken.

To simplify this example, all of the instructions and data are assumed to already be in cache. Table 6 shows the structural hazard information corresponding to the two paths in Fig. 6, and Fig. 7 depicts the pipeline diagrams for the worst-case and best-case unions of the two paths as a visual representation of the values contained in the bottom half of

```
      C Source Code           Inst     Assembly Code
------------------------     ----     ----------------------------
main()                         0      mov    %g0,%o2
{                              1      sethi  %hi(L01),%o0
  int i;                       2      ldd    [%o0+%lo(L01)],%f1
  int count = 0;               3      add    %o1,%o1,%o1
  float fcount = 0;            4      add    %o1,1,%o1
  extern int incr;             5      sub    %o2,%o1,%o2
  extern float fincr;          6      mov    %g0,%o3
                               7      sethi  %hi(_fincr),%o4
  count -= i + i + 1;          8      sethi  %hi(_incr),%o5
                               9      cmp    %o3,5
  for (i = 0; i < 10; ++i)    10 L18: bge,a  L19
  {                           11      sub    %o3,%o2,%o1
    if (i < 5)                12      add    %o2,1,%o2
    {                         13      ld     [%o4+%lo(_fincr)],%f0
      ++count;                14      ba     L16
      fcount *= fincr;        15      fmuls  %f1,%f0,%f1
    }                         16 L19: add    %o1,1,%o1
    else                      17      ld     [%o5+%lo(_incr)],%o0
    {                         18      sub    %o0,%o1,%o0
      incr -= i - count + 1;  19      add    %o3,%o2,%o1
      incr += i + count - 2;  20      sub    %o1,2,%o1
      count += incr;          21      add    %o0,%o1,%o0
    }                         22      st     %o0,[%o5+%lo(_incr)]
  }                           23      add    %o2,%o0,%o2
}                             24 L16: add    %o3,1,%o3
                              25      cmp    %o3,10
                              26      bl,a   L18
                              27      cmp    %o3,5
                              28      retl
                              29      nop
```

Figure 5: Program Containing a Loop with Two Paths

Table 6. Fig. 7 shows how little information is used to store the union, as opposed to the pipeline diagrams in Fig. 6. It is only necessary to know when each stage is first and last occupied. Some additional information concerning the occupancy of the stages is also calculated during best-case analysis, and this will be discussed in Section 3.2.4. To calculate the union of the paths during worst-case analysis, one finds the *earliest* initial occupation (relative to cycle 1) and *last* finishing occupation (relative to the last cycle of the longest path) of each stage. As Fig. 6 shows, the corresponding instructions in both paths in this example begin the IF, ID, EX, CA and WB stages at the same time. Since Path 1 never occupies the FEX or the FWB stages, the worst-case union will store the

Path 1 Instructions

```
inst 10: bge,a L19
inst 11: sub   %o3,%o2, %o1
inst 16: add   %o1,1,%o1
inst 17: ld    [%o5+%lo(_incr)],%o0
inst 18: sub   %o0,%o1,%o0
inst 19: add   %o2,%o0,%o2
inst 20: sub   %o1,2,%o1
inst 21: add   %o0,%o1,%o0
inst 22: st    %o0,[%o5+%lo(_incr)]
inst 23: add   %o2,%o0,%o2
inst 24: add   %o3,1,%o3
inst 25: cmp   %o3,10
inst 26: bl,a  L18
inst 27: cmp   %o3,5
```

Path 2 Instructions

```
inst 10: bge,a L19
inst 11: sub   %o3,%o2,%o1
inst 12: add   %o2,1,%o2
inst 13: ld    [%o4+%lo(_fincr)],%f0
inst 14: ba    L16
inst 15: fmuls %f1,%f0,%f1
inst 24: add   %o3,1,%o3
inst 25: cmp   %o3,10
inst 26: bl,a  L18
inst 27: cmp   %o3,5
```

Path 1 Pipeline Diagram

		stage						
		IF	ID	EX	FEX	CA	WB	FWB
cycle	1	**10**						
	2	11	**10**					
	3	16	11					
	4	17	16	**11**				
	5	18	17	16		**11**		
	6	19	18	17		16	**11**	
	7	19	18			17	16	
	8	20	19	18			17	
	9	21	20	19		18		
	10	22	21	20		19	18	
	11	23	22	21		20	19	
	12	24	23	22		21	20	
	13	25	24	23		22	21	
	14	25	24	23		22		
	15	26	25	24		23		
	16	**27**	26	25		24	23	
	17		**27**			25	24	
	18			**27**			25	
	19					**27**		
	20						**27**	
	21							
	...							
	33							

Path 2 Pipeline Diagram

		stage						
		IF	ID	EX	FEX	CA	WB	FWB
cycle	1	**10**						
	2	11	**10**					
	3	12	11					
	4	13	12	**11**				
	5	14	13	12		**11**		
	6	15	14	13		12	**11**	
	7	24	15			13	12	
	8	25	24		**15**			**13**
	9	26	25	24	15			
	10	**27**	26	25	15	24		
	11		**27**		15	25	24	
	12			**27**	15		25	
	13				15	**27**		
	14				15		**27**	
	15				15			
	...				...			
	32				**15**			
	33							**15**

Figure 6: Pipeline Diagrams for the Two Loop Paths in Fig. 5

Worst-Case Union Pipeline Diagram

		stage						
		IF	ID	EX	FEX	CA	WB	FWB
cycle	1	**10**						
	2		**10**					
	3							
	4			**11**				
	5					**11**		
	6						**11**	
	7							
	8				**15**			**13**
	9							
	10							
	11							
	12							
	13							
	14							
	15							
	16	**27**						
	17		**27**					
	18			**27**				
	19					**27**		
	20						**27**	
	...							
	32				**15**			
	33							**15**

Best-Case Union Pipeline Diagram

		stage						
		IF	ID	EX	FEX	CA	WB	FWB
cycle	1	**10**						
	2		**10**					
	3							
	4			**11**				
	5					**11**		
	6						**11**	
	7							
	8							
	9							
	10	**27**						
	11		**27**					
	12			**27**				
	13					**27**		
	14						**27**	
	15							
	16							
	17							
	18							
	19							
	20							
	...							
	32							
	33							

Figure 7: Pipeline Diagrams for the Worst-Case and Best-Case Unions

Table 6: Structural Hazard Information for Unions in Fig. 7

Path 1 Info	IF	ID	EX	FEX	CA	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
Path 2 Info	**IF**	**ID**	**EX**	**FEX**	**CA**	**WB**	**FWB**
Cycles from Beg	0	1	3	7	4	5	7
Cycles from End	23	22	21	1	20	19	0
WC Union Info	**IF**	**ID**	**EX**	**FEX**	**CA**	**WB**	**FWB**
Cycles from Beg	0	1	3	7	4	5	7
Cycles from End	17	16	15	1	14	13	0
BC Union Info	**IF**	**ID**	**EX**	**FEX**	**CA**	**WB**	**FWB**
Cycles from Beg	0	1	3	N/A	4	5	N/A
Cycles from End	23	22	21	N/A	20	19	N/A

beginning (as well as the finishing) times of Path 2. The two paths differ with respect to the completion of the integer pipeline stages. For instance, in Path 1, instruction 27 finishes the IF stage during cycle 16, seventeen cycles before the last executing cycle of the longer path (Path 2). On the other hand, in Path 2, instruction 27 leaves the IF stage 23 cycles before the path completes. Since the worst-case union seeks the latest possible ending times of the stages, it will adopt Path 1's finishing times for the integer stages. Thus, as Fig. 7 shows, the worst-case union's pipeline finishes the integer stages during cycles 16, 17, 18, 19 and 20, respectively, and when these values are measured from the last cycle of Path 2, one subtracts each from 33 to obtain the values in the row labeled Cycles from End under the WC Union Info heading in Table 6.

During best-case analysis, it is necessary to modify the union calculation in order to avoid detection of structural and data hazards that would cause an overestimation in the BCET. In best case, the timing analyzer finds the *latest* initial occupation and *earliest* finishing occupation of each stage. If a stage is not occupied in one of the paths, then in best case that stage will not be occupied by the union. In Fig. 7 and in Table 6, the FEX and FWB stages of the best-case union are vacant because Path 1 contains no floating-point instructions.

3.2.2 Worst-Case Algorithm for Timing a Loop

The algorithm for estimating the WCET for a loop is given in Fig. 8, where *n* is the maximum number of iterations associated with the loop. This algorithm is based on an earlier method for determining the WCET of a loop in the context of predicting instruction cache performance. A correctness argument for it is given in [7]. The WHILE loop in the algorithm terminates when the number of calculated iterations

```
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
      worst-case 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 worst-case
   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
      worst-case union of the information for all
      exit paths in the set.
   Store this information with the exit block
      for the loop.
```

Figure 8: Worst-Case Loop Analysis Algorithm

reaches n - 1 or no more first misses (first hits) are counted as misses (hits). If p is the number of paths in the loop, then the WHILE loop in the algorithm will iterate up to (n - 1) or (p + 1) times, since a first miss (first hit) can miss (hit) at most once during the loop execution.

The goal of the algorithm in Fig. 8 is to select the longest path on each iteration of the loop. As an illustration, consider the program shown in Fig. 5, which contains a loop. The two paths of the loop are described in Fig. 6. Path 1 contains only integer

instructions while Path 2 contains a floating-point multiply instruction. The static cache simulator has determined that instructions 12, 16, 20 and 24 are first misses while the rest of the instructions in the loop are always hits.

If Path 1 executes during the first iteration of the loop, it will take twenty cycles, according to the pipeline diagram in Fig. 6, plus the combined miss penalties of its first misses, instructions 16, 20 and 24. Thus, its time would be 20 + 9*3 = 47 cycles. If Path 2 executes during the first iteration of the loop, it will take 33 cycles, plus another eighteen taking into account its first misses, instructions 12 and 24, for a total of 51 cycles. Therefore, Path 2 has a longer worst-case time for the loop's first iteration.

When the timing analyzer prepares to examine the second iteration of the loop, Path 2's first misses will hereafter be hits in cache. If Path 1 executes during the second iteration of the loop, it will take sixteen cycles, plus the miss penalties for its two remaining first misses, instructions 16 and 20, yielding a total of 34 cycles. If Path 2 executes on the second iteration of the loop, it will take 26 cycles with no additional miss penalty. Hence Path 1 is the worst-case path for the second iteration of the loop.

By the third iteration of the loop, all of the first misses have been encountered, so they will be treated as hits in the pipeline for each of the remaining iterations. If Path 1 executes on the third iteration of the loop, it will take sixteen additional cycles, while Path 2 would require 29 additional cycles. These execution times are four cycles fewer than the pipeline diagrams in Fig. 6 indicate because the pipeline is already filled. Since Path 2 has a longer execution time than Path 1, the timing analyzer will choose Path 2 for the third iteration of the loop. No new first misses or first hits are encountered during the third iteration, so the timing analyzer will exit the WHILE loop in the worst-case algorithm in Fig. 8. The next phase of the algorithm uses Path 2 for all the remaining

iterations before the last one. When an iteration involving Path 2 follows a previous iteration executing Path 2 as well, the FEX stage of instruction 15 overlaps with more integer instructions than if Path 2 had followed an iteration traversing Path 1. As a result, Path 2's execution time for each iteration from the fourth through the ninth is 26 cycles.

The last iteration of the loop is treated separately. The timing analyzer uses the path that will cause the final iteration to have the longest WCET. Because exit paths can be distinct from continue paths, an exit path may contain a first miss instruction that has not been encountered in any selected continue path, and such a first miss will not be encountered until the last iteration. However, for the loop in Fig. 6, the exit paths are identical to the continue paths, so its last iteration will follow Path 2, and this iteration will contribute another 26 cycles to the loop's WCET, precisely as with each of the previous six iterations. In summary, the WCET for the entire execution of the loop is 51 + 34 + 29 + 6*26 + 26 = 296 cycles.

However, it is possible that the longest exit path for a loop is shorter than the longest continue path. Consider the program *Exit*, depicted in Figs. 9 and 10. The final iteration of the loop does not perform the floating-point division. Had the timing analyzer not distinguished between continue and exit paths, the continue path containing the `fdivd` would have been the worst-case path on every iteration, yielding an execution time of 87 cycles on the first iteration plus 83 cycles for each of the remaining four iterations,[4] totaling 419 cycles. Instead, the fifth iteration should only take only three additional cycles for a total of 339 cycles. Thus, the analysis avoids an unnecessarily large

[4] The first iteration takes four more cycles to execute because the timing analyzer assumes the pipeline is initially empty.

```
      C Source Code
-------------------------
main()
{
  double a[5];
  int i;

  for (i = 0;  ; i++)
  {
    if (i == 4)
      break;
    else
      a[i] = i / 2.0;
  }
}
```

```
Inst     Assembly Code
----    --------------------------
  0      save   %sp,(-112),%sp
  1      mov    %g0,%o2
  2      sethi  %hi(L01),%o3
  3      add    %sp,.1_a,%o4
  4      mov    %o4,%g1
  5     add    %o4,32,%g2
  6 L17: cmp    %g1,%g2
  7      bne,a  L18
  8      st     %o2,[%sp + xfer_1]
  9      ret
 10     restore
 11 L18: ldd    [%o3 + %lo(L01)],%f0
 12      ld     [%sp + xfer_1],%f2
 13      fitod  %f2,%f2
 14      fdivd  %f2,%f0,%f2
 15      std    %f2,[%g1]
 16      add    %g1,8,%g1
 17      add    %o2,1,%o2
 18      ba     L17
 19      nop
```

Figure 9: C Source Code and Assembly Code for the Program *Exit*

Continue Path Pipeline Diagram

		stage						
		IF	ID	EX	FEX	CA	WB	FWB
cycle	1	**6**						
	2	7	**6**					
	3	8	7	**6**				
	4	11	8			**6**		
	5	12	11	8			**6**	
	6	13	12	11		8		
	7	13	12	11		8		
	8	14	13	12		11		
	9	14	13	12		11		
	10	14	13			12		**11**
	11	15	14		**13**			12
	12	15	14		13			
	...	15	14		13			
	23	15	14		13			
	24	16	15		14			13
	25	17	16	15	14			
	...	17	16	15	14			
	79	17	16	15	**14**			
	80	18	17	16		15		**14**
	81	18	17	16		15		
	82	18	17	16		15		
	83	**19**	18	17		16		
	84		**19**			17	16	
	85			**19**			17	
	86					**19**		
	87						**19**	

Exit Path Pipeline Diagram

		stage						
		IF	ID	EX	FEX	CA	WB	FWB
cycle	1	**6**						
	2	7	**6**					
	3	**8**	7	**6**				
	4		**8**			**6**		
	5			**8**			**6**	
	6					8		
	7					**8**		

Figure 10: Pipeline diagrams for the Two Paths through Loop in *Exit*

overestimation of 80 cycles on this simple loop.

3.2.3 Best-Case Algorithm for Timing a Loop

The best-case algorithm given in Fig. 11 is somewhat simpler than the worst-case one described in the previous section. It is also based on the best-case strategy for caching only analysis [5]. Here, *n* represents the minimum number of iterations associated with the loop. The algorithm calculates the BCET for the first iteration, for the first *n* - 1 iterations and finally for all *n* iterations. In the case of a function, the BCET is calculated by the ELSE portion of the algorithm since the timing analyzer considers a function to have only a final iteration.

During the first iteration of a loop, the instructions classified as first misses will be

```
pipeline_information = NULL.
IF n > 1 THEN
   Find the shortest continue path where all first misses are
      treated as misses and all first hits are treated as hits.
   pipeline_information = the best-case union of the
      information for all paths.
   Find the shortest continue path where all first misses are
      treated as hits and all first hits are treated as misses.
   Concatenate pipeline_information with the best-case union of
      the pipeline information for all paths n( - 2) times.
   For each set of exit paths that have a transition to a unique
       exit block DO
      Find the shortest exit path in the set where all first misses
         are treated as hits and all first hits are treated as misses.
      Concatenate pipeline_information with the best-case union
         of the information for all the exit paths in this set.
      Store this information with this exit block for the loop.
ELSE
   For each set of exit paths that have a transition to a unique
       exit block DO
      Find the shortest exit path in the set where all first misses
         are treated as misses and all first hits are treated as hits.
      Concatenate pipeline_information with the best-case union
         of the information for all the exit paths in this set.
      Store this information with this exit block for the loop.
```

Figure 11: Best-Case Loop Analysis Algorithm

misses since they are being encountered for the first time. No first miss can be treated as a hit until it has already been encountered. By the definition of first hit [3], instructions categorized as first hits will be treated as hits on the first iteration, and as misses on subsequent iterations. Because of the way a first hit is defined, one can safely assume that such instructions will miss in cache after the first iteration. Therefore, beginning with the second iteration of the loop, all first misses will become hits in the best-case algorithm, which may cause an underestimation in a loop having conditional control flow and first misses in multiple paths. This underestimation makes the timing analyzer's prediction of the BCET more conservative.

As in the worst-case loop algorithm, it is essential to handle the final iteration separately. It is common for a loop to break upon a certain condition, in which case the exit path is significantly shorter than the best-case continue path. This situation occurs in the program *Exit*, described in the previous section. In the worst-case loop algorithm, failure to consider the final iteration separately would have merely led to a less tight WCET prediction; in best case, however, such an overestimation would render the BCET prediction invalid.

3.2.4 Use of Vacant Cycles

During best-case analysis, it is sometimes necessary to ignore a potential data hazard to avoid an unwanted overestimation in execution time. Fig. 12 shows a situation in which a data dependency exists between the last instruction before a loop and the first instruction inside the loop. The lower two-thirds of the pipeline diagram show the behavior of the loop's instructions in isolation, without regard to the context of the loop. When the timing analyzer views the loop simply as a pipeline construct to insert after

SPARC Instructions

```
      34  ldd    [%o4],%f0

L24:  35  faddd  %f4,%f0,%f4
      36  add    %o4,8,%o4
      37  cmp    %o4,%o5
      38  bl,a   L24
      39  ldd    [%o4],%f0
```

Pipeline Diagram

		stage						
		IF	ID	EX	FEX	CA	WB	FWB
cycle	514	**34**						
	515		**34**					
	516			**34**				
	517					34		
	518					**34**		
	519							**34**
first iteration :	1	**35**						
	2	36	**35**					
	3	36			**35**			
	4	36			35			
	5	36			35			
	6	36			35			
	7	36						**35**
	...	...						
	11	36						
	12	37	36					
	13	38	37	**36**				
	14	39	38	37		**36**		
	15		39			37	**36**	
	16			39			37	
	17					39		
	18					39		
	19							39
last iteration :	73	35						
	74	36	35					
	75	37	36		35			
	76	38	37	36	35			
	77	**39**	38	37	35	36		
	78		**39**		**35**	37	36	
	79			**39**			**37**	35
	80					39		
	81					**39**		
	82							**39**

Figure 12: Data Hazard upon Entering a Loop

instruction 34, the pipeline shape of the loop will change as a result of the data hazard. For instance, instruction 35 can enter the IF stage at cycle 515, the ID at 516, but it cannot begin the FEX until the value of `%f0` can be forwarded. Thus, instruction 35 stays in ID two extra cycles before going on to the FEX during cycles 519-522 and

finally the FWB during cycle 523.

However, even though instruction 35 leaves the ID, FEX and FWB stages two cycles later than originally expected when the loop was timed by itself, the rest of the loop's execution is unaffected by this data hazard. The reason is that instruction 36 misses in cache and cannot enter the ID stage until well after instruction 35 has vacated it. Thus, the delay due to the hazard would be overlapped with the instruction fetch miss. In worst-case analysis, when the timing analyzer detects a structural or data hazard, it delays both the victim's starting and ending times for the stage it is being prevented to enter on time. But in the best-case analysis this will lead to an unwanted overestimation.

Since it is desirable for the timing analyzer to be an efficient tool, it is advantageous to store as little information about the child loop as necessary, as shown in Tables 3-5 and Fig. 7. To avoid such an overestimation in this situation depicted in Fig. 12, the best-case analysis also keeps track of how many cycles each stage is vacant during all the iterations of the child loop. To compute the number of vacant cycles, the timing analyzer first determines when each stage is first occupied during the loop (which usually occurs during the first iteration) and when each stage is last occupied (typically during the last iteration). The number of cycles from the first occupation of a stage through its last occupation is the amount of time that stage is considered to be *active* during the loop's execution. For instance, as Fig. 12 and Table 7 indicate, the ID stage is first occupied during cycle 2 and last occupied during cycle 78, so the ID stage is *active* for 77 cycles. The timing analyzer also counts how many cycles each stage is *occupied* during the execution of the loop, and subtracts this number from its number of *active* cycles to obtain the number of *vacant* cycles for that stage. For example, the ID stage is

Table 7: Computing Vacant Cycles for Loop in Fig. 12

Loop Info	IF	ID	EX	FEX	CA	WB	FWB
First Occupied	1	2	13	3	14	15	7
Last Occupied	77	78	79	78	81	79	82
Number of Cycles Active	77	77	67	76	68	65	76
Number of Cycles Occupied	59	50	30	40	40	20	20
Number of Cycles Vacant	18	27	37	36	28	45	56

occupied for five cycles on each iteration of the loop in Fig. 12. Since the loop iterates ten times, the ID stage is occupied for 50 cycles during the entire execution of the loop. Subtracting the ID's *occupied* time from its *active* time yields the number of *vacant* cycles for the ID stage: namely 77 - 50 = 27 vacant cycles. Table 7 shows the number of active, occupied and vacant cycles for each pipeline stage for the loop in Fig. 12.

If there is a data or structural hazard for a particular stage, the delay is reduced by the number of vacant cycles in that stage; or if the number of vacant cycles for a stage is at least the amount of the delay due to a hazard, then the algorithm ignores the hazard. For instance, in Fig. 12, the number of vacant cycles in the ID, FEX and FWB stages are 27, 36 and 56, respectively, which is substantially more than the amount of delay experienced by instruction 35. Thus, the BCET of the child loop will not increase despite the data hazard.

On the other hand, had instruction 36 been a hit in cache on the first iteration, ignoring the data hazard would give rise to an *underestimation* when the child loop is inserted after instruction 34. In this case, the loop by itself would have an execution time of 73 cycles instead of 82 cycles. However, the number of cycles that each stage is

occupied during the loop would be unchanged. The number of vacant cycles in the ID, FEX and FWB would be 18, 27 and 47, respectively, each nine cycles fewer than in the situation in which instruction 36 was a hit on the first iteration of the loop. The number of vacant cycles for each of these stages is still larger than the amount of the delay due to instruction 35's data hazard, so again the delay is ignored. This causes an underestimation in the BCET equal to the data hazard delay. The timing analyzer does not know whether instruction 36 is a hit or miss in cache during the analysis of the program construct that contains the loop as its child. All that is known about the loop is its total execution time and the information in Table 7, in order to maintain a reasonable limit on how much information to store about each loop and function in the program.

3.3 Timing an Entire Program

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. The nodes representing the outer level of function instances are treated as loops that will iterate only once when entered.

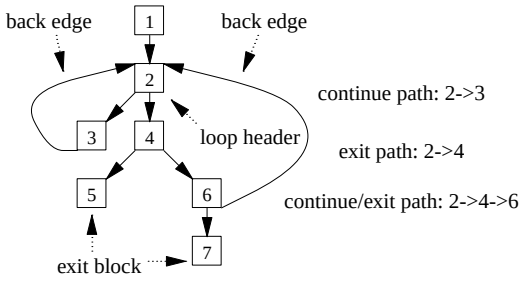

Figure 13: Example Introducing Loop Terminology

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. There will be a worst-case time calculated that corresponds to each exit block. Thus, when the timing analyzer is calculating the worst-case time for a path containing a child loop, it uses the child loop times associated with the exit block of the child loop that is the next block along the path. For instance, consider a loop depicted by the block diagram in Fig. 13. Each block, which consists of several assembly instructions, is represented by a numbered box. Blocks 5 and 7 are not part of the loop, but are rather exit blocks from the loop. Suppose that this loop is nested in an outer loop so that a backedge connects the end of block 5 to the beginning of block 1, and other backedge leads from the end of block 7 to the beginning of block 1. This outer loop has block 1 as its loop header block, and it contains two paths, one containing block 5 but not block 7, and the other containing block 7 but not block 5. It is necessary for the timing analyzer to distinguish the inner loop's total time depending on which exit path is taken. The path that exits to block 7 will have a longer time for the last iteration of the inner loop than the exit path to block 5. When the timing analyzer is processing the outer loop path containing block 5, it will follow these steps:

```
(1)  Evaluate the BCET/WCET for block 1.
(2)  Add the child loop time associated with exit block 5.
(3)  Add the time for block 5.
```

As soon as the timing analyzer reaches block 5 in the outer loop's path, it detects that the inner loop must execute before the first instruction in block 5, and it looks up the timing estimate stored with this block. A similar scenario takes place upon reaching block 7 in the other outer loop path.

3.3.1 First Miss Transitions in Worst Case

When incorporating a child loop's time into a parent loop, an adjustment is necessary if the child contains an instruction that is classified as a *first miss* in both the child and in the parent. Consider a program in which a child loop, `loop_2`, iterates ten times and contains an instruction *i* that is a *first miss* in the context of `loop_2`, and the parent, `loop_1`, also sees *i* as a *first miss* and also has 10 iterations. Instruction *i* should miss only the first time it is referenced: during the first iteration of `loop_2` within the first iteration of `loop_1`. The 99 subsequent references of *i* should all be hits.

Fig. 14 shows two pipeline diagrams for such a `loop_2`, where instruction 16 is a *first miss* at both loop levels. In the worst-case analysis, the instruction is considered a **hit** in the pipeline. The miss penalty is added when `loop_2` is being examined in the context of `loop_1`, and only added to the first iteration.

It is possible that this approach will produce an overestimation in the WCET. If the instruction that is categorized as a *first miss* spends more than one cycle in the IF stage as a result of an earlier hazard in the pipeline, then the miss penalty *overlaps with* the hazard. In Fig. 14, instruction 16, when seen as a hit in the pipeline, spends two cycles in the IF stage because of a structural hazard between instructions 13 and 14. But when instruction 16 misses in cache, it spends ten cycles in the IF stage, unaffected by the hazard. Adding the miss penalty after `loop_2`'s pipeline behavior has been evaluated results in an overestimation of one cycle for `loop_2`'s entire WCET.

3.3.2 First Miss Transitions in Best Case

The timing analyzer uses a slightly different approach to *fm*=>*fm* transitions for best case. The worst-case method for handling these transitions cannot be used in best case

Inner Loop Instructions

```
inst 13: st    %g2,[%g1]
inst 14: add   %g1,4,%g1
inst 15: cmp   %g1,%g3
inst 16: bl    L19
inst 17: sub   %g2,1,%g2
```

Worst-Case Pipeline Diagram

		stage						
		IF	ID	EX	FEX	CA	WB	FWB
cycle	1	**13**						
	2	14	**13**					
	3	15	14	**13**				
	4	16	15	14		**13**		
	5	16	15	14		13		
	6	**17**	16	15		14		
	7		**17**			15	**14**	
	8			**17**			15	
	9					**17**		
	10						**17**	

Best-Case Pipeline Diagram

		stage						
		IF	ID	EX	FEX	CA	WB	FWB
cycle	1	**13**						
	2	14	**13**					
	3	15	14	**13**				
	4	16	15	14		**13**		
	5	16	15	14		13		
	6	16		15		14		
	7	16				15	**14**	
	8	16					15	
	9	16						
	...	...						
	13	16						
	14	**17**	16					
	15		**17**					
	16			**17**				
	17					**17**		
	18						**17**	

Figure 14: Pipeline diagrams for First Miss Transition

because it may cause a small overestimation, and best-case overestimations result in invalid timing predictions. In the case of `loop_2` in Fig. 14, instruction 16 is again a first miss in both loop levels. The best-case approach considers the instruction to be a *miss* in the pipeline, in order to allow us to exploit as much pipeline overlap as possible in `loop_2`. When the timing analyzer evaluates `loop_2` in the context of `loop_1`, the miss penalty is *subtracted* from each iteration after the first.

However, as in worst case, the miss penalty does not always represent the difference between how many cycles the first miss instruction 16 spends in the IF stage when it is viewed as a hit and when it is viewed as a miss. In Fig. 14, although the miss penalty is nine cycles, instruction 16 only spends eight additional cycles in the IF stage when it misses in cache, because of the structural hazard between instructions 13 and 14. Subtracting the miss penalty from `loop_2`'s second through last iterations, when

instruction 16 is a hit, will cause an underestimation of one cycle per iteration. Hence, the BCET for the first entire execution of `loop_2` will be underestimated by 9 cycles.

3.3.3 First Hit Transitions in Worst Case

A less common type of instruction categorization transition that requires special attention in worst-case analysis is one involving a *miss* or *first miss* instruction in an inner loop categorized as a *first hit* at the next higher level. Consider the short program shown in Fig. 15. For this example, the cache configuration has two lines of four instructions each. The static cache simulator detects that instruction 0 in `fun()` is a first miss within the context of that function. Since a function only has one iteration, its first miss classification is equivalent to being an always miss. In addition, instruction 0 is determined to be a first hit in the context of the loop inside `main()`. Instructions 0 and 1 of `fun()` and instructions 0 and 1 of `main()` comprise program line 0,

C Source Code		SPARC Instructions		Program Line	Cache Line
`void fun()`		0	`retl`	**0**	**0**
`{`		1	`nop`	**0**	**0**
`  return;`					
`}`					
`main()`		0	`save  %sp,(-96),%sp`	**0**	**0**
`{`		1	`mov  %g0,%l0`	**0**	**0**
`  int i;`		2	`add  %l0,1,%l0`	**1**	**1**
	L19:	3	`call  _fun,0`	**1**	**1**
`  for (i = 0; i < 10; i++)`		4	`nop`	**1**	**1**
`  {`		5	`cmp  %l0,10`	**1**	**1**
`    fun();`		6	`bl,a  L19`	**2**	**0**
`  }`		7	`add  %l0,1,%l0`	**2**	**0**
`}`		8	`ret`	**2**	**0**
		9	`restore`	**2**	**0**

Figure 15: Program Containing *Miss -> First Hit* Instruction

instructions 2-5 of `main()` comprise program line 1, and instructions 6-9 of `main()` make up program line 2. Program line 0 arrives into cache when `main()` begins execution. The call to `fun()` is made from an instruction that maps to program line 1. When control transfers from `main()`'s instruction 4 to the function `fun()`, the `retl` instruction is already in cache. However, after control returns to `main()`, reference to instruction 6 causes cache line 0 to be overwritten because program line 2 also maps to cache line 0. When the loop iterates a second time, `fun()` is called again. But this time when the `retl` instruction is referenced, program line 0 must replace program line 2 in cache line 0, which means that `retl` instruction is a cache miss. Because of this cache conflict between `fun()` and the loop in `main()` that calls it, program lines 0 and 2 will repeatedly alternate occupation of cache line 0. Thus, while instruction 0 in `fun()` is a hit in cache during the first iteration of the loop in `main()`, it is a miss for all of the nine remaining iterations. Consequently, the static cache simulator concludes that this instruction should be classified as a first hit when viewed in the context of the loop in `main()`. It is necessary to detect that the `retl` instruction will only be a cache miss the first time it is referenced. The timing analyzer, when computing the execution time of `fun()`, realizes that such a categorization transition exists and that the function is being called from inside a loop. In this situation the timing analyzer considers the instruction as a *hit* in the pipeline when analyzing `fun()` in worst case and will add the miss penalty only to the second through final iterations of the loop that calls `fun()`.

3.3.4 Adjustments to Worst Case

Occasionally there is a situation in a program in which a loop being timed contains more paths than the maximum number of iterations. This can occur when a function has

multiple paths since a function only **iterates** once. Fig. 16 shows a function that contains three paths. The instruction cache was reconfigured to contain 32 lines of four bytes each. There is one instruction per cache line, and the entire function fits in cache. As a result, every instruction in `fun()` is classified as a first miss. Table 8 shows the result of the timing analysis of the function's three paths. When analyzing Path 1, all of its instructions are first misses encountered for the first time. The combined miss penalty from all seven instructions in the path is 63 cycles. Pipeline analysis of Path 1 determined that, had all the instructions been hits, the execution time would have been

```
      C Source Code            Inst     Assembly Code
-------------------------      ----    -------------------------
int fun (i)                     0       cmp    %o0,%g0
int i;                          1       bne,a  L14
{                               2       cmp    %o0,1
  if (i == 0)                   3       retl
    return 25;                  4       mov    25,%o0
  else if (i == 1)              5       bne    L15
    return 50;                  6       nop
  else                          7 L14:  retl
    return 75;                  8       mov    50,%o0
}                               9 L15:  retl
                               10       mov    75,%o0
```

Figure 16: A Function with Multiple Paths

Table 8: Path Information Pertaining to Function `fun()` in Fig. 16

Path Number	List of Instructions	WCET if newly encountered *fm = miss*	WCET if newly encountered *fm = hit*
1	0,1,2,5,6,9,10	74	11
2	0,1,2,5,6,7,8	29	11
3	0,1,2,3,4	27	9

11 cycles: 7 cycles due to the seven instructions each occupying the pipeline stages for one cycle apiece, plus the 4 cycles required to drain the pipeline. Adding the combined miss penalty results in a total of 74 cycles. Path 2 contains only two instructions (7 and 8) that were not encountered in Path 1's analysis, and Path 3 similarly contains only two first misses (instructions 3 and 4) not encountered in Paths 1 and 2. If `fun()` is called from a loop, during worst-case analysis the first misses will be encountered during the first three iterations. After the third iteration, each subsequent call to `fun()` will take eleven cycles.

Since the timing analyzer operates in a bottom-up manner, it does not know that `fun()` is being called from a loop until it begins to analyze that loop. If the timing analyzer does not detect the situation that a loop or function being timed has more paths than iterations, then an underestimation in the WCET prediction may result. The longest path in `fun()` takes 74 cycles when all first misses are misses, and 11 cycles when they are hits. The timing analyzer, when discovering a loop that calls `fun()`, would use 74 cycles as the function's WCET during the first iteration, and 11 cycles for the remaining iterations. However, it is possible that after traversing Path 1 during the first call to `fun()`, the function may use Path 2 during the second call. Thus, the second call of `fun()` will take 29 cycles, not 11 cycles, resulting in an underestimation of 18 cycles. A potentially underestimated WCET result is not acceptable.

The timing analyzer employs a simple procedure to handle this situation. It calculates a *base time* for the loop by finding the longest path where all first misses are treated as hits. An *adjust value* is calculated that is equal to the number of first misses in the entire loop times the cache miss penalty. When viewing the inner loop or function from an outer loop context, the *adjust value* will be added only to the first iteration of the outer

loop. For example, according to Table 8, the base time for `fun()` is 11 cycles and the adjust value is 99 cycles (eleven first misses multiplied by the nine cycle miss penalty). If `fun()` is called from a loop, the timing analyzer computes the first call to `fun()` to take 110 cycles, and then 11 cycles for each subsequent iteration. In doing so, the timing analyzer is assessing the miss penalty from every first miss in the function the first time it is invoked, thereby avoiding a WCET underestimation on the second and third calls to `fun()`. However, if the loop calling `fun()` has only one or two iterations, then the timing analyzer will overestimate the loop's time, since in reality not all of `fun()`'s first misses would be encountered.

CHAPTER 4

TIMING ANALYSIS RESULTS

This chapter discusses the accuracy of the timing analyzer's predictions of best-case and worst-case execution time.

4.1 The Simulator and Test Programs

The *ease* environment creates an instrumented executable file that invokes a simulator at each basic block in the program [10, 11]. The simulator traces the execution of the program and counts the number of clock cycles elapsed taking into consideration the SPARC instruction set and cache configuration. This author modified the existing cache simulator within *ease* to support measurements that take into account the MicroSPARC I's pipeline behavior. In addition, the modified simulator assumed a much smaller instruction cache, eight lines of sixteen bytes (four instructions) per line, than the MicroSPARC I's 4K cache in order to observe cache conflicts that would be more common in larger programs. Six simple programs were selected to assess the effectiveness of the timing analyzer. A description of these programs is given in Table 9. Column 2 gives the size of the assembly code for each program, assuming that an instruction occupies four bytes. For example, the *Matcnt* program contains 203 assembly instructions, so its code size is 203 * 4 = 812 bytes. The programs are each four to seventeen times larger than the 128-byte cache. Column 3 shows that each

Table 9: Test Programs

Name	Num Bytes	Num Functions	Description or Emphasis
Des	2,240	5	Encrypts and decrypts 64 bits
Matcnt	812	8	Counts and sums nonnegative values in a 100x100 integer matrix
Matmul	768	7	Multiplies two 50x50 integer matrices
Matsum	644	7	Sums nonnegative values in a 100x100 integer matrix
Sort	556	5	Bubblesort array of 500 integers into ascending order
Stats	1,428	9	Std. dev. & corr. coef. of two arrays of 1000 floating point values

program was highly modularized to illustrate the handling of timing predictions across function calls.

Assessing the accuracy of the timing analyzer requires comparing the timing analyzer's (static) prediction with the simulator's (dynamic) measured time. Since the MicroSPARC I has a clock speed of 50 MHz [1], one can multiply the number of cycles by 20 nanoseconds to obtain an actual real-world time.[5] Since the execution time of different programs can differ widely, it is useful to consider the *ratio* of the timing analyzer's estimated cycle time to the simulator's observed time. Of course, the best possible timing prediction would yield a ratio of 1, when the estimated and observed times are the same. In the worst-case analysis one finds ratios greater than one, meaning that the timing prediction is an overestimate, being somewhat pessimistic whenever not being exact. Analogously, for best-case analysis, one should find a ratio less than or equal to 1 indicating an underestimation of execution time whenever the exact time cannot be precisely determined.

To recognize the utility of the timing analyzer, one can compare its estimated ratio to

[5] The number of cycles given by the timing analyzer does not take into account such issues as data caching and utility functions. All data cache accesses are assumed to be hits. Library functions such as `printf()` are assigned a time of 0 cycles.

a naive ratio: what the ratio would have been without performing either the pipeline or instruction cache analysis or neither. In particular, when assessing the timing analyzer's predictions of WCET considering only pipelining, the naive execution time assumes that all instruction accesses are hits and each executing instruction takes the number of cycles necessary to complete all of its pipeline stages with no overlap with the instructions before or after it. For a typical integer instruction, this means that it cannot enter the IF stage until the preceding instruction has left the WB stage. The naive WCET for combined pipeline and cache analysis considers all instruction assesses to be misses and assumes that no pipeline overlap exists between instructions. Best-case naive execution times are computed by considering all cache accesses as hits and assuming the maximum possible overlap between instructions. Thus, for all three types of analysis, a program's naive BCET is equal to the minimum number of instructions that could be executed. In order for the timing analyzer to be a useful tool, the estimated ratios should be significantly closer to 1 (toward a perfect prediction) than the respective naive ratios.

4.2 Pipeline Only

Table 10 shows the results of the *pipeline only* analysis for the six test programs listed in Table 9. The worst-case *pipeline 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 variable of an outer loop. At this point the

Table 10: Test Program Results for Pipeline-only Analysis

Name	Best Case				Worst Case			
	Observed Cycles	Estimated Cycles	Estim. Ratio	Naive Ratio	Observed Cycles	Estimated Cycles	Estim. Ratio	Naive Ratio
Des	34,837	15,684	0.45	0.36	66,594	68,254	1.02	3.82
Matcnt	1,013,307	1,013,207	1.00	0.38	1,063,572	1,063,572	1.00	2.38
Matmul	4,347,541	4,347,541	1.00	0.33	4,347,806	4,347,806	1.00	2.13
Matsum	913,275	913,175	1.00	0.35	933,540	933,540	1.00	2.28
Sort	11,158	4,174	0.37	0.32	3,380,660	6,748,925	2.00	8.13
Stats	447,478	447,477	1.00	0.41	900,231	900,231	1.00	1.70

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. A similar scenario to what happens in *Sort* is a nested loop such as this one:

```
for (i = 0; i < MAX; ++i)
  for (j = 0; j < i; ++j)
  {
   /* body of loop */
  }
```

in which the total number of times the body of the inner loop is invoked is approximately $\frac{1}{2}(MAX^2)$. In worst-case analysis the number of inner loop invocations would be MAX^2 and in best case the number of invocations would be MAX. Consequently, the number of iterations performed was overrepresented on average by about a factor of two for this specific loop during worst-case timing analysis. The error for best case is potentially more extreme because the timing analyzer assumes the inner loop will iterate only once for each iteration of the outer loop. Note that both of these problems are encountered by other timing tools and have nothing to do with the pipeline analysis.

The timing analyzer predicted the best-case *pipeline only* performance of *Matmul* and *Stats* exactly. The times reported for *Matcnt* and *Matsum* were both underestimated by 100 cycles (about 0.01%) because a data dependency upon entering an inner loop was discounted by the *Vacant cycle* method described in the previous chapter. The only way to detect this dependency as a hazard would be to know more information about the inner loop than just the beginning shape of its union. The best-case times for *Des* and *Sort* were more substantially underestimated for the same reason that the worst-case analysis was overestimated: multiple loop paths in which the dynamic execution takes a different path based on the *value of a variable* in the program, which cannot be easily determined by static analysis.

4.3 Cache Only

As reported in [5], the *instruction caching only* timing analysis results given in Table 11 are quite accurate. This analysis had exact predictions for *Matmul* and *Stats* since there were no conditional constructs except to exit loops. Besides looping constructs, the program *Matsum* had only one `if-then` construct to check if array elements were

Table 11: Test Program Results for Cache-only Analysis

Name	Best Case				Worst Case			
	Observed Cycles	Estimated Cycles	Estim. Ratio	Naive Ratio	Observed Cycles	Estimated Cycles	Estim. Ratio	Naive Ratio
Des	59,998	19,345	0.32	0.21	142,956	163,015	1.14	3.86
Matcnt	929,073	929,073	1.00	0.41	1,169,055	1,259,055	1.08	3.79
Matmul	1,527,648	1,527,648	1.00	0.94	1,527,648	1,527,648	1.00	9.36
Matsum	687,219	687,219	1.00	0.47	707,219	707,219	1.00	4.85
Sort	10,439	3,901	0.37	0.35	7,639,611	15,253,902	2.00	8.17
Stats	372,410	372,410	1.00	0.49	372,410	372,410	1.00	4.90

nonnegative. Thus, *Matsum*'s caching behavior was also predicted exactly. The *Matcnt* program used an 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 static cache simulator 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 did not have to be calculated. As a result, 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. The execution of this single path accounted for 40.61% of the total instructions referenced during the execution of the program. The programs *Des* and *Sort* had overestimations in the predicted WCET and underestimations in the predicted BCET caused by the same problems described previously for the *pipeline only* measurements.

4.4 Combined Analysis Results

The integrated *pipeline and caching* analysis also resulted in quite tight predictions, as shown in Table 12. Again the predictions for the programs *Matmul* and *Matsum* were very accurate. Note that the worst-case estimated cycles was slightly greater than the observed cycles for both of these programs. This overestimation was due to the problem

Table 12: Test Program Results for Pipeline and Cache Analysis

Name	Best Case				Worst Case			
	Observed Cycles	Estimated Cycles	Estim. Ratio	Naive Ratio	Observed Cycles	Estimated Cycles	Estim. Ratio	Naive Ratio
Des	65,615	22,247	0.34	0.19	149,706	169,613	1.13	5.02
Matcnt	1,549,095	1,548,798	1.00	0.25	1,769,321	1,859,323	1.05	3.69
Matmul	4,444,666	4,420,068	0.99	0.32	4,444,911	4,445,413	1.00	4.98
Matsum	1,257,239	1,157,240	0.92	0.26	1,277,465	1,277,477	1.00	4.08
Sort	19,957	4,428	0.22	0.18	7,765,648	15,504,693	2.00	10.78
Stats	607,399	601,406	0.99	0.30	1,016,048	1,016,145	1.00	3.12

of an instruction's caching behavior changing between loop levels. This change requires an adjustment as described in Section 3.3.1. This approach treats the instruction as a hit in the pipeline analysis and simply adds 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 subtracted and an accurate estimation is obtained. However, in these two programs the potential overlap between the miss penalty and a load hazard stall was not detected. In particular, *Matmul* had 50 miss penalties completely overlapping with stalls from an integer multiply instruction and another 52 misses overlapping with a one-cycle load hazard, resulting in the 502 cycle overestimation. The cause of the overestimation in *Stats* was similar, due to the presence of long-running floating-point instructions.[6] The *Matcnt*, *Des*, and *Sort* programs had its usual overestimations due to a cache conflict, data dependencies, and an inaccurate number of estimated loop iterations, respectively.

For the benchmark programs with little conditional control flow, *Matcnt*, *Matmul*, *Matsum* and *Stats*, the BCET for *pipeline and caching* analysis was within 8% of the

[6] In order to make an exact prediction in these situations, the timing analyzer would need to store a different union of the loop in which the instruction is considered a miss in the pipeline. The amount of information required would become unwieldy if there are n instructions within the loop that have a $fm => fm$ transition: it would need to store 2^n unions of the loop, thus making the algorithm too complex to be practical.

dynamically observed time. The underestimation was largely due to a situation in which an instruction is classified as a first miss in both an inner and outer loop, and when the instruction is a hit in cache, it spends an extra cycle in the IF stage because of a data hazard involving the previous instruction. The timing analyzer treats the instruction as a miss in the pipeline, and makes sure the miss penalty is only applied the first time the program references the instruction by subtracting the miss penalty from all subsequent references in the nested loop. Again, one can achieve an exact prediction by storing pipeline information about *both* cases (whether an instruction classified as a first miss should be treated as a miss or hit in the pipeline) for each first miss in the pipeline, but just as in worst case, this would potentially mean storing an exponential amount of data, making the algorithm overly inefficient. In *Matcnt*, some of the underestimation was also due to the use of the *Vacant cycle* method to ignore a data dependency when in fact a hazard took place when the program executed. The BCET of *Des* and *Sort* were significantly underestimated just as in the *pipeline only* analysis, again because of data dependencies and an inaccurate number of loop iterations.

Many real-time programs are larger than the ones listed in Table 9 that were analyzed for obtaining the results described in this chapter. The timing analyzer is most accurate when a test program has no conditional control flow, no data dependencies and when the number of loop iterations can be predicted at compile-time. But larger programs will likely contain conditional control flow, more data dependencies and more loops in which the number of iterations cannot be determined statically. Consequently, the timing analyzer's performance will typically be more similar to programs *Des* and *Sort* than to the other four programs. To address the question of how well the timing analyzer can evaluate a larger program, the author created a program which subsumed all six test

programs plus a new `main()` function which called each subprogram in turn. This program occupied 6572 bytes, 51 times the size of the simulated cache. In the worst-case analysis, the simulated and estimated cycle counts were each 85 cycles greater than the sum of the simulated and estimated values for the six programs given in Table 12, and the Estimated Ratio was 1.48. This ratio was influenced most by the subprogram *Sort* since its own estimated ratio and total executing time were both much greater than those of the other five subprograms.

The benefit of integrating the pipeline and instruction cache analysis is shown in Table 13. Had the analyses been handled independently, one would anticipate a greater overestimation in predicting WCET, since the cache miss penalty would not have the opportunity to overlap with a pipeline stall, as depicted in Fig. 1. The effect of an independent analysis strategy would be to add the cache miss penalty to the total time of a path when an instruction is predicted to be a miss and treat the instruction as a hit in the pipeline. As a result, the test programs would have been overestimated, on average, by an additional 3%.

Table 13: Ratios for Overlapped versus Independent Analysis

Name	Estim. Ratio With Overlapped Analysis	Estim. Ratio With Independent Analysis
Des	1.133	1.174
Matcnt	1.051	1.057
Matmul	1.000	1.000
Matsum	1.000	1.016
Sort	1.997	2.029
Stats	1.000	1.082
average	1.197	1.226

CHAPTER 5

USER INTERFACE

Vpo can automatically determine the minimum and maximum number of iterations of many loops in real-time programs. However, if the compiler cannot statically determine the loop bounds, the timing analyzer will query the user for them. Afterwards, the analysis proceeds as before, and when finished the timing analyzer invokes a graphical user interface that is depicted in Fig. 17.

Figure 17: Timing Analyzer User Interface

The user-interface is a tool allowing the user to quickly obtain information revealed by the timing analysis. It can provide the calculations derived during timing analysis concerning the BCET and WCET of paths, loops and entire programs [8, 12]. When the user interface is running, the user sees three windows on the screen. The main window on the left allows the user to request timing predictions at various levels: for functions, loops, paths and subpaths consisting of a sequence of basic blocks, and ranges of individual source instructions. For example, one can request an analysis of one particular loop, and then choose one of the paths through that loop. The timing predictions for BCET and WCET appear in the main window. The middle window depicts the C source code and the right window shows the corresponding assembly code. Whenever the user selects a different construct in the main window, the highlighted lines in the source and assembly windows are simultaneously updated and scrolled to the appropriate position. The user interface also permits the user to use the mouse to select a portion of source code as a way to request a particular path to highlight and analyze.

While the design and implementation of the user-interface are described elsewhere [12], the timing analyzer supplies the timing estimates and pipeline diagrams for it. Within the timing analysis tree, which is the major data structure that is traversed to evaluate an entire program, are the nodes representing loops and function instances. Each loop has a list of its execution paths, and with each path is stored its BCET and WCET. The user-interface can query the user for a path within the loop he wishes to consider, and merely look up the execution times for that path that have already been stored.

For the situation in which the user wants the execution times for a subpath, the user-interface can call a routine within the timing analyzer to compute either the BCET or

WCET of those blocks that comprise the subpath. The routine in the timing analyzer that computes the execution time of a path can be passed two parameters denoting the starting and ending block numbers of a subpath, plus two additional parameters which denote the starting assembly instruction within the first block of the subpath and the ending instruction within the last block. Default values for the instruction numbers indicate that the user wants the blocks that comprise the subpath to be timed in their entirety, while default values for block numbers mean that the entire path is to be evaluated. The user-interface obtains these parameters and passes them to the timing analyzer's path-timing routine which returns the appropriate execution time. To satisfy the user's request, the user-interface needs to call this routine twice: once to obtain the WCET and a second time to obtain the BCET.

The timing analyzer's routine to compute a path's execution time also creates a pipeline diagram similar to the ones depicted in Fig. 18. If the user selects a path or subpath containing no child loops or function calls, then the interface gives the user the option to view the pipeline diagram created by the timing analyzer for that section of assembly code. If the user tells the user-interface to produce a pipeline diagram, the user-interface will call the path-timing routine to create the pipeline diagram as it is processing the instructions in the path (or subpath) in the manner described in Section 3.1.

Figure 18: Pipeline Diagrams as Shown in User Interface

CHAPTER 6

RELATED 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 effect of pipelining and instruction caching [13, 14, 15]. There have also been some recent studies on predicting pipeline performance by Harmon *et. al.* [16] and Narasimhan and Nilsen [17]. Yet, these studies did not address caching issues.[7] 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. Li *et. al.* [18] used an integer linear programming approach to model instruction cache behavior. Arnold *et. al.* [5] implemented a timing analysis system to tightly bound instruction cache performance. However, these approaches 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 cache. Lim *et. al.* [19] described a method based on an extension of a previous timing tool [20]. Lim's method differs quite significantly from the approach described in this thesis. It builds on flow analysis techniques found in optimizing compilers. Lim's

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

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 Tables 3 and 4.

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 Fig. 1, the instruction fetch miss penalty of one instruction (instruction 1) can be completely hidden by a stall with a long running instruction (data hazard stall on instruction 2). Whether the fetch of instruction 1 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. The approach taken in this thesis, as described in Chapter 2, of categorizing the caching behavior of each instruction before the timing analysis, allows the detection

of such situations. For instance, 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.

CHAPTER 7

FUTURE WORK

Enhancements to the timing analyzer are ongoing. One goal is to make the tool retargetable, so that if a user wishes to obtain timing estimates on a different processor, all that would be necessary is a modification of the input file to the timing analyzer (see Fig. 2). While the performance of the timing analyzer described in this thesis was compared to a simulator of the MicroSPARC I's instruction cache and pipeline, it will also be beneficial to compare the timing predictions against measurements obtained from a logic analyzer running the test programs on a MicroSPARC I processor. To ensure that the timing analyzer can give estimates accurate for the MicroSPARC I, additional hardware features, such as wrap-around filling of cache lines and data caching, need to be taken into consideration.

The MicroSPARC I employs a wrap-around fill method of loading words into its cache upon a cache miss [1]. Each line in the instruction cache contains eight words or 32 bytes. These words are grouped into four pairs. When a cache miss occurs on word *w*, eight words including *w*, properly aligned, are loaded from main memory. The words are inserted into the cache line one at a time. The first word that arrives in the cache is *w*, seven cycles after the miss occurred. During the eighth cycle, the word paired with *w*, either $w + 1$ if *w* is even or $w - 1$ if *w* is odd, is brought into cache. The ninth cycle is a dead cycle; no word is written. The next pair of words is then loaded into cache,

followed by another dead cycle, and so on until the entire line has been loaded. As an example, consider Table 14, a situation in which requesting word 5 results in a miss.

As a result of the wrap-around filling of the cache lines, the miss penalty is not constant, in contrast to the assumption in this thesis that the miss penalty is always nine cycles. For example, suppose that three instructions, which map to word numbers 5, 6 and 7, respectively, within the same cache line, are executed. If referencing word 5 causes a miss, then the entire line will be loaded into cache. The instruction mapping to word 5 will suffer a miss penalty of 7 cycles, and the instruction mapping to word 6 will itself be delayed two cycles. In addition, a branch may transfer control to an instruction in the same cache line. For instance, if referencing word 0 results in a miss, and a branch is then taken to word 7, the instruction mapping to word 7 will experience a 9 cycle delay.

Unlike instruction caching, many of the addresses of references to data can change during the execution of a program, making the task of bounding execution time more challenging. However, many of the data references are known. For instance, static and global data references do retain their 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 vary for

Table 14: When Cache Words Are Loaded If Word 5 Causes a Miss

cache word	0	1	2	3	4	5	6	7
cycle when loaded	13	14	16	17	8	7	10	11

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.

CHAPTER 8

CONCLUSION

This thesis has presented a technique for bounding the 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 in 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 accurately concatenate the performance of paths in an efficient manner when predicting the performance of loops. Fourth, the tool uses a timing analysis tree to predict the performance of an entire program. Finally, a user interface allows users to obtain bounds on portions of the program. The simulated results show that the method of analysis, which takes into account the processor's pipeline and instruction cache behavior simultaneously, leads to tight bounds on the execution time of programs. In the case of worst-case analysis, detecting the overlap of pipeline hazards and cache misses provides tighter predictions than performing the timing analysis of both features independently.

REFERENCES

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

[2] 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).

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

[4] 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).

[5] R. Arnold, F. Mueller, D. Whalley, and M. Harmon, "Bounding Instruction Cache Performance," *ACM Transactions on Computer Systems*, (submitted June 1995).

[6] 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).

[7] 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).

[8] L. Ko, D. B. Whalley, and M. G. Harmon, "Supporting User-Friendly Analysis of Timing Constraints," *Proceedings of the ACM SIGPLAN Notices 1995 Workshop on Languages, Compilers, and Tools for Real-Time Systems* **30**(11) pp. 99-107 (November 1995).

[9] C. A. Healy, D. B. Whalley, and M. G. Harmon, "Integrating the Timing Analysis of Pipelining and Instruction Caching," *Proceedings of the Sixteenth IEEE Real-Time Systems Symposium*, pp. 288-297 (December 1995).

[10] J. W. Davidson and D. B. Whalley, "Ease: An Environment for Architecture Study and Experimentation," *Proceedings SIGMETRICS '90 Conference on Measurement and Modeling of Computer Systems*, pp. 259-260 (May 1990).

[11] 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).

[12] L. Ko, *Supporting User-Friendly Analysis of Timing Constraints,* Masters Project, Florida State University, Tallahassee, FL (April 1995).

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

[14] 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).

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

[16] 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).

[17] 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).

[18] Y. S. Li, S. Malik, and A. Wolfe, "Performance Estimation of Embedded Software with Instruction Cache Modeling," *International Conference on Computer-Aided Design*, (November 1995).

[19] 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).

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

BIOGRAPHICAL SKETCH

Christopher Andrew Healy was born in Danbury, Connecticut in 1971. He earned the Bachelor of Science degree in mathematics from Florida State University in 1993. After obtaining the master's degree, he plans to embark on the doctoral program in Computer Science at F.S.U.

ACKNOWLEDGEMENTS

I wish to thank my major professor, Dr. David Whalley, for his patience, guidance and support during my research. I am also grateful for the assistance both he and Dr. Kacmar offered in typesetting this thesis. Dr. Riccardi and Dr. Baker also provided helpful suggestions that improved the quality of this thesis. The timing analyzer described in this thesis is an extension of an earlier tool created by Robert Arnold which bounded instruction cache performance. Frank Mueller implemented the static cache simulator that provides necessary information to the timing analyzer. Lo Ko and Emily Ratliff implemented the graphical user interface. The research upon which this thesis is based was supported in part by the Office of Naval Research under contract number N00014-94-1-0006.

TABLE OF CONTENTS

Page

List of Tables vi

List of Figures vii

Abstract ix

1 INTRODUCTION 1

2 PREVIOUS WORK 4

3 TIMING ANALYSIS 7
- 3.1 Analyzing A Single Path of Instructions 8
- 3.2 Loop Analysis 15
  - 3.2.1 The Union Concept 15
  - 3.2.2 Worst-Case Algorithm for Timing a Loop 20
  - 3.2.3 Best-Case Algorithm for Timing a Loop 25
  - 3.2.4 Use of Vacant Cycles 26
- 3.3 Timing an Entire Program 30
  - 3.3.1 First Miss Transitions in Worst Case 32
  - 3.3.2 First Miss Transitions in Best Case 32
  - 3.3.3 First Hit Transitions in Worst Case 34
  - 3.3.4 Adjustments to Worst Case 35

4 TIMING ANALYSIS RESULTS 39
- 4.1 The Simulator and Test Programs 39
- 4.2 Pipeline Only 41
- 4.3 Cache Only 43
- 4.4 Combined Analysis Results 44

5 USER INTERFACE 48

6 RELATED WORK 52

7 FUTURE WORK 56

8 CONCLUSION 59

References 60

Biographical Sketch 62

LIST OF TABLES

TABLE NUMBER AND DESCRIPTION **PAGE**

1. Definitions of Worst-Case Instruction Categories 5

2. Definitions of Best-Case Instruction Categories 5

3. Creating Beginning Pipeline Information for `Square()` 12

4. Creating Ending Pipeline Information for `Square()` 12

5. Data Hazard Information for the Instructions in `Square()` 14

6. Structural Hazard Information for Unions in Fig. 7 19

7. Computing Vacant Cycles for Loop in Fig. 12 29

8. Path Information Pertaining to Function `fun()` in Fig. 16 36

9. Test Programs 40

10. Test Program Results for Pipeline-only Analysis 42

11. Test Program Results for Cache-only Analysis 43

12. Test Program Results for Pipeline and Cache Analysis 45

13. Ratios for Overlapped versus Independent Analysis 47

14. When Cache Words Are Loaded If Word 5 Causes a Miss 57

LIST OF FIGURES

FIGURE NUMBER AND DESCRIPTION **PAGE**

1. Example of Overlapping Pipeline Stages with a Cache Miss 3

2. Overview of Bounding Pipeline and Cache Performance 4

3. A Skeleton Program 8

4. Path through Function `Square()` 11

5. Program Containing a Loop with Two Paths 17

6. Pipeline Diagrams for the Two Loop Paths in Fig. 5 18

7. Pipeline Diagrams for the Worst-Case and Best-Case Unions 19

8. Worst-Case Loop Analysis Algorithm 21

9. C Source Code and Assembly Code for the Program *Exit* 24

10. Pipeline diagrams for the Two Paths through Loop in *Exit* 24

11. Best-Case Loop Analysis Algorithm 25

12. Data Hazard upon Entering a Loop 27

13. Example Introducing Loop Terminology 30

14. Pipeline diagrams for First Miss Transition 33

15. Program Containing *Miss -> First Hit* Instruction 34

16. A Function with Multiple Paths 36

17. Graphical User Interface for the Timing Analyzer 48

18. Pipeline Diagrams as Shown in User Interface 51

ABSTRACT

Recently designed machines contain pipelines and instruction 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 the memory references performed. Furthermore, these penalties are not independent since delays due to pipeline stalls and cache miss penalties may overlap. This thesis describes an approach for predicting the execution time 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 execution time 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. The results indicate that the timing analyzer efficiently produces tight predictions of best-case and worst-case performance on machines with pipelining and instruction caching.