THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

AUTOMATIC UTILIZATION OF CONSTRAINTS FOR TIMING ANALYSIS

By

CHRISTOPHER A. HEALY

A Dissertation submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Degree Awarded:
Summer Semester, 1999

The members of the Committee approve the dissertation of Christopher A. Healy defended on July 2, 1999.

David B. Whalley
Professor Directing Thesis

Steven F. Bellenot
Outside Committee Member

Theodore P. Baker
Committee Member

Kyle Gallivan
Committee Member

Robert A. van Engelen
Committee Member

Approved:

T. P. Baker, 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. Responses that are given past a deadline is not acceptable. A real-time system is often comprised of a set of tasks that are statically scheduled. 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. 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), rather than the best-case execution time (BCET), 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.

This dissertation discusses research in timing analysis to provide tighter WCET and BCET predictions. A previous version of a timing analyzer focused on architectural features, specifically integrating the analysis of pipelining and instruction caching [1, 2]. One could extend this implementation to take into account additional hardware features. However, the author believes that addressing machine-independent issues in timing

analysis will have a greater and longer-lasting benefit than merely focusing on the architecture. New architectural features are being developed at a rapid pace. Thus, it is difficult for timing analysis research to keep up with the latest hardware features. On the other hand, even if a timing analyzer perfectly models a processor's architecture, significant WCET overestimations and BCET underestimations can still result because of dependences on data values that can constrain the number of loop iterations and the set of paths that can be taken in a program. Two types of constraints will be discussed in this dissertation: *loop iteration constraints* that influence the number of iterations of loops, and *branch constraints* that indicate whether or not a particular branch will be taken or fall through. When the term *constraint* appears hereafter in this dissertation, it shall mean both types of constraints collectively. This dissertation describes how these constraints in a program can be automatically detected and exploited to tighten the execution time predictions.

The remainder of the dissertation will proceed as follows. Chapter 2 examines related work in the area of predicting execution time. Chapter 3 presents the context in which the timing analyzer was originally designed with respect to its input/output and ancillary software. Chapter 4 discusses the work on loop iteration constraints to calculate the number of loop iterations accurately and automatically. Chapter 5 describes the other extension to the timing analyzer pertaining to detecting and exploiting branch constraints. Chapter 6 briefly summarizes the major results of the timing analysis. Chapter 7 describes future work and Chapter 8 presents the conclusions.

CHAPTER 2

RELATED WORK

Predicting execution time of programs is an emerging area of research in real-time systems. Initial work in this area concentrated on analyzing source programs. Puschner and Koza [3] associated the number of machine cycles to individual C statements or consecutive statements not containing conditional control flow. Niehaus [4] showed how the execution time can correspond to basic blocks after intermediate code generation. Park [5] created an Information Description Language so that the user could specify the number of loop iterations or that two source code statements must execute in the same path. However, all these studies ignored hardware effects. Over time, timing analysis research encompassed the study of architectural features, such as pipelined execution and caches [6, 7, 8, 9, 10, 2]. More recent research is concerned with how dependences on data values can constrain paths and thereby influence the program's execution time. The major difference between the work described in this dissertation and the related work performed elsewhere is the way in which the number of loop iterations and the branch constraints are made known to the timing analyzer. Other research groups that use constraint information require the user to painstakingly enter this information [9, 11, 12]. Chapters 4 and 5 will describe how this information can instead be automatically detected by a compiler and exploited by a timing analyzer.

Many existing timing analyzers require that a user specify the number of iterations of

each loop in the program. This specification may be requested interactively [13, 9]. Thus, each time the timing analyzer is invoked for a program, the bounds for every loop in the program must be specified, which is error prone and tedious for the user. Alternatively, one could specify this information as assertions in the source code to prevent repeated specifications of the same information [14, 3]. However, there are still several disadvantages. First, the user is still required to write the assertions. Second, there is no guarantee that the user will specify the correct minimum and maximum iterations. This problem may easily occur when a user changes the loop, but forgets to update the corresponding assertion. Also, code generation strategies, such as whether to place instructions for the loop exit condition code at the beginning or end of the loop, may cause the number of loop iterations to vary by one iteration. A user should only be required to examine the source code and not be required to know the code generation strategies of the compiler. Finally, compiler optimizations, such as loop unrolling, may affect the number of times a loop iterates. Inhibiting different code generation strategies or compiler optimizations to more easily estimate loop bounds would sacrifice performance, which is quite undesirable.

Other previous work in timing analysis has been accomplished using constraint-based systems. Li *et al.* [15, 9] developed an Implicit Path Enumeration (IPE) technique that used Integer Linear Programming (ILP) to solve constraints about the program to obtain timing predictions. The cost function was a sum of terms of the form $c_i x_i$, where for each block i, c_i is the execution time of the block i and x_i is the number of times the block executes. Their approach uses structural constraints based on the program's control flow and functional constraints entered by the user that deal with the number of times that each The work of Ottosson and Sjödin [11] extended the IPE technique by

using finite domain constraints to model the architectural features of the hardware. However, in both approaches these constraints were entered manually by the user, which is both a tedious and error-prone task.

Recent work by Ermedahl and Gustafsson [16], Lundqvist and Stenström [17] and Liu and Gomez [18] use abstract interpretation and symbolic execution to automatically derive many branch constraints. These approaches are quite powerful, but effectively requires simulating all paths of a loop for every loop iteration. Thus, these approaches require significant analysis overhead, which would be undesirable when analyzing long running programs.

Wilhelm, Ferdinand *et al.* [19, 20, 12] have also contributed to the area of timing prediction that uses constraints. They separate timing analysis into two distinct phases: cache analysis and path analysis. The purpose of this partition is to use ILP only to perform the path analysis. Their cache analysis is based on the principles developed in [21]. Their path analysis technique is similar to that of Li *et al.*, requiring the user to enter the constraints. Even though their overall approach is powerful and yields accurate WCET bounds, the ILP phase may be inefficient to implement in practice.

Much tighter bounds on the WCET and BCET can result when a timing analyzer incorporates information about the program related to loop iteration and branch constraints. All of the approaches thus far proposed to extend timing analysis to include the exploitation of constraints have either required the user to enter this information manually, and/or required significant analysis overhead. It would be much more convenient for the user if a timing analyzer could automate the process of detecting and exploiting such constraints in an efficient manner. This is the major motivation of this dissertation.

CHAPTER 3

FRAMEWORK FOR THE RESEARCH

The timing analyzer described in this dissertation 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* [22], a static instruction cache simulator and a timing analyzer with a graphical user interface. Figure 1 depicts an overview of the approach for predicting the execution time of code segments or entire programs on machines with pipelines and instruction caches. Table 1 outlines the work that has been accomplished for the timing analysis environment.

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

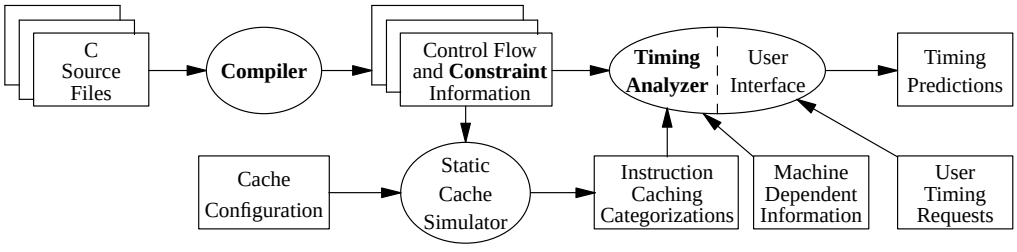


Figure 1: Overview of the Timing Analysis Environment

Table 1: Work Accomplished for Timing Analyzer

Module	Lines	Student Contributors	Purpose
compiler	64,000		Provides information on control flow, loop iterations and branch constraints.
static cache simulator	15,000	Frank Mueller	Provides instruction cache categorizations.
		Randall White	Provides the data cache categorizations.
timing analyzer	19,000	Robert Arnold	Bounds execution time based on instruction cache performance.
		Christopher Healy	Bounds execution time based on pipelining, wrap-around-file cache, loop iteration and branch constraints.
		Randall White	Bounds execution time bases on data caching.
user interface	10,000	Lo Ko, Emily Ratliff, Nagham Al-Yaqoubi	Provides WCET and BCET for user-selected code portions. Warns user if timing constraints can be violated.

given cache configuration. The caching behavior of an instruction is assigned one of four categories, described in Tables 2 and 3, for each loop level in which an instruction is contained. The theory and implementation of static cache simulation is described in more detail elsewhere [21, 23, 2, 24, 25]. The timing analyzer uses the instruction caching 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 characteristics in order to detect structural and data hazards between instructions.

Given a program's control-flow information and instruction caching categorizations

Table 2: 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 3: 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	The instruction may be 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.

along with the processor's instruction set information, the timing analyzer then derives best-case and worst-case estimates for each path, loop and function within the program. A timing analysis tree is constructed, where the each node of the tree corresponds to a

loop or function in the function instance graph. Each node is considered a natural loop.[1] A node that represents a function instance is treated as a loop that will iterate exactly once when entered. The loops in the timing analysis tree are processed in a bottom-up manner. In other words, the WCET and BCET for a loop are not calculated until the times for all of its immediate child loops are known. This means that 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()`.

The version of the timing analyzer described in this dissertation is an extension of an earlier timing tool [25, 2, 26] that bounded instruction cache and pipeline performance. When the timing analyzer has completed its analysis, it invokes a graphical user interface [27] allowing the user to request timing bounds for portions of the program. These portions may be at any one of several levels of the analysis: the entire program, a function, loop, code section, path, sub-path or ranges of instructions. Some of the research that has been associated with the timing analyzer, though not directly related to this dissertation, includes analysis of data caches [28, 29, 10], wrap-around fill instruction caches [10] and partitioning control flow in cases where the number of paths is arbitrarily large [30]. Excerpts of this dissertation, including a concise description of the algorithm and results, can be found in [31, 32].

A description of the programs used to test the timing analyzer is given in Table 4. Six of these programs, *Des*, *Matcnt*, *Matsum*, *Matmul*, *Sort* and *Stats*, were used in the

[1] 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.

results of the original timing analyzer. The remaining programs were added to the test suite to illustrate situations of various constraints that are described in Chapters 4 and 5. The programs printed in boldface in Table 4 are published in *Numerical Recipes in C* [33, 34]. The code size of all programs ranged from 22 to 668 assembly instructions, with an average of 211 instructions. The reason why the programs in the test set are relatively small was so that it would be feasible for the author to determine manually the actual worst-case and best-case input data.

For each program a direct-mapped instruction cache configuration containing 8 lines of 16 bytes was used. It was assumed that a cache hit required one cycle, a cache miss required ten cycles, and all data cache references were assumed to be hits. This is the same cache configuration that has been used in several previous timing analysis studies

Table 4: Test Programs

Name	Description or Emphasis
Des	Encrypts and decrypts 64 bits
Expint	Computes an exponential integral
Fresnel	Computes non-complex Fresnel integrals
Gaujac	Computes Abscissas and Weights of a 10 point Gauss-Jacobi quadrature formula
Hes	Reduces a 100x100 matrix to Hessenberg form
Integ	Evaluates a double integral over a trapezoidal region
Interp	Polynomial interpolation of 500 points
LU	Performs LU Decomposition on a 100x100 matrix
Matcnt	Counts and sums nonnegative values in a 100x100 integer matrix
Matmul	Multiplies two 50x50 integer matrices
Matsum	Sums nonnegative values in a 100x100 integer matrix
Sort	Bubblesort array of 500 integers into ascending order
Sprsin	Converts a 20x20 integer matrix into row-index sparse storage mode
Stats	Std. dev. & corr. coef. of two arrays of 1000 floating point values
Summidall	Sums the middle half and all elements of a 1000 integer vector
Summinmax	Sums the min. and max. of corresponding elements of two 1000 element vectors
Sumnegpos	Sums the negative, positive and all elements of a 1000 integer vector
Sumoddeven	Sums the odd and even numbered elements of a 1000 integer vector
Sym	Tests if a 500x500 matrix is symmetric

[25, 26, 31].

The implementation of the timing analysis environment includes about 19,000 lines of C source code in the timing analyzer itself (14,000 written by the author), plus other modules depicted in Figure 1. The compiler, static cache simulator and the algebraic solver were implemented by other researchers at FSU. To obtain results that integrate instruction cache and pipeline effects, the author had previously amended a traditional cache simulator [35]. This modification required about 2,000 lines of source code.

Assessing the accuracy of the timing analyzer was accomplished by comparing the timing analyzer's (static) prediction with the simulator's (dynamic) measurements. 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.

Table 5 shows the results for *Cache Only* analysis and Table 6 gives the results when both caching and pipelining are analyzed. These results show the state of the timing analyzer before work on the dissertation was begun. To recognize the utility of the timing analyzer, one can compare its estimated ratio to a naive ratio: what the ratio would have been without performing any analysis. Table 5 shows that cache analysis can provide much tighter bounds on the execution time versus the naive ratios.

Table 5: Results for Cache-Only Analysis

Worst-Case Results					
Name	Observed Cycles	Naive Cycles	Naive Ratio	Cache Only Cycles	Cache Only Ratio
Des	149,706	770,142	5.144	398,604	2.663
Expint	58,217	2,933,194	50.384	2,004,151	34.426
Fresnel	47,749	106,121	2.222	73,199	1.533
Gaujac	786,786	1,579,588	2.006	1,153,123	1.466
Hes	55,834,609	686,873,410	12.302	404,879,389	7.251
Integ	22,538,082	99,585,206	4.419	54,544,607	2.420
Interp	25,469,403	107,183,344	4.208	75,543,529	2.966
LU	23,055,832	791,692,885	34.338	420,919,578	18.257
Matcnt	1,769,321	6,525,017	3.688	3,262,463	1.844
Matmul	4,444,911	22,122,016	4.977	9,370,402	2.108
Matsum	1,277,465	5,214,645	4.082	2,401,380	1.880
Sort	7,672,281	80,913,015	10.546	38,220,912	4.982
Sprsin	28,339	188,294	6.644	76,838	2.711
Stats	1,016,048	3,168,159	3.118	1,852,107	1.823
Summidall	15,340	212,207	13.834	104,108	6.787
Summinmax	16,080	201,179	12.511	102,089	6.349
Sumnegpos	11,067	159,137	14.379	78,065	7.054
Sumoddeven	15,092	195,343	12.943	96,181	6.373
Sym	2,747,654	71,752,986	26.114	24,673,698	8.980
Average	7,734,420	99,019,784	11.993	54,723,917	6.414
Best-Case Results					
Name	Observed Cycles	Naive Cycles	Naive Ratio	Cache Only Cycles	Cache Only Ratio
Des	65,615	12,559	0.191	19,183	0.292
Expint	125	29	0.232	102	0.816
Fresnel	181	43	0.238	151	0.834
Gaujac	45,270	12,117	0.268	34,104	0.753
Hes	306,733	4,427	0.014	13,301	0.043
Integ	19,160,842	2,510,015	0.131	2,532,560	0.132
Interp	6,485,878	47,509	0.007	119,590	0.018
LU	12,883,939	216,528	0.017	232,782	0.018
Matcnt	1,549,095	383,241	0.247	1,020,783	0.659
Matmul	4,444,666	1,429,980	0.322	1,774,995	0.399
Matsum	1,257,239	323,214	0.257	957,111	0.761
Sort	19,966	9,600	0.481	9,888	0.495
Sprsin	17,436	7,313	0.419	15,701	0.900
Stats	607,399	182,312	0.300	417,230	0.687
Summidall	15,340	7,015	0.457	7,069	0.461
Summinmax	13,080	12,013	0.918	12,058	0.922
Sumnegpos	9,067	8,010	0.883	8,037	0.886
Sumoddeven	94	59	0.628	63	0.670
Sym	160	38	0.238	137	0.856
Average	2,467,480	271,896	0.329	377,623	0.558

Table 6: Results After Adding Pipeline Analysis

Worst-Case Results					
Name	Observed Cycles	Cache Only Cycles	Cache Only Ratio	+ Pipelining Cycles	+ Pipelining Ratio
Des	149,706	398,604	2.663	172,509	1.152
Expint	58,217	2,004,151	34.426	1,293290	22.215
Fresnel	47,749	73,199	1.533	48,887	1.024
Gaujac	786,786	1,153,123	1.466	790,116	1.004
Hes	55,834,609	404,879,389	7.251	130,574,296	2.339
Integ	22,538,082	54,544,607	2.420	30,023,163	1.332
Interp	25,469,403	75,543,529	2.966	50,701,362	1.991
LU	23,055,832	420,919,578	18.257	124,577,237	5.403
Matcnt	1,769,321	3,262,463	1.844	1,861,150	1.052
Matmul	4,444,911	9,370,402	2.108	4,448,212	1.001
Matsum	1,277,465	2,401,380	1.880	1,279,322	1.001
Sort	7,672,281	38,220,912	4.982	15,251,603	1.988
Sprsin	28,339	76,838	2.711	28,664	1.011
Stats	1,016,048	1,852,107	1.823	1,016,128	1.000
Summidall	15,340	104,108	6.787	18,090	1.179
Summinmax	16,080	102,089	6.349	17,080	1.062
Sumnegpos	11,067	78,065	7.054	13,068	1.181
Sumoddeven	15,092	96,181	6.373	16,112	1.068
Sym	2,747,654	24,673,698	8.980	5,481,220	1.995
Average	7,734,420	54,723,917	6.414	19,347,974	2.631

Best-Case Results					
Name	Observed Cycles	Cache Only Cycles	Cache Only Ratio	+ Pipelining Cycles	+ Pipelining Ratio
Des	65,615	19,183	0.292	22,247	0.339
Expint	125	102	0.816	118	0.944
Fresnel	181	151	0.834	172	0.950
Gaujac	45,270	34,104	0.753	44,566	0.984
Hes	306,733	13,301	0.043	14,006	0.046
Integ	19,160,842	2,532,560	0.132	12,808,073	0.668
Interp	6,485,878	119,590	0.018	143,064	0.022
LU	12,883,939	232,782	0.018	284,011	0.022
Matcnt	1,549,095	1,020,783	0.659	1,548,798	1.000
Matmul	4,444,666	1,774,995	0.399	4,420,068	0.994
Matsum	1,257,239	957,111	0.761	1,167,140	0.923
Sort	19,966	9,888	0.495	19,950	0.999
Sprsin	17,436	15,701	0.900	17,379	0.997
Stats	607,399	417,230	0.687	601,406	0.990
Summidall	15,340	7,069	0.461	8,072	0.526
Summinmax	13,080	12,058	0.922	13,062	0.999
Sumnegpos	9,067	8,037	0.886	9,049	0.998
Sumoddeven	94	63	0.670	63	0.670
Sym	160	137	0.856	160	1.000
Average	2,467,480	377,623	0.558	1,111,653	0.741

Likewise, Table 6 shows how much tighter the WCET and BCET predictions become when pipeline analysis is added.[2] The naive WCET considers all instruction accesses 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, 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.

[2] Tables 12 and 22 later in this dissertation show the further tightening of the WCET and BCET based on the new analysis described in this dissertation. Table 23 summarizes all the worst-case and best-case ratios from every level of analysis so that the reader can make quick comparisons.

CHAPTER 4

OBTAINING TIGHT BOUNDS OF LOOP ITERATIONS

This chapter discusses general methods how loop iteration constraints are automatically calculated. Most of a program's execution time is spent inside of loops, so to be able to predict the WCET and BCET of a program, one must know the number of iterations that can be performed by the loops in the program. Under certain conditions, such as a loop with a single exit, many compilers statically determine the exact number of loop iterations [22]. Besides timing analysis, applications for determining this number include loop unrolling [36], software pipelining [37], and exploiting parallelism across loop iterations [38]. When the number of iterations cannot be exactly determined, it would be desirable in a real-time system to know the lower and upper iteration bounds. These bounds can be used by a timing analysis tool to more accurately predict BCETs and WCETs.

Several existing timing analyzers require the user to manually enter the number of iterations for each loop, which is a tedious and error prone process. It would be more appropriate to have the compiler automatically and efficiently determine the bounds for each loop in a program when possible. This chapter describes three approaches that support timing analysis by bounding the number of loop iterations. First, an algorithm is presented that determines a bounded number of iterations for loops with multiple exits. Second, support is provided for loops whose number of iterations is dependent on loop-

invariant variables. Finally, a method is given to accurately predict the average number of iterations for inner loops, whose number of iterations varies depending upon the values of counter variables of enclosing outer loops. All three of these approaches are efficiently implemented and result in less work for a user. The last approach also results in tighter timing analysis predictions. These approaches were implemented by modifying the *vpo* compiler [22] to analyze loops and this loop analysis information is passed to a timing analyzer [25, 26, 29] to predict performance.

4.1 Bounding Iterations for Loops with Multiple Exits

This section presents a method to determine a bounded number of iterations for natural loops with multiple exits. The method includes the following steps. (1) First, the conditional branches within the loop that can affect the number of loop iterations are identified. (2) Next, the compiler calculates when each of the identified branches can change its result based on the number of loop iterations performed. (3) Afterwards, the range of loop iterations when each of these branches can be reached is determined. (4) Finally, the minimum and maximum number of iterations for the loop is calculated. These steps are described in the following subsections.

4.1.1 Branches Affecting the Number of Loop Iterations

Some terms are now defined to facilitate the presentation of the methods given in this chapter. A more complete description of these terms can be found elsewhere [39]. A *basic block* is a sequence of instructions with a single entry point at the beginning and a single exit point at the end. A *natural loop* is a loop with a single entry point. The *header* of a natural loop is the single basic block where the loop is entered. Transitions from within the loop to the header are called *back edges*. Block A *dominates* block B if

every path from the initial node of the control flow graph to B has to first go through A. For instance, the header block of a natural loop dominates all other blocks in the loop. Likewise, block B *post-dominates* block A if all control paths from block A eventually lead to block B. A block always dominates and post-dominates itself. For this dissertation, the number of loop iterations is defined to be the number of times the header is executed once the loop is entered [25].

Figure 2(a) contains the code for a toy C function that will be used to illustrate the algorithm for calculating loop iteration bounds for loops with multiple exits. Figure 2(b) depicts the RTLs, representing SPARC assembly instructions, that the *vpo* compiler has generated for this function. (No delay slots have been filled in order to simplify the example.) Figure 2(c) explains the RTL notation used. The loop consists of basic blocks 2, 3, 5, 6, 7, and 8. The header of the loop is block 7.

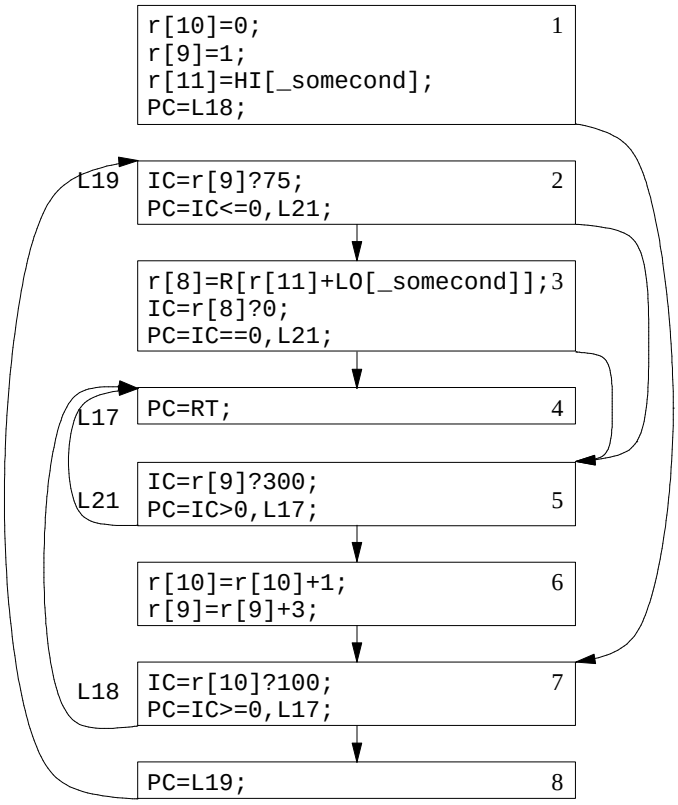
An *iteration branch* in a loop is a conditional transfer of control, where the choice between the two outgoing transitions can directly or indirectly affect the number of loop iterations. The iteration branches in the loop that can directly affect this number are branches that have (1) a transition to a basic block outside the loop or (2) a transition to the header block of the loop or to a block that is post-dominated by the header. Iteration branches that can indirectly affect the number of loop iterations are those branches whose two successors are post-dominated by different iteration branches. Figure 3 shows an algorithm to calculate the set of iteration branches I for a loop. The worst-case complexity of the algorithm is $O(B^2)$, where B is the number of basic blocks in the loop. However, the average complexity would be closer to $O(B)$ since iteration branches that indirectly affect the number of loop iterations are not common.

The algorithm shown in Figure 3 identifies block 5 as containing an iteration branch

```
main()
{
   int i, j;
   extern int somecond;

   for (i = 0, j = 1; i < 100; i++, j += 3)
      if (j > 75 && somecond || j > 300)
         break;
}
```

(a) Source Code



(b) Corresponding SPARC Instructions

`r[9]`	: allocated for variable j
`r[10]`	: allocated for variable i
`HI[<address>]`	: high portion of address
`LO[<address>]`	: low portion of address
`R[<address>]`	: integer memory reference
`IC=<item>?<item>;`	: comparison
`PC=IC<relop>0,<label>;`	: conditional branch
`PC=RT;`	: return
`PC=<label>;`	: unconditional jump

(c) Explanation of RTL Notation in Figure 2(b)

Figure 2: Example Loop with Multiple Exits

```
// Find the iteration branches that can directly affect the number of iterations.
I = {}
FOR each block B in the loop L DO
    IF (B has two succs S1 and S2) THEN
        IF (S1 ∉ L) OR (S2 ∉ L) OR (S1 ∈ PostDom(Header(L))) OR
            (S2 ∈ PostDom(Header(L))) THEN
                I = I ∪ B
        END IF
    END IF
END FOR

// Find the iteration branches that can indirectly affect the number of iterations.
DO
    FOR each block B in the loop L DO
        IF (B has two succs S1 and S2) AND (B ∉ I) THEN
            IF (there exists J,K ∈ I) AND (J ≠ K) AND (S1 ∈ PostDom(J)) AND
                (S2 ∈ PostDom(K)) THEN
                I = I ∪ B
            END IF
        END IF
    END FOR
WHILE (any change to I)
```

Figure 3: Finding the Set of Iteration Branches for a Loop

since it has a transition to block 6, which is post-dominated by the loop header. Blocks 3, 5, and 7 are identified as having iteration branches since they have a transition to block 4, which is not in the loop. Block 2 is added to the set of blocks containing iteration branches since it can transfer to either block 3 or block 5, which have been identified as containing iteration branches. In other words, block 2 can indirectly affect the number of iterations of the loop.

Once the blocks containing iteration branches for the loop have been identified, a precedence is established that represents the order that these blocks can be executed on any given iteration of the loop. This precedence relationship can be represented as a Directed Acyclic Graph (DAG). The nodes in the DAG represent the blocks containing the iteration branches and two additional nodes, *continue* and *break*. Figure 4 shows the DAG depicting the precedence relationship between the blocks containing iteration

branches from Figure 2. The construction of the DAG can conceptually be accomplished by starting with the graph representing the loop, replacing all back edges with transitions to *continue*, replacing each transition out of the loop with a transition to *break*, and collapsing all nodes that do not represent iteration branches. The actual implementation of the DAG construction started with only nodes representing *continue*, *break*, and blocks containing iteration branches and used domination and post-domination information to establish the edges between the nodes. This algorithm is essentially a sort and requires $O(I^2)$ complexity, where I is the number of iteration branches in the loop.

4.1.2 When Each Iteration Branch Changes Direction

In this subsection a technique is presented that calculates when each iteration branch can change its result based on the number of loop iterations performed. This technique is similar to those used by other compilers that can calculate the number of iterations of a loop with a single exit [22]. For each iteration branch *vpo* derives the information shown

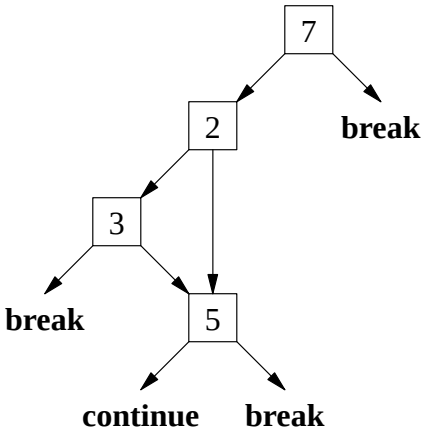


Figure 4: Precedence Relationship between Iteration Branches in Figure 2

in Table 7. When all of the requirements listed in Table 7 are satisfied, the iteration branch is classified as *known*. Otherwise, the iteration branch is classified as *unknown*. Note that detection of *unknown* iteration branches in a loop does not mean that the number of iterations of a loop cannot be bounded. Using the derived values, the compiler applies Equation 1 to straightforwardly calculate on which iteration, N_i, that a *known* iteration branch *i* will change direction. Table 8 shows the values derived for the example in Figure 2. The iteration branch in block 3 is classified as *unknown* since the *variable* `somecond` is not a basic induction variable. The complexity of this algorithm is $O(I)$, where I is the number of iteration branches, since each iteration branch need only be examined once.

$$N_i = \left\lfloor \frac{limit_i - (initial_i + before_i) + adjust_i}{before_i + after_i} \right\rfloor + 1 \qquad (1)$$

In addition, various checks have to be made in case the iteration branch will always or never be satisfied. These checks depend on whether the *limit* is greater or less than the *initial* value, whether the sum of the *before* and *after* values are greater or less than zero, and the relational operator used in the comparison. Figure 5 shows two loops that require special checks. The implementation detects that the loop in Figure 5(a) exits after a single iteration. Recall that the timing analyzer's definition of the number of iterations is the number of times that the loop header block (i.e. testing `i > 100` in the example) is executed once the loop is entered [25]. The loop in Figure 5(b) is classified as *unbounded* since the loop may never exit depending on how overflow of negative integer values is handled.

Table 7: Information Calculated for Each Iteration Branch

Term	Explanation	Requirement
variable	The control variable on which the branch depends, which is the variable being compared in the block containing the iteration branch.	The control variable must be a basic induction variable, which is a variable v whose only assignments within the loop are of the form v := v ± c where c is a constant [39].
limit	The value being compared to the *variable* in the block containing the branch.	The limit must be a constant. Section 4.2 describes how this requirement can be relaxed.
relop	The relational operator used to compare the *variable* and the *limit*.	The initial description requires that the relational operator be an inequality operator (i.e. <, ≤, ≥, and >). Subsection 4.1.5 explains how this restriction is relaxed to handle more accurately the equality operators (i.e. == and !=).
initial	The value of the *variable* when the loop is entered.[3]	The initial value must be a constant. Section 4.2 describes how this requirement can be relaxed.
before	The amount by which the *variable* is changed before reaching the iteration branch in each iteration.	The amount by which the control variable is incremented or decremented must be a constant and these constant changes must occur on each complete iteration of the loop.[4]
after	The amount by which the *variable* is changed after reaching the iteration branch in each iteration.	The amount by which the control variable is incremented or decremented must be a constant and these constant changes must occur on each complete iteration of the loop.
adjust	An adjustment value of 0 or 1, which compensates for the difference between relational operators (e.g. < and ≤).	

Table 8: Derived Information for Each Iteration Branch in Figure 2

branch	variable	register	limit	relop	initial	before	after	adjust	class	N
block 2	j	r[9]	75	<=	1	0	3	1	known	26
block 3	somecond	r[8]	0	==	N/A	0	0	N/A	unknown	N/A
block 5	j	r[9]	300	>	1	0	3	1	known	101
block 7	i	r[10]	100	>=	0	0	1	0	known	101

[3] This value is found by searching backwards in the control flow for assignments to *variable*. The search starts with the preheader, which is the block that has a transition to the loop header and is not in the loop.

[4] In other words, the basic blocks containing these changes must dominate every predecessor block of the header that is in the loop.

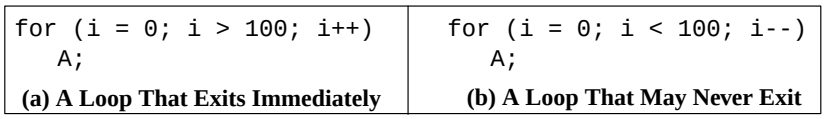


Figure 5: Two Loops Requiring Special Checks

4.1.3 When Each Iteration Branch Can Be Reached

The next step is to determine the iterations on which it is possible to execute each node of the DAG. This information is recorded as a range of iterations and a range is attached to each node and edge. The DAG is processed in a preorder manner (i.e. all predecessors of a node are processed before the node is processed). Calculating these ranges requires $O(I)$ complexity, where I is the number of iteration branches. The head of the DAG is assigned the range [1..∞]. All other nodes are assigned a range that is the union of the ranges of all incoming edges. The outgoing edges of a node i are assigned ranges using one of the following two rules:

(1) If iteration branch i is *known*, then $relop_i$ and the direction of the increment (i.e. the sign of $before_i+after_i$) is used to determine which edge is taken on the first N_i-1 iterations. That edge is assigned the range that is the intersection of [1..N_i-1] and the range of node i. The other outgoing edge is assigned the range that is the intersection of [N_i..∞] and the range of node i. If a range assigned to an outgoing edge is empty, then this edge corresponds to an infeasible transition and is deleted from the DAG.

(2) If iteration branch i is *unknown*, then both outgoing edges are assigned the same range as node i.

Figure 6 shows the DAG of iteration branches in Figure 4 with the range of possible iterations for each node and edge also depicted. Nodes with *known* iteration branches are marked with a **K** and *unknown* iteration branches are marked with a **U**. Iteration branch 7 will take the transition to branch 2 on the first 100 iterations. Note this iteration range of [1..100] corresponds to the variable `i`'s value range of [0..99]. At this point, all

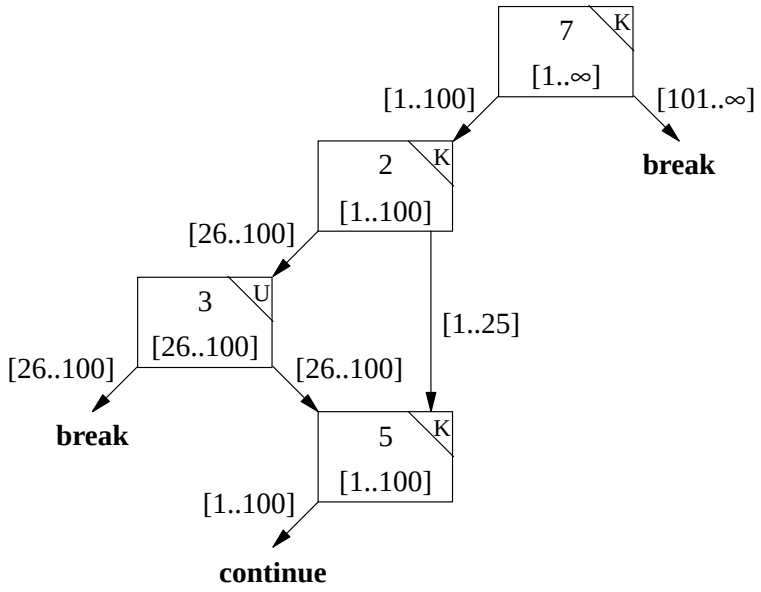


Figure 6: DAG of Branches with Ranges of Iterations

values of variables have been abstracted as ranges of loop iterations. Node 3 is marked with a **U** to denote that its iteration branch is *unknown*. Thus, its two outgoing edges have ranges that match the range in node 3. Node 5's transition to a *break* is deleted since the range associated with that transition is empty (i.e. the transition is not possible).

4.1.4 Determining Minimum and Maximum Iterations

The ranges of iterations associated with each node and edge of the DAG can be used to calculate the minimum and maximum number of iterations for the loop. To determine the minimum and maximum iteration value for each iteration branch, the DAG is processed in a postorder manner (i.e. all successors of the node are processed before the node can be processed), which requires $O(I)$ complexity, where I is the number of iteration branches. The minimum and maximum iteration values for the root node of the

DAG will be the minimum and maximum iteration values for the entire loop. Figure 7 defines the notation used in this subsection. Note that the range has been calculated using the rules defined in Subsection 2.3.

The following rules are used to assign minimum and maximum iteration values to edges.

(1) If an edge is to a *break*, then both the *edge_exit_min* and *edge_exit_max* are assigned the value of *edge_range_min*. (If there is a transition to a *break*, then the loop can only make that transition once.) This is the only point where a *bounded* value can be introduced since these are the only points where the loop can exit.

(2) If an edge is to a *continue*, then the *edge_exit_min* and *edge_exit_max* values for that edge are marked as *unbounded*, which will be denoted by '_'. (These transitions do not supply any information about when the loop exits.)

(3) Otherwise, the incoming edge is to a node representing an iteration branch and the *edge_exit_min* and *edge_exit_max* values assigned to the edge depend upon one of three possible relations between the range of the edge and the iteration values of the node. These

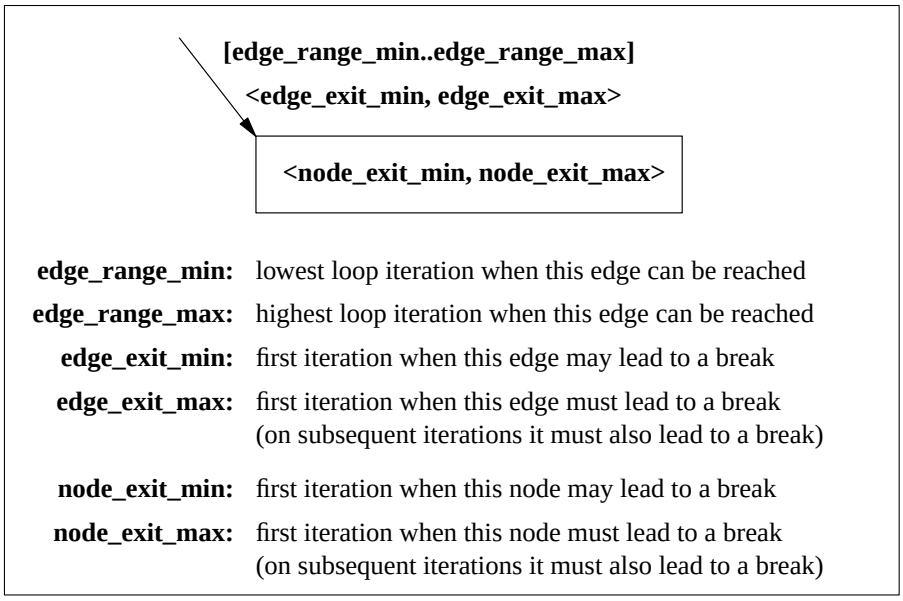


Figure 7: Notation Used in Rules for Assigning Iteration Values

relations and the corresponding edge assignments are depicted in Table 9. For example, the edge assignment when *node_exit_min* satisfies case 1 and *node_exit_max* satisfies case 2 would be <*edge_range_min, node_exit_max*>. Case 1 depicts that the *edge_exit* is set to *edge_range_min* since this is the first iteration the edge can be traversed when the edge may lead to a *break*. Case 2 shows that the *edge_exit* is set to the *node_exit* when it is within the range of iterations that the edge is executed. Case 3 illustrates that the *edge_exit* is set to *unbounded* when there is no iteration on which the edge will be traversed after the edge can lead to a *break*.

The following rules are used to assign minimum and maximum iteration values to nodes.

(1) The *node_exit_min* for a node is set to the smallest of the *bounded edge_exit_min* values on the outgoing edges of the node or is denoted as *unbounded* if both outgoing edges have *unbounded edge_exit_min* values. (The smallest value represents the first possibility to exit the loop.)

(2) If the iteration branch associated with a node is classified as *known*, then the *node_exit_max* for the node is set to the smallest of the *bounded edge_exit_max* values on the outgoing edges or is denoted as *unbounded* if both outgoing edges have *unbounded edge_exit_max* values. (The loop has to exit when it will encounter a *break*.)

(3) If the iteration branch associated with a node is classified as *unknown*, then the *node_exit_max* for the node is set to the largest of the *edge_exit_max* values on the outgoing edges of the node or is denoted as *unbounded* if either outgoing edge has an *unbounded edge_exit_max* value. (Use the largest value when it is not guaranteed that the node will actually reach the exit associated with a lower value.)

Table 9: Rules for Assigning Iteration Values to an Incoming Edge

Case	Condition	Test	Edge_Exit Assignment
1	● ———	node_exit < edge_range_min	edge_range_min
2	——●——	edge_range_min <= node_exit && node_exit <= edge_range_max	node_exit
3	——— ●	edge_range_max < node_exit	—

——— [edge_range_min..edge_range_max]

● node_exit (i.e. node_exit_min or node_exit_max)

Figure 8 shows the same DAG as in Figure 6, but with minimum and maximum iteration values assigned to edges and nodes. The pair of values represented on the edges and in the nodes are the minimum and maximum iteration values, respectively. Node 5 and its incoming edges are assigned *unbounded* values since there is no transition to a *break* for the range of loop iterations in which they are executed. Node 3 is assigned a minimum iteration value of 26 since that is the first possible iteration at which the node can take a transition to a *break*. Node 3's maximum iteration value is *unbounded* since node 3's iteration branch is classified as *unknown* and there is no guarantee that the transition to the *break* from node 3 will ever be taken. The minimum and maximum iterations for the entire loop is 26 and 101, respectively, since these are the iteration values in node 7, which is the root exit condition.

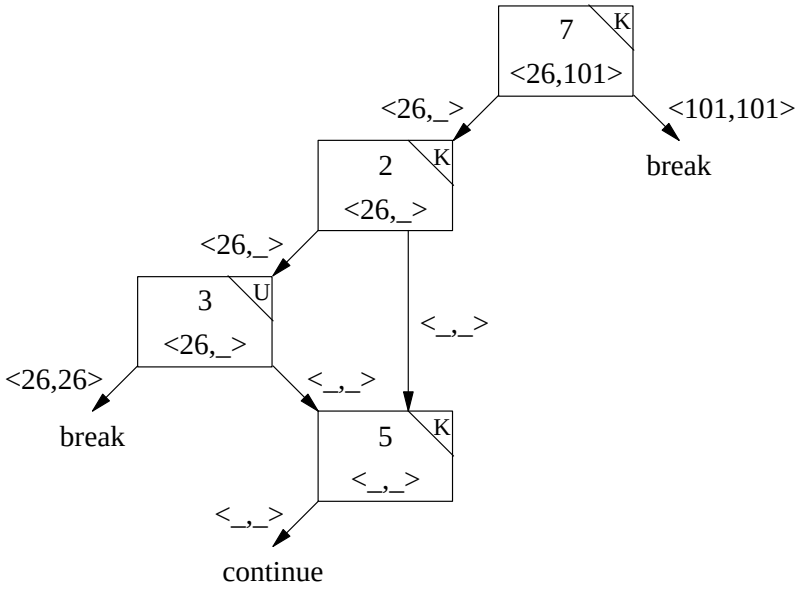


Figure 8: DAG of Iteration Branches with Minimum and Maximum Iterations

4.1.5 Iteration Branches Using Equality Operators

As stated in Table 7, an iteration branch using an equality operator (i.e. == or !=) was initially described as always being treated as an *unknown* branch. This may result in looser, but safe iteration bounds for loops containing these iteration branches. One reason for not addressing iteration branches that use the equality operators is that they may cause loop iteration ranges to become noncontiguous and would complicate the algorithms for bounding the number of iterations. However, in many cases iteration branches with equality operators can be handled using only contiguous ranges of iterations. For instance, Figure 9(a) contains a loop with an equality operator that the implementation was able to successfully bound. The implementation classifies iteration branches with equality operators as *known* when the following three additional requirements to those specified in Table 7 are satisfied. (1) First, every path ending in a back edge in the loop must include the iteration branch. Figure 9(b) shows an example

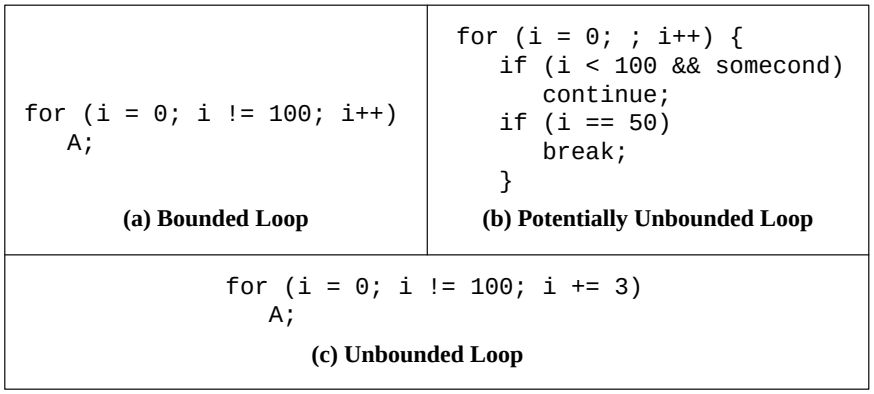


Figure 9: Examples of Loops with Iteration Branches Using Equality Operators

of a loop that may not execute the test for equality on the iteration in which the loop could exit. (2) Next, one of the outgoing transitions of the iteration branch with an equality operator must be to a *break*. (3) Finally, the following expression, which is part of Equation 1, must result in an integral value.

$$\frac{limit_i - (initial_i + before_i)}{before_i + after_i}$$

In other words, the *variable* must equal the *limit* of the iteration branch on some iteration. Figure 9(c) depicts a situation where the *variable* `i` will be assigned values (0, 3, ..., 99, 102, ...) that will skip over the *limit* (100).

4.2 Non-Constant Loop-Invariant Number of Iterations

Sometimes a bounded number of iterations for a loop cannot be determined since the loop exit conditions involve the values of variables. Traditionally, timing analyzers have resolved this problem by requiring a user to specify the maximum number of iterations for a loop interactively [13, 9] or as an assertion in the source code [14, 3]. Unfortunately, there is no guarantee that the user will specify the correct number of iterations. Compilers may employ different code generation strategies or compiler optimizations that can affect the number of loop iterations. Thus, even an astute user may incorrectly specify the number of loop iterations.

All of the variables on which the number of loop iterations depend are frequently loop invariant. In this case, a loop-invariant expression is calculated to represent the number of loop iterations. Essentially, the technique is to use Equation 1 defined in

Subsection 4.1.2, relaxing the requirement that the *limit* and *initial* values have to be constants. Figure 10 shows an example function and corresponding SPARC RTLs. (Some other compiler optimizations, such as loop strength reduction, have not yet been performed to simplify the example.) In this example, the control variable for the loop is `r[13]` and the limit is `r[12]`, which is loop invariant. The block preceding the loop is examined to determine the value associated with the limit, which is expanded in the following steps:

```
1. r[12]                    #  from instruction 12
2. r[9]+r[10]               #  from instruction 5
3. r[9]+R[r[10]+LO[_n]]     #  from instruction 4
4. r[9]+R[HI[_n]+LO[_n]]    #  from instruction 3
5. m+n
```

The register `r[9]` has been allocated to the argument `m`, whose value was also passed to the function in the same register. The compiler remembers the register and the blocks where each live range of a local variable or argument is allocated to a register. Thus, the compiler was able to associate the register `r[9]` with the argument `m` and that the memory reference is to the global variable `n`. The timing analyzer uses Equation 1 to generate a symbolic expression (containing the local variable `m` and global variable `n`) to represent the number of iterations.

$$
\begin{aligned}
N &= \left\lfloor \frac{limit - (initial + before) + adjust}{before + after} \right\rfloor + 1 \\
&= \left\lfloor \frac{m + n - (1 + 1) + 0}{1 + 0} \right\rfloor + 1 \\
&= m + n - 1
\end{aligned}
$$

When the compiler can determine that the number of iterations is non-constant and

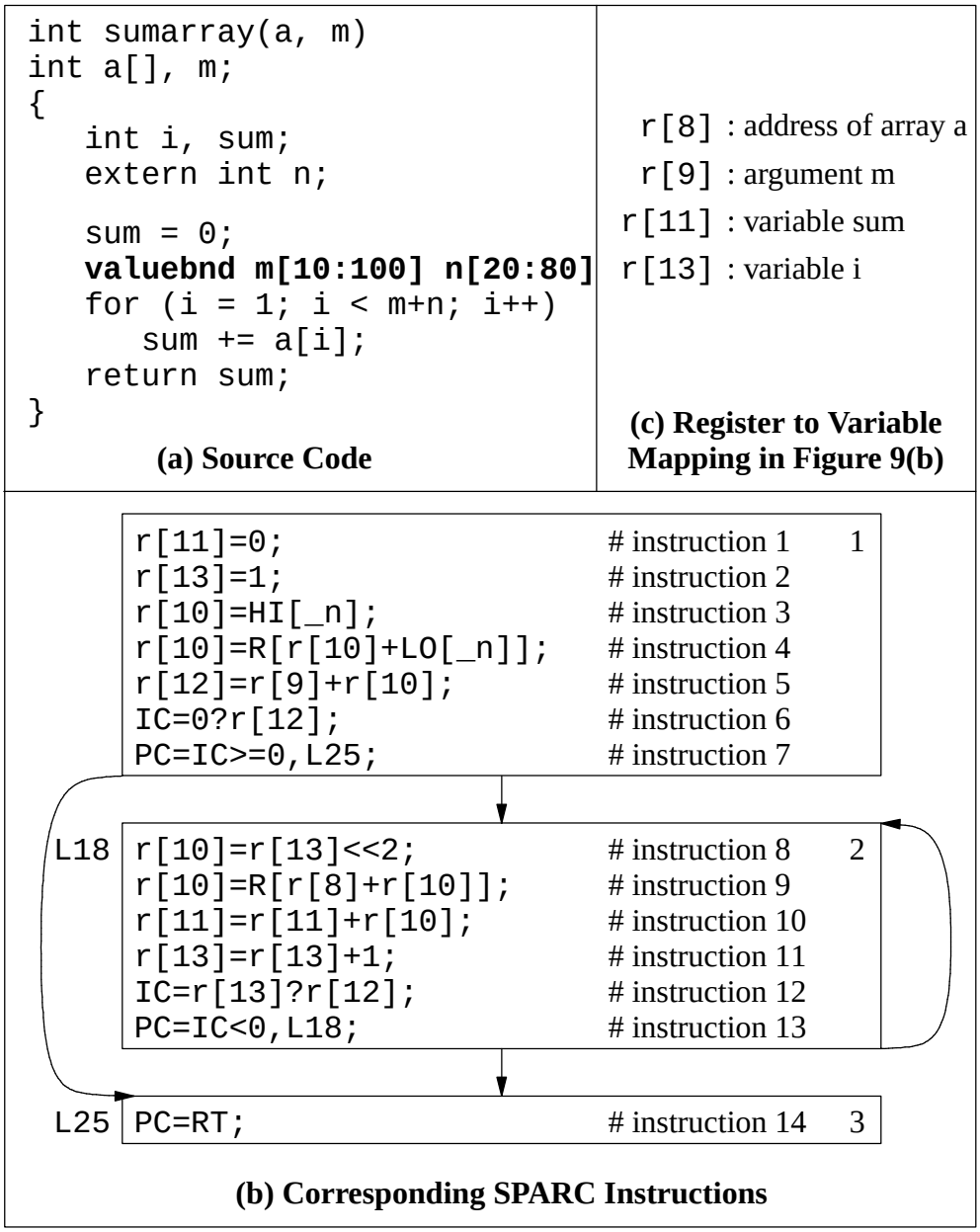


Figure 10: Loop with a Non-constant Loop-Invariant Number of Iterations

loop invariant, the loop-invariant expression is passed to the timing analyzer. The user is prompted by the timing analyzer for the minimum and maximum values for each

variable in this expression. To simplify identification of these variables, the timing analyzer also informs the user of the function and line number associated with the loop. After receiving the minimum and maximum values for these variables, the timing analyzer automatically calculates the minimum and maximum number of loop iterations.[5]

The compiler was also modified to allow the user to specify assertions about the minimum and maximum values of variables associated with loops. The boldface line in Figure 10(a) contains assertions for the minimum and maximum values of the variables `m` and `n`. The compiler uses the loop-invariant expression and replaces the variables with the minimum and maximum specified values. The minimum number of iterations of 29 and the maximum number of iterations of 179 is automatically passed to the timing analyzer and no user intervention is required. Note that the form of a value assertion is analogous to the form of timing constraint loop assertion that can be specified in the same environment [40].

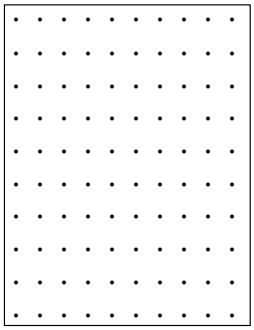
When a loop-invariant expression cannot be calculated, the timing analyzer will prompt the user for the minimum and maximum number of iterations instead of values of variables. However, the author has found that a constant or loop-invariant number of iterations can be typically calculated for most loops in the numerical benchmarks and applications that have been examined.

[5] Note that the timing analyzer will not permit the number of iterations to be fewer than 1. In the above example, a user may indicate that the minimum values of `m` and `n` are both 0. Simply substituting these values in the expression would result in the number of loop iterations being –1. But if the loop is entered, then it has to execute at least one iteration since the number of iterations is defined as the number of times the loop header block is executed.

4.3 Bounding Iterations for Non-Rectangular Loops

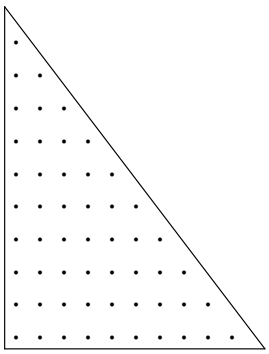
The approaches described so far do not address loops whose number of iterations can vary. The previous sections described approaches to determine the minimum and maximum number of iterations for a loop, given that the number of iterations depends only upon either constant or loop-invariant values. Figure 11 shows two simple loops to depict the essential difference between a rectangular and non-rectangular loop nest. For rectangular loops, the lower and upper bounds of the loop index variables are constant. In contrast, the number of iterations of a non-rectangular loop varies. For instance, the number of iterations typically depends on the values of counter variables from outer loops. These loops have long presented a problem for timing analyzers since the resulting timing predictions are typically quite loose. In fact, these predictions may indicate that a program does not meet its timing constraints, when it actually does.

```
for (i = 0; i < 10; i++)
   for (j = 0; j < 10; j++)
```



(a) Rectangular Loop Nest

```
for (i = 0; i < 10; i++)
   for (j = i; j < 10; j++)
```



(b) Non-rectangular Loop Nest

Figure 11: Rectangular versus Non-Rectangular Loop Nest

The author's initial approach to calculating loop iterations only dealt with doubly nested loops that were triangular in nature as the loop nest in Figure 11(b) [31]. This simple approach has been superseded by a more general approach presented here that is not limited to the nesting depth or how the individual loop index variables depend on one another.

This section describes a general and efficient method for obtaining tight timing predictions for non-rectangular loops usually encountered in programs. This is accomplished by formulating the number of loop iterations in terms of summations, where each summation represents the number of iterations to be executed by a loop. Such an equation can be efficiently solved given that certain restrictions are met.

This work on bounding iterations for non-rectangular loops was inspired by the work of Sakellariou [41, 42]. He calculated the total number of iterations for loops that are dependent on counter variables of outer loops in order to obtain better load balance by assigning approximately the same number of loop iterations to each processor. The approach used was to formulate summations representing the number of loop iterations by hand and to interface to a mathematical package off line to solve the equations. This section describes an approach to automatically calculate the average number of times that a loop will iterate during the timing analysis of a program and to use this information to obtain tighter timing predictions.

4.3.1 Formulating the Number of Iterations

This subsection shows how a loop nest may be formulated in terms of summations. This

framework was based on work by Sakellariou [41, 42]. The number of iterations of a single loop, where the loop variable is incremented by one (unit stride), can be represented by a summation when the lower bound (a) is less than or equal to the upper bound (b), as shown in Equation 2. Figure 12 shows how two different loop nests can be formulated in terms of summations. The total number of iterations to be executed by the innermost loop in each loop nest are calculated by solving the corresponding equation. The Bernoulli formula shown in Equation 3, where $p \geq 1$ & $n \geq 1$ and B_k is a Bernoulli number of order k, can be used to evaluate terms in a summation.

The constraint on the bounds in Equation 2 results from the fact that the value of the sum must equal 0 if the lower bound a is greater than the upper bound b. This constraint is in accordance with the usual semantics of summations in conventional mathematical notation, in which the upper bound is implicitly assumed to be greater than or equal to the lower bound. Therefore, the explicit constraint is necessary to accurately count the number of iterations of so-called *zero-trip* loops. Zero-trip loops do not execute the loop body when the lower bound exceeds the upper bound, given that the stride is positive.

It is possible to represent summations with non-unit strides, where the stride s is specified along with the lower bound a and upper bound b. Equation 4 shows how a non-unit stride can be used in a conventional summation, where e is an expression and $e[i \leftarrow si + a]$ denotes the substitution of all free occurrences of i by $si + a$. This way, summations with strides can be represented by uniform summations.

$$N = \sum_{i=a}^{b} 1 = \begin{cases} b - a + 1 & \text{if } a \le b \\ 0 & \textit{otherwise} \end{cases} \tag{2}$$

$$\sum_{i=1}^{n} i^p = \frac{1}{p+1} \sum_{k=0}^{p} \binom{p+1}{k} B_k (n+1)^{p-k+1} \tag{3}$$

```
for (i=1; i<99;
     i++)
   for (j=i+1;
        j<100;
        j++)
```

$$N = \sum_{i=1}^{98} \sum_{j=i+1}^{99} 1$$
$$= \sum_{i=1}^{98} (\sum_{j=1}^{99} 1 - \sum_{j=1}^{i} 1)$$
$$= \sum_{i=1}^{98} (99 - i)$$
$$= \sum_{i=1}^{98} 99 - \sum_{i=1}^{98} i$$
$$= 4,851$$

(a) Loop Nest from Sort Program

```
for (j=1; j<=100; j++)
   for (i=j; i<=100; i++)
      for (k=1; k<j; k++)
```

$$N = \sum_{j=1}^{100} \sum_{i=j}^{100} \sum_{k=1}^{j-1} 1$$
$$= \sum_{j=1}^{100} \sum_{i=j}^{100} (j-1)$$
$$= \sum_{j=1}^{100} (\sum_{i=1}^{100} (j-1) - \sum_{i=1}^{j-1} (j-1))$$
$$= \sum_{j=1}^{100} (\sum_{i=1}^{100} j - \sum_{i=1}^{100} 1 - \sum_{i=1}^{j-1} j + \sum_{i=1}^{j-1} 1)$$
$$= \sum_{j=1}^{100} (102j - j^2 - 101)$$
$$= 102 \sum_{j=1}^{100} j - \sum_{j=1}^{100} j^2 - \sum_{j=1}^{100} 101$$
$$= 166,650$$

(b) Loop Nest from LU Decomposition Program

Figure 12: Deriving the Total Number of Iterations for Two Loop Nests

$$I = \sum_{i=a}^{b,s} e = \sum_{i=0}^{\lfloor (b-a)/s \rfloor} e\,[i \leftarrow si + a] \tag{4}$$

Summations with non-unit strides are more difficult to evaluate since one has to deal with summations of floors. Equation 5 shows how a floor can be converted to an

expression involving a modulo operation (*%*). A modulo operation can often be simplified using Equation 6 [42]. However, summations involving modulo operations are more difficult to simplify when two or more loops have non-unit strides and the bounds are symbolic. Fortunately, this situation rarely occurs. Equations 2-6 can be used to correctly determine that the total iterations for the loop nest in Figure 13 is 1,717. Unfortunately, sometimes an expression in a summation may contain a product of two or more terms containing modulo operations. In this case, an approximation of the iteration count is used, which is shown in Equation 7.

$$\left\lfloor \frac{n}{m} \right\rfloor = \frac{n - n\%m}{m}, \text{ if } m > 0 \ \& \ n > 0 \tag{5}$$

$$\sum_{i=0}^{n} (i\%d)^p = \begin{cases} \sum_{i=0}^{n} i^p & \text{if } n < d \\ \sum_{j=0}^{\lfloor n/d \rfloor - 1} \sum_{i=0}^{d-1} i^p + \sum_{i=0}^{n\%d} i^p & \text{if } n \geq d \end{cases} \tag{6}$$

$$\sum_{i=a}^{b,s} e \approx \sum_{i=a}^{\lfloor b/s \rfloor} e/s \tag{7}$$

As suggested by Sakellariou [41, 42], a computer algebra system can be exploited off line to solve the equations of summations. However, computer algebra systems, such as *Maple*, give inaccurate results when the bounds restriction on the summation is violated.

```
for (i=0; i<100; i++)
   for (j=i; j<100; j+=3)
```

Figure 13: A Loop Nest Containing a Non-unit Stride

In general, every loop iteration count problem that is cast as a summation should evaluate to zero if the lower bound is greater than the upper bound. However, it is not always possible to evaluate the test when the bounds are symbolic. For example, consider the loop nest in Figure 14. The inner loop is a zero-trip loop for values of i greater than 2. A *partially zero-trip* is defined to be a loop that is zero-trip depending on values of index variables of outer loop(s). By applying Equation 8, the iteration count of the partially zero-trip loop can be defined as shown in Figure 14. Clearly, the result is $N = 3$. However, a naive evaluation without the bounds test results in $N = -7$. This means that when a computer algebra system is to be used off line, the summations should be guarded with bounds tests. Unfortunately, computer algebra systems cannot effectively deal with the simplification of nested summations with additional tests on the bounds of inner summations. The reason is that the test may be symbolic, as shown in Figure 14. The solution is to isolate possible conditions on the iteration variable from the test and to simplify summations as shown in Equation 8 for any expression e. Note that c may not necessarily lie within the range $[a..b]$ and relations besides $<$ may be used.

```
for (i=1; i<8; i++)
    for (j=i; j<3; j++)
```

$$N = \sum_{i=1}^{7} \begin{cases} 3 - i & \text{if } i < 3 \\ 0 & otherwise \end{cases}$$

Figure 14: A Partially Zero-Trip Loop

$$\sum_{i=a}^{b} \begin{cases} e & \text{if } i < c \\ 0 & otherwise \end{cases} = \begin{cases} \sum_{i=a}^{min(b,c)} e & \text{if } a < c \\ 0 & otherwise \end{cases} \quad (8)$$

4.3.2 Implementation

The implementation for evaluating the summations described in the previous section was accomplished by using the General-Purpose Algebraic Simplifier (GPAS) portion of the Ctadel system [43, 44]. The author's timing analyzer [2] and Ctadel were compiled separately, but Ctadel is directly integrated into the timing analyzer by linking the object files. This avoids unnecessary overhead that would result from passing messages between the timing analyzer and GPAS if they were different processes. The summations are formulated in the timing analyzer and GPAS is invoked as a C function with the summation parameters as arguments.

Another complication when dealing with zero-trip loops in the timing analyzer is due to the way the timing analyzer counts iterations. As mentioned in Section 3.2, the number of loop iterations is the number of times the loop header is executed, as opposed to the number of times the loop body is encountered. Thus, when a loop is entered, it is guaranteed to iterate at least once. The zero-trip case in Equation 8 can be modified to indicate a single iteration, as shown in Equation 9. Figure 15 shows how the loop nest in

$$\sum_{i=a}^{b} \begin{cases} e & \text{if } i < c \\ 1 & otherwise \end{cases} = \begin{cases} \sum_{i=a}^{min(b,c-1)} e & \text{if } a < c \\ 0 & otherwise \end{cases} + \begin{cases} \sum_{i=max(a,c)}^{b} 1 & \text{if } c \le b \\ 0 & otherwise \end{cases} \quad (9)$$

Figure 14 can be formulated as a summation and solved to produce an accurate number of iterations. Note that the test in Figure 15 has iteration variable *i* isolated to the left of the relation. In practice, however, an isolation algorithm is used by GPAS to analyze the test and isolate the variable.

It is known that the detection of zero-trip loops in the general case is NP-complete, because it amounts to solving a linear programming problem. Similarly, adjusting the bounds of loops to avoid partially zero-trip loops is NP-complete. This normalization process can be performed with the Fourier-Motzkin (FM) elimination method [45]. However, one can argue that real-world algorithms rarely exhibit (partially) zero-trip loops, because algorithms with partially zero-trip loops are deemed to be inefficient.

The timing analyzer verifies that there are no zero-trip loops for an inner loop by expanding its initial value and limit. Likewise, the timing analyzer is able to verify that there are no partially zero-trip loops in the loop nest. However, if the verification is inconclusive, the loop nest may or may not contain (partial) zero-trip loops. For

$$
\begin{aligned}
N &= \sum_{i=1}^{7} \begin{cases} 3-i & \text{if } i < 3 \\ 1 & \textit{otherwise} \end{cases} \\
&= \sum_{i=1}^{2} (3-i) + \sum_{i=3}^{7} 1 \\
&= 3+5 \\
&= 8
\end{aligned}
$$

Figure 15: Deriving the Number of Iterations for the Loop Nest in Figure 14

instance, consider the loop nest in Figure 16. The expansion of the innermost loop initial value and limit is depicted in Table 10. The timing analyzer is able to guarantee that the inner loop is not zero-trip since the initial value is never greater than the limit.

Now consider the loop in Figure 17 and the corresponding expansion of the initial value and limit in Table 11. The test is inconclusive. However, the loop nest is not zero-trip due to the $j<i$ condition in the middle loop. Since the range analysis can be used to safely verify if a loop is partially zero-trip, it is possible to use the results in deciding which summation solver to use. For example, the loop in Figure 16 can be safely cast into a summation without a bounds tests, while the summations for the loop in Figure 17 requires a bounds test (see Figure 15 for an example bounds test). The disadvantage of having a bounds test is that a loop with a stride poses problems for solving the summation because the summation bounds test may contain modulo operations on the iteration variable, which prohibits the application of Equation 9.

The timing analyzer decides among three possible solution methods to evaluate the summation representing a loop nest:

(1) GPAS evaluates the summation without testing the bounds of the index variables.
(2) GPAS evaluates the summation while testing the bounds.
(3) The timing analyzer derives conservative lower and upper bounds on the sum, based on constant bounds given in outer level loops.

The algorithm for selecting the appropriate method is described in Figure 18. The exact solutions are computed using safe assumptions in the possible presence of partially zero-trip loops, using either method (1) or (2). This algorithm will resort to method (3) only

```
for (i=0; i<10; i++)
  for (j=i; j<11; j++)
    for (k=i-3; k<j+8; k++)
```

Figure 16: Innermost Loop Detected Zero-Trip Free by the Timing Analyzer

Table 10: Expanding Initial and Limit Values of Innermost Loop in Figure 16

Initial Value	Limit
i–3 [0..9]–3 [–3..6]	j+8 [i..10]+8 [[0..9]..10]+8 [0..10]+8 [8..18]

```
  for (i=1; i<10; i++)
    for (j=0; j<i; j++)
      for (k=j; k<i; k++)
```

Figure 17: Innermost Loop Nest Detected Zero-Trip Free by GPAS

Table 11: Expanding Initial and Limit Values of Innermost Loop in Figure 17

Initial Value	Limit
j [0..i] [0..[1..9]] [0..9]	i–1 [1..9]–1 [0..8]

in the presence of multiple loops with non-unit strides, in which the strides are relatively prime and the bounds on the index variables are not all constant.

The following approach is used in the timing analyzer to obtain tight predictions of non-rectangular loops whose total iterations in a loop nest are known. The timing analyzer calculates WCET and BCET predictions based on the maximum and minimum

The timing analyzer attempts to determine if the loop nest is not (partially) zero-trip.

IF the check is successful **THEN**
    The loop nest is formulated into summation without bounds tests and presented to GPAS.
ELSE
    The check is inconclusive and the loop nest is cast into a summation with bounds tests.
    The rewritten summation is presented to GPAS.

IF GPAS is able to solve the summation **THEN**
    RETURN the integer count.
ELSE
    GPAS could not solve the summation in the presence of two or more loops with non-unit strides.
    RETURN conservative bounds on the sum.

Figure 18: Algorithm for Selecting a Solution Method for Summations

number of iterations for a non-rectangular loop, respectively. These predictions are made in case a user requests the WCET or BCET predictions for the loop. In addition to these absolute predictions, the timing analyzer also calculates *average* WCET and BCET predictions for each loop. To calculate the average number of iterations for a loop, the timing analyzer divides the total iterations by the total number of times the loop is entered. For instance, the total number of iterations for the innermost loop from the *sort* program in Figure 12 was 4,851. The timing analyzer also calculates the number of times the current loop is entered by calculating the total number of iterations for the loop that encloses the current loop. In the example shown in Figure 12, the innermost loop is entered 98 times. Thus, the average number of iterations for the loop is 49.5 (4,851/98). The average number of iterations is used to calculate the average WCET and BCET predictions. When a non-integer is calculated, the timing analyzer rounds up for the

WCET prediction and truncates for the BCET prediction since the loop analysis algorithm is designed to work on an integral number of iterations.

4.4 Results

Table 12 shows the results of the timing analysis taking into account loop iteration constraints. The columns labeled "+ Pipelining" show the results if the timing analyzer did not perform the loop iteration constraint analysis described in this chapter. The "+ Iter. Count" columns show that the timing predictions become much tighter when these loop iterations constraints are taken into account. In particular, the lines printed in boldface indicate the programs whose timing predictions became tighter as a result of this additional analysis.

Among the programs listed in Table 6, *Hes*, *Integ*, *Interp*, *LU*, *Sort* and *Sym* benefit from using this approach since they each contain one or more non-rectangular loops. *Interp* showed a significant improvement in best case since the best case number of iterations for a non-rectangular inner loop was 1, which was significantly lower than the average number of iterations. If the timing analyzer did not use an average number of inner loop iterations in worst case, then the number of loop iterations for the triangular loops in *Interp*, *Sort*, and *Sym* would have been approximately double. The WCET of these programs are nearly exact using the average number of iterations. The *Integ* program had a higher best-case "+ Pipelining" ratio and a lower worst-case "+ Pipelining" ratio since there were other loops in this program that contributed more significantly to the total execution time. The *Sort* and *Sym* programs did not have a

Table 12: Results After Adding Accurate Iteration Counts

Worst-Case Results					
Name	Observed Cycles	+ Pipelining Cycles	+ Pipelining Ratio	+ Iter. Count Cycles	+ Iter. Count Ratio
Des	149,706	172,509	1.152	172,509	1.152
Expint	58,217	1,293,290	22.215	1,293,290	22.215
Fresnel	47,749	48,887	1.024	48,887	1.024
Gaujac	786,786	790,116	1.004	790,116	1.004
Hes	**55,834,609**	**130,574,296**	**2.339**	**56,739,136**	**1.016**
Integ	**22,538,082**	**30,023,163**	**1.332**	**22,553,163**	**1.001**
Interp	**25,469,403**	**50,701,362**	**1.991**	**25,478,409**	**1.000**
LU	**23,055,832**	**124,577,237**	**5.403**	**23,572,337**	**1.022**
Matcnt	1,769,321	1,861,150	1.052	1,861,150	1.052
Matmul	4,444,911	4,448,212	1.001	4,448,212	1.001
Matsum	1,277,465	1,279,322	1.001	1,279,322	1.001
Sort	**7,672,281**	**251,603**	**1.988**	**7,672,292**	**1.000**
Sprsin	28,339	28,664	1.011	28,664	1.011
Stats	1,016,048	1,016,128	1.000	1,016,128	1.000
Summidall	15,340	18,090	1.179	18,090	1.179
Summinmax	16,080	17,080	1.062	17,080	1.062
Sumnegpos	11,067	13,068	1.181	13,068	1.181
Sumoddeven	15,093	16,112	1.068	16,112	1.068
Sym	**2,747,654**	**5,481,220**	**1.995**	**2,747,708**	**1.000**
Average	7,734,420	19,347,974	2.631	7,882,403	2.157

Best-Case Results					
Name	Observed Cycles	+ Pipelining Cycles	+ Pipelining Ratio	+ Iter. Count Cycles	+ Iter. Count Ratio
Des	65,615	22,247	0.339	22,247	0.339
Expint	125	118	0.944	118	0.944
Fresnel	181	172	0.950	172	0.950
Gaujac	45,270	44,566	0.984	44,566	0.984
Hes	**306,733**	**14,006**	**0.046**	**258,908**	**0.844**
Integ	**19,160,842**	**12,808,073**	**0.668**	**19,135,118**	**0.999**
Interp	**6,485,878**	**143,064**	**0.022**	**6,479,865**	**0.999**
LU	**12,883,939**	**284,011**	**0.022**	**637,365**	**0.049**
Matcnt	1,549,095	1,548,798	1.000	1,548,798	1.000
Matmul	4,444,666	4,420,068	0.994	4,420,068	0.994
Matsum	1,257,239	1,167,140	0.923	1,167,140	0.923
Sort	19,966	19,950	0.999	19,950	0.999
Sprsin	17,436	17,379	0.997	17,379	0.997
Stats	607,399	601,406	0.990	601,406	0.990
Summidall	15,340	8,072	0.526	8,072	0.526
Summinmax	13,080	13,062	0.999	13,062	0.999
Sumnegpos	9,067	9,049	0.998	9,049	0.998
Sumoddeven	94	63	0.670	63	0.670
Sym	160	160	1.000	160	1.000
Average	2,467,480	1,111,653	0.741	1,809,658	0.853

significant underestimation (i.e. "+ Pipelining" ratio) in best case. In the best case for *Sort* the values were initially sorted and the sort function exited once the array has been detected to be in ascending order. Likewise, the *Sym* program terminates when it finds the first pair of values that are not equal.

Hes and *LU* are unlike the other programs in that they contain some triply nested loops. In some loop nests the loop variables of the innermost and middle loops depend on the outermost index variable. In other loop nests the innermost loop variable depends on the loop variable of the middle loop, which in turn depends on the loop variable of the outer loop. GPAS correctly determines the exact number of loop iterations in all of these cases and the results are more accurate WCET predictions compared to its "+ Pipelining" ratios. When the timing analyzer computes the number of iterations of a loop in a non-rectangular nest, it is sometimes necessary to round this number to an integer. The programs *Hes*, *Integ* and *LU* each contain two or more loops in which this rounding causes slightly conservative predictions. For example, in the case of *LU*, there were two inner loops that were triply nested. When computing the number of iterations of these loops, the rounding of iterations introduced a 1-2% overestimation in worst case and a 1-3% underestimation in best case. Together these two loops comprised 89.8% of the instructions executed in the program. Another reason for the conservative predictions for *LU* is that when the number of iterations of nested loops is averaged, the resulting execution time prediction times may be looser since each iteration may have a different execution time. In worst case, for each iteration *i* until the last iteration, the

execution time of *i* is greater than or equal to that of *i*+1. Thus, the execution times for successive iterations are monotonically decreasing or stay the same. Analogously, in best case, the execution times are monotonically increasing or stay the same for successive iterations.

4.5 Conclusions

This chapter has presented three different methods for bounding the number of iterations of a loop. First, a method was described that determines the minimum and maximum number of iterations of loops with multiple exits and also detects infeasible paths. For instance, loops of the form in Figure 19(a) that can exit prematurely when some condition becomes true are quite common and the bounded number of iterations of such loops can be detected by the general algorithm presented in this chapter.

Second, a non-constant loop-invariant number of iterations is calculated when the

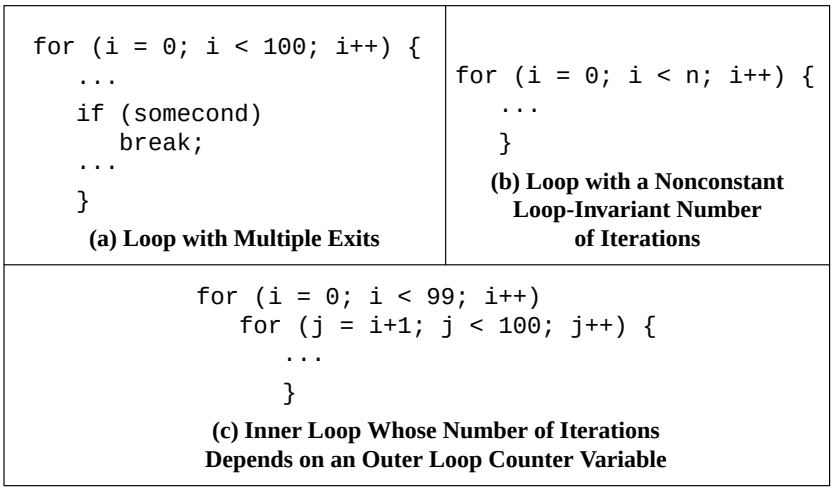


Figure 19: Common Forms of Loops

variables on which the number of iterations depends cannot change values inside of the loop. Figure 19(b) depicts an example of this common type of loop. The user can specify the minimum and maximum values of these variables by placing assertions in the source code or by interactively responding to prompts from the timing analyzer. These assertions are more reliable than specifying the minimum and maximum number of loop iterations directly since the user does not have to be aware of the code generation strategies or optimizations performed by the compiler.

Finally, timing analysis support is given to tightly predict the execution time of non-rectangular loops whose number of iterations is dependent on counter variables of outer level loops. These loops, such as the one shown in Figure 19(c), appear frequently in programs and can result in significant underestimations in best-case predictions and overestimations in worst-case predictions. Using the methods of this chapter, it is possible to more tightly predict loops when the initial value or limit of the control variable in an inner loop depends on a control variable of an enclosing outer loop.

These methods have been successfully integrated in an existing compiler and an associated timing analyzer that predicts the performance for optimized code on a machine that exploits caching and pipelining. The result is tighter and more reliable timing analysis predictions and less work for the user.

CHAPTER 5

BRANCH CONSTRAINT DETECTION AND EXPLOITATION

Even with perfect architectural modeling and a correct calculation of the number of loop iterations, significant overestimations of WCET and underestimations of BCET can still occur. The reasons for the loose timing predictions are due to dependences on data values that can constrain the outcome of conditional branches and restrict the set of paths that can be taken. While branch constraint information has been used in the past by some timing analyzers, it has typically been specified manually, which is both tedious and error prone. This chapter describes how branch constraints can be automatically detected by a compiler and exploited by a timing analyzer.

5.1 Automatic Detection of Constraints

A branch constraint causes the outcome of a conditional branch to be known under certain conditions. The compiler employs techniques to detect these constraints, which are classified as *effect-based* and *iteration-based*.

5.1.1 Detecting Effect-Based Constraints

The compiler performs analysis to determine if the outcome of a conditional branch is known at any given point in the control flow. First, the compiler calculates the set of registers and variables upon which a branch (and its associated comparison) depends. This set is calculated by expanding the effects of the comparison instruction associated

with the branch. For instance, consider the SPARC instructions represented as RTLs (Register Transfer Lists) and the associated expanded comparison in Figure 20. A comparison is expanded by searching backwards for assignments to registers in the comparison until all registers are replaced or the beginning of a block with multiple predecessors is encountered. Loop-invariant registers in the expression are expanded from the preheader of the loop in which they are assigned values. Next, the compiler determines the set of effects associated with assignments to registers and variables for each basic block. Each branch is examined to see if it could be affected by the block. Thus, the compiler can determine that a basic block updating the global variable `g` could affect the result of the branch in Figure 20. Updates to the registers `r[1]` (`%g1`) or `r[8]` (`%o0`) would have no effect.

A state is associated with each conditional branch, which can have one of three values: *unknown*, *fall-through*, or *jump*. The compiler determines if a branch becomes known by substituting the value assigned for the variable or register and evaluating the expanded comparison. The compiler then issues a directive to the timing analyzer for each branch placed in an *unknown*, *fall-through*, or *jump* state by an effect in the block.

Instructions in a Basic Block

```
r[1]=HI[_g];          /* sethi %hi(_g),%g1        */
r[8]=R[r[1]+LO[_g]]; /* ld    [%g1+%lo(_g)],%o0 */
IC=r[8]?5;            /* cmp   %o0,5              */
PC=IC<0,L20;          /* bl    L20                */
```

Expanded Comparison

```
IC=R[HI[_g]+LO[_g]]?5;
```

Figure 20: Example of Expanding a Comparison

Thus, this analysis requires $O(B*C)$ complexity, where B is the number of basic blocks and C is the number of conditional branches. A more complete explanation for detecting branch states has been described in previous work [46].

Consider the source code in Figure 21(a). The corresponding control flow that is

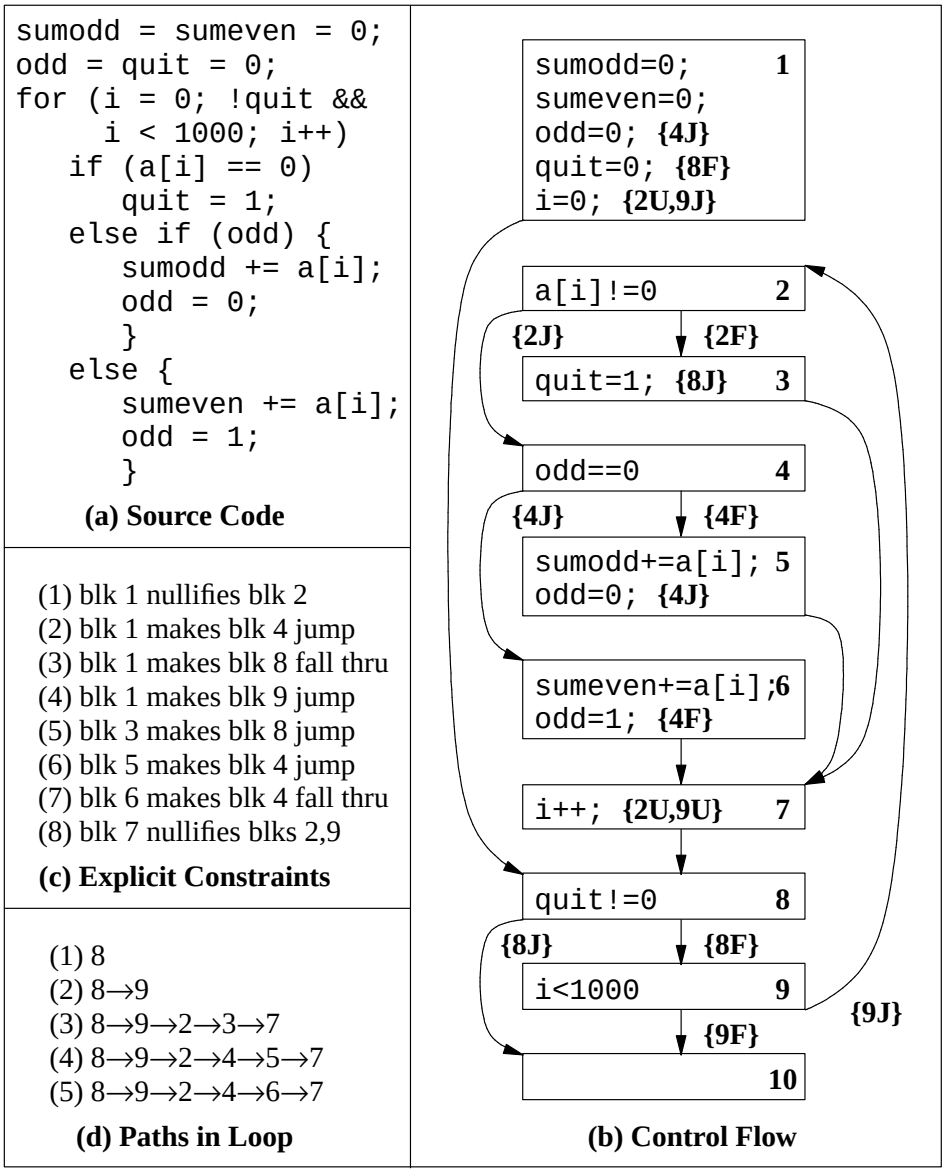


Figure 21: Effects of Assignments on Branches

generated by the compiler is shown in Figure 21(b). While the control flow in the figure is represented at the source code level, the analysis is performed by the compiler at the machine instruction level after compiler optimizations are applied to provide more accurate timing predictions. Note that some branches in Figure 21(b) have conditions that are reversed from the code in Figure 21(a) to depict the branch conditions that are evaluated at the machine instruction level. Only when the condition associated with a branch in a block is evaluated to be true will the jump (**J**) occur. If the condition is not true, then control will fall (**F**) into the next sequential block. The control flow also shows the effect-based constraints, which are enclosed in curly braces and associated with basic blocks or control-flow transitions. Figure 21(c) describes the explicit branch constraints that are automatically detected by the compiler and passed to a timing analyzer. The initialization of `i` in block 1 (`i=0;`) puts the branch in block 2 (`a[i]!=0`) in an *unknown* state (**2U**) and the branch in block 9 (`i<1000`) in a *jump* state (**9J**). In addition, the assignments to `odd` in blocks 1 and 5 (`odd=0;`) and in block 6 (`odd=1;`) cause the branch in block 4 (`odd==0`) to *jump* (**4J**) and *fall through* (**4F**), respectively. Likewise, the assignment to `quit` in blocks 1 (`quit=0;`) and 3 (`quit=1;`) cause the branch in block 8 (`quit!=0`) to *fall through* (**8F**) and *jump* (**8J**), respectively. Finally, the increment of `i` in block 7 (`i++;`) sets the states of the branches in blocks 2 (`a[i]!=0`) and 9 (`i<1000`) to *unknown* (**2U**,**9U**) since they depend on the value of `i`.

Figure 21(b) also shows implicit branch constraints. When a branch has a given outcome, then it will have the same outcome again unless the variables or registers being compared are affected. Thus, a fall-through (**F**) or jump (**J**) transition from a branch will implicitly cause that same branch to be in a *fall-through* or *jump* state, respectively.

These implicit constraints are not explicitly passed to a timing analyzer since a timing analyzer can create them when it is performing analysis on paths.

The source code in Figure 22(a) and corresponding control flow in Figure 22(b) depict a situation where one conditional branch may be logically correlated with another branch. In other words, the direction taken by one conditional branch may indicate the direction taken by another conditional branch. If block 2 (`a[i]>=0`) falls into block 3, then the value of `a[i]` is negative and block 5 (`a[i]<=0`) must jump to block 7 (**5J**).

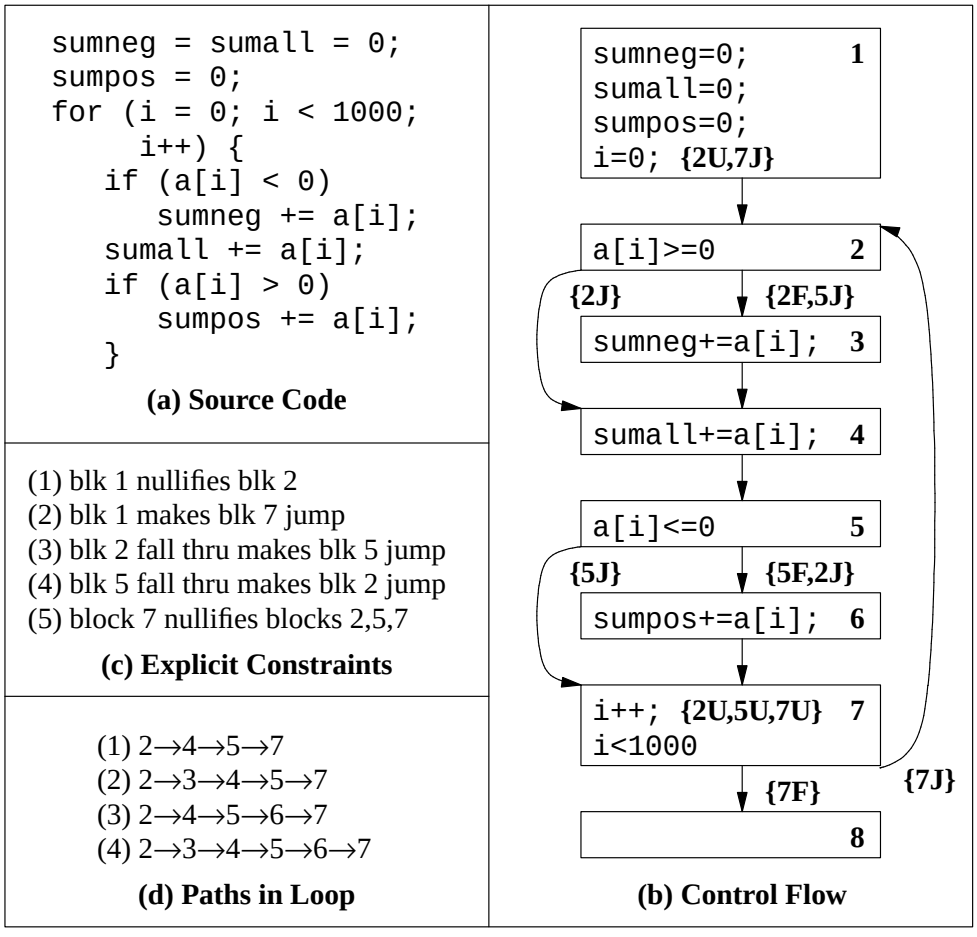


Figure 22: Logical Correlation between Branches

This is described by branch constraint 3 in Figure 22(c). Note that if block 2 (`a[i]>=0`) jumps to block 4, there is no guarantee that block 5 (`a[i]<=0`) will fall through to block 6 since the value of `a[i]` could have been zero. The compiler evaluates each pair of branches in a function to determine if there is a logical correlation between one branch and another. Thus, this analysis requires $O(C^2)$ complexity, where C is the number of conditional branches. Note that a branch is always logically correlated with itself and these self correlations are implicit constraints. The exact conditions when one branch is logically correlated with another have been described in previous work [46].

5.1.2 Detecting Iteration-Based Constraints

A basic induction variable is a variable or register that is incremented or decremented by a constant value on each iteration of a loop [39]. Some branches compare a basic induction variable to a constant. In these situations, the compiler can determine the ranges of iterations in which such a branch will fall through or jump. The compiler produces directives for a timing analyzer that indicate ranges of iterations for each of the two outgoing edges of the block containing the branch. The manner in which this information is derived was described in Section 4.1.3.

Consider the source code and corresponding control flow shown in Figures 23(a) and 23(b). While `i` can range from 0..999 as each path in the loop is entered, the number of corresponding iterations in the loop will range from 1..1000. Thus, the compiler associates ranges of iterations with transitions from blocks that compare basic induction variables to constants. For instance, block 3 (`i<=249`) will only fall through to block 4 when the loop is performing the last 750 iterations (**[251..1000]**). Constraints 5-8 in

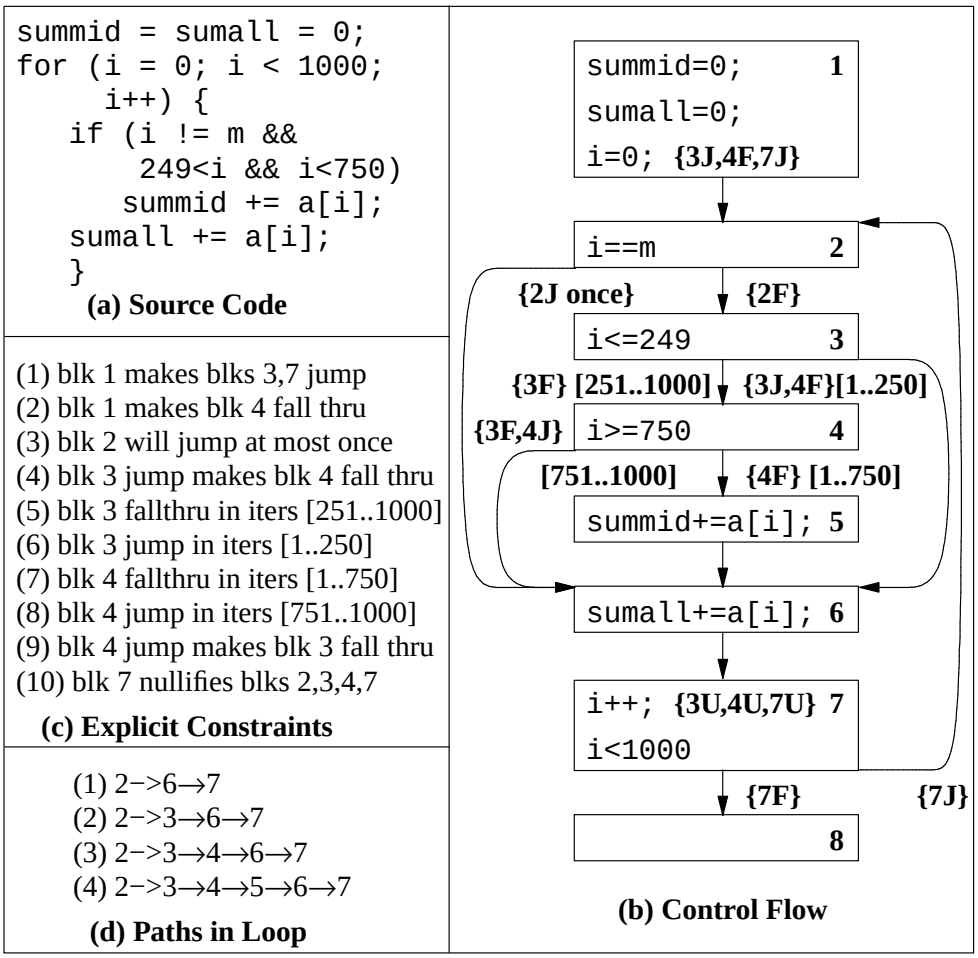


Figure 23: Ranges of Iterations and Branch Outcomes

Figure 23(c) depict the range of iterations when various transitions in the loop can be taken. An implicit iteration-based constraint is that the header of the loop (block 2 in Figure 23(b)) can be executed in every loop iteration (**[1..1000]** for Figure 23). Sometimes a basic induction variable is compared to a non-constant loop invariant value, as shown in block 2 (`i==m`) of Figure 23(b). The value of `m` is not known, but it is invariant with respect to the loop. When the comparison of such a branch is an equality test (== or !=), then the transition that occurs when the two values are equal can take

place at most once for each execution of the loop since the basic induction variable changes by a constant value on each iteration. Constraint 3 in Figure 23(c) shows that the compiler determines that block 2 will jump to block 6 at most once (**2J once**). The analysis to detect iteration-based constraints requires $O(C)$ complexity, where C is the number of conditional branches, since each branch must be inspected once.

5.2 Using Constraints in a Timing Analyzer

The analysis techniques described in the previous section to identify branch constraints could be used by a variety of timing analyzers, which include those that use an integer linear programming (ILP) solver. While an ILP approach can be simple, elegant, and quite powerful, there are a few disadvantages. For instance, an ILP approach works best when each basic block can be associated with a single time, which allows this time to be expressed as a constraint associated with that block. Caching and pipelining change the context in which a block could be executed and can often affect its associated execution time. While approaches have been suggested for addressing caching behavior [9], it is still unclear how pipelining can be accurately modeled across multiple blocks. More importantly, the time required for the analysis with an ILP approach has worst-case exponential complexity. A program that required only a few seconds of timing analysis using a more traditional approach [26] required minutes using an ILP approach [9]. In fact, ILP methods can be used to solve many compiler optimization problems, but are infrequently used in production compilers due to potentially excessive compilation time. Finally, when a timing constraint is violated, a user would like to know where the time is being spent in the code associated with the constraints. The timing analysis approach described in this dissertation not only produces WCET and BCET predictions for an

entire program, but also gives the WCET and BCET for each function, loop and path in the program [27]. In contrast, an ILP approach only calculates a single WCET and BCET prediction for the entire program. Thus, the author decided it would be worthwhile to investigate how branch constraints could be exploited by a non-ILP based timing analyzer.

The remainder of this section will describe the details of how the timing analyzer makes use of the branch constraints to compute the WCET and BCET predictions for a particular loop or function. In particular, constraints on paths are generated from the branch constraints. For example, effect-based branch constraints can be used to determine if a given path is infeasible, or that one path cannot follow some other path on a subsequent iteration of the loop. Further constraints arise from analyzing which paths can execute on the first iteration. Iteration-based branch constraints are used to determine the range of iterations a particular path may be taken during the loop execution. Once the path constraints have been calculated, they are used in the worst-case and best-case loop analysis algorithms. The purpose of using these path constraints is to tighten the execution time predictions. For instance, if the timing analyzer can determine that the longest (shortest) path is infeasible or can only execute for a proper subset of the loop's iterations, then the WCET (BCET) bound will be tighter.

5.2.1 Overview for Generating Path Constraints

The timing analyzer uses branch constraints to calculate a minimum and maximum number of iterations associated with each path during the execution of a loop. Table 13 depicts worst-case iteration information associated with each loop path described in Figures 21(d), 22(d), and 23(d). Table 14 shows the analogous path iteration information

Table 13: Worst-Case Path Information for Figures 21(d), 22(d), and 23(d)

Loop	Total Iters	Path ID	Path Type	Possible Iterations	Unique Iters	Min Iters	Max Iters
Loop in Figure 21	1,001	1 2 3 4 5	exit exit cont cont cont	[1001..1001] [1001..1001] [1000..1000] [2..1000] [1..1000]	∅ ∅ ∅ ∅ ∅	0 0 0 0 0	1 1 1 500 500
Loop in Figure 22	1,000	1 2 3 4	both both both N/A	[1..1000] [1..1000] [1..1000] N/A	∅ ∅ ∅ N/A	0 0 0 N/A	1,000 1,000 1,000 N/A
Loop in Figure 23	1,000	1 2 3 4	both cont both cont	[1..1000] [1..250] [751..1000] [251..750]	∅ [1..250]–1 [751..1000]–1 [251..750]–1	0 249 249 499	1 250 250 500

for best case. The second and third example loops are not shown in Table 14 because their best case iteration information is identical to their worst case information from Table 13. The first loop example from Figure 21 does have a different number of iterations for worst case and best case, and this results in a different set of possible iterations and number of maximum iterations for each path. The total number of loop iterations is automatically calculated using techniques described in the previous section [31]. A path is a sequence of blocks in a loop connected by control-flow transitions.

Table 14: Best-Case Path Information for Figure 21(d)

Loop	Total Iters	Path Type	Path ID	Possible Iterations	Unique Iters	Min Iters	Max Iters
Loop in Figure 21	2	exit exit cont cont cont	1 2 3 4 5	[2..2] [2..2] [1..1] [2..2] [1..1]	∅ ∅ ∅ ∅ ∅	0 0 0 0 0	1 1 1 1 1

Each path starts with the loop header. *Exit* paths are terminated by a block with a transition out of the loop. *Continue* paths are terminated by a block with a transition to the loop header. The "Path Type" column shows that a path may be classified in one of four ways. *Exit* and *cont* represent that the path is an exit or continue path, respectively. The word *both* means the path is both an exit and a continue path, and *N/A* means that the path is infeasible. The "Possible Iterations" column indicates the range of possible iterations for each path. The "Unique Iters" column represents the unique iterations associated with each path. The final two columns show the minimum and maximum number of times the path could be executed in the loop.

Figure 24 gives a high-level description of the algorithm used to calculate the information given in the last five columns of Table 13 and 14. The next four sections 5.2.2 through 5.2.5 provide examples to illustrate how this information is calculated. Except for the construction of the REACH_SELF table, the complexity of the algorithm is $O(P^2)$, where P is the number of paths in the loop. The author found that, in practice, the construction of the REACH_SELF table was not time consuming since most paths in a loop could either immediately follow themselves or could only exit the loop.

5.2.2 Using Effect-Based Constraints

Effect-based constraints are associated with a block or a transition between blocks. For each path in a loop the timing analyzer traverses the basic blocks and transitions between blocks in the order in which the path would be executed. When an effect-based constraint is encountered, it is added to a list of constraints for that path. If another effect-based constraint is later encountered for that same branch, then the current constraint is nullified.

```
/* remove infeasible paths */
FOR each path P in the loop DO
   Propagate effect-based constraints in P.
   IF any transition in P is not feasible THEN
      Remove P from the loop.

/* calculate CAN_FOLLOW table using effect-based constraints */
FOR each path P in the loop DO
   IF P is a continue path THEN
      FOR each path Q in the loop DO
         Propagate effect-based constraints
            at end of P through Q.
         IF any infeasible transition in Q THEN
            CAN_FOLLOW[P][Q] = FALSE.
         ELSE
            CAN_FOLLOW[P][Q] = TRUE.
   ELSE
      FOR each path Q in the loop DO
         CAN_FOLLOW[P][Q] = FALSE.

/* calculate REACH_SELF table using CAN_FOLLOW table */
FOR each path P in the loop DO
   IF CANFOLLOW[P][P] THEN
      REACH_SELF[P] = 1.
   ELSIF P is not a continue path THEN
      REACH_SELF[P] = 0.
   ELSE
      Recursively inspect the CAN_FOLLOW table
      to determine the shortest number of paths
      to be traversed before P can be reached.
      Zero represents P cannot reach itself.

/* process once constraints */
FOR each path P in the loop DO
   IF a once constraint was found on
      a transition in P THEN
      P->once = TRUE.
   ELSE
      P->once = FALSE.
   P->nonuniqiters = 0.
   FOR each block B in P DO
      IF B's other outgoing transition has a
         once constraint THEN
         P->nonuniqiters += 1.

/* initialize possible iteration path information, where N
   represents the total loop iterations */
FOR each path P in the loop DO
   P->range = ∅.
   IF P is a continue path THEN
      P->range = P->range ∪ [1..max(N-1,1)].
   IF P is an exit path THEN
      P->range = P->range ∪ [N..N].
```

Figure 24: Algorithm for Calculating Path Iteration Information in Tables 13,14
(continued on next page)

```
/* constrain possible iterations using iteration-based constraints */
FOR each path P in the loop DO
   Propagate iteration-based constraints in P.
   P->range = P->range ∩
              iteration range at end of P.
   IF P->range = ∅ THEN
      Remove P from the loop.

/* constrain iterations of each path that cannot reach itself */
Construct a DAG D representing the execution
   order of paths P where REACH_SELF[P] == 0.
FOR each non-leaf path P in D, where P is not
    processed until all paths it can reach
    are processed DO
   S = first immediate successor of P.
   P->range.low = S->range.low - 1.
   P->range.high = S->range.high - 1.
   FOR each remaining path S that is an
       immediate successor of P in D DO
      IF S->range.low - 1 < P->range.low THEN
         P->range.low = S->range.low - 1.
      IF S->range.high - 1 > P->range.high THEN
         P->range.high = S->range.high - 1.

/* calculate unique iterations for each path */
FOR each path P in the loop DO
   P->uniqrange = P->range
   FOR each path Q, where Q ≠ P DO
      P->uniqrange = P->uniqrange – Q->range.

/* assign minimum number of iterations for each path */
FOR each path P in the loop DO
   P->miniter =
      number of iterations in P->uniqrange.
   P->miniter -= P->nonuniqiters.

/* assign maximum number of iterations for each path */
FOR each path P in the loop DO
   IF REACH_SELF[P] = 0 OR P->once THEN
      P->maxiter = 1.
   ELSE
      P->maxiter =
         number of iterations in P->range.
      IF REACH_SELF[P] > 1 THEN
         P->maxiter =
            ceil(P->maxiter/REACH_SELF[P]).

/* assign each path to a set of paths */
s = 0.
FOR each path P in the loop DO
   IF P->range ∩ with existing set i THEN
      P->set = i;
   ELSE
      P->set = ++s;
```

Figure 24: Algorithm for Calculating Path Iteration Information in Tables 13,14 (cont'd.)

Effect-based constraints can be used to detect infeasible paths. Figure 25 depicts the constraints being propagated through path 4 in Figure 22(d). The transition from block 2 to block 3 causes the branch in block 5 to be placed in a *jump* state (**5J**). The branch in block 5 is encountered with this constraint (**5J**) still in effect and the transition from block 5 to block 6 in path 4 is deemed illegal. When such an infeasible path is encountered, the timing analyzer removes the path to prevent any additional analysis time to be spent on it.

The maximum number of iterations for a path can sometimes be constrained by effect-based constraints. Consider paths 1 and 2 in Figure 21(d), which are *exit* paths because they end with a transition to block 10 that is outside the loop. Constraint 5 in Figure 21(c) indicates that when block 3 (`quit=1;`) in Figure 21(b) is executed, block 8 (`quit!=0`) will jump to block 10. When the timing analyzer detects that an effect-based constraint can reach the end of the path without nullification, the timing analyzer propagates the constraint through all the paths of the loop to see if it can reach the branch identified in the constraint. Figure 26 illustrates that the constraint causing the branch in block 8 to *jump* (**8J**) reaches the end of path 3 and that paths 2, 3, 4, and 5 cannot follow path 3 since they require a fall through from block 8 to block 9. Figure 27 shows that the constraint for branch 4 reaching the end of paths 4 and 5 from Figure 21 contains the opposite outcome of branch 4 in the same path. These constraints can reach

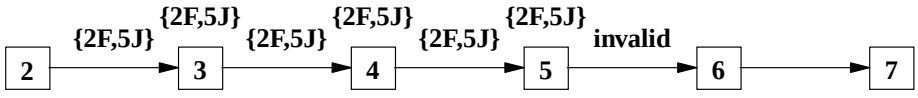


Figure 25: Path 4 in Figure 22(d) Is Not Feasible

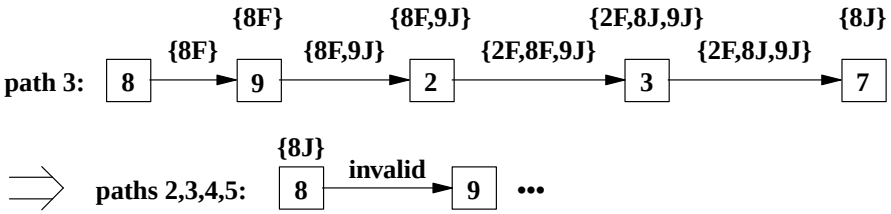


Figure 26: Paths 2-5 Cannot Follow Path 3 in Figure 21(d)

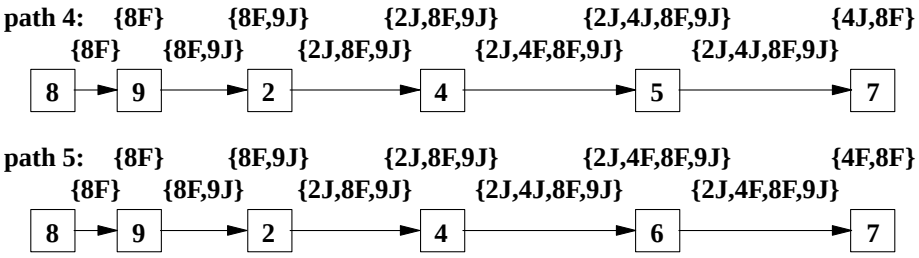


Figure 27: Paths 4 and 5 Cannot Immediately Follow the Same Path in Figure 21(d)

block 4 on the next iteration without being affected. This causes these paths not to follow themselves on the next loop iteration.

A *Can Follow* matrix is constructed by the timing analyzer that indicates for each path the set of paths that can legally follow it on the next iteration. If a constraint from one path can reach its associated branch in other paths without being nullified, then such paths that have transitions that do not satisfy the constraint are marked as illegal in the matrix. No paths are allowed to follow a path that only exits. Table 15 depicts the matrix of paths that can legally follow each path in Figure 21(d).

After the Can Follow matrix is completed, it is examined to see if restrictions on the number of iterations associated with each path can be applied. In general, the timing analyzer examines the matrix for each path to determine the fewest number of other

Table 15: Can Follow Matrix for Figure 21

Current Path in Loop	Paths That Can Immediately Follow				
	1	2	3	4	5
1	N	N	N	N	N
2	N	N	N	N	N
3	Y	N	N	N	N
4	N	Y	Y	N	Y
5	N	Y	Y	Y	N

paths required to be traversed before the current path can be executed again. If the algorithm indicates that a path cannot reach itself, then the path will be assigned a maximum of one iteration. Paths 1, 2, and 3 of Figure 21(d) are all assigned a maximum number of one iteration because they cannot reach themselves after executing. If a path cannot directly follow itself, but can eventually be reached again, then it cannot execute on every iteration of the loop. If the algorithm indicates that the *K* iterations required to be executed before a *continue* path can reach itself is greater than one, then it is assigned a maximum number of iterations from *ceil*(*R*/*K*), where *R* is the possible number of iterations for the path. Paths 4 and 5 of Figure 21(d) can only execute again on the second iteration after it last executed. Thus, paths 4 and 5 are assigned *ceil*(999/2) and *ceil*(1,000/2), respectively, or 500 maximum iterations.

5.2.3 Using Effect-Based Constraints On Entering a Loop

The previous section discussed how branch constraints are used to create path constraints within a loop. But there are further constraints that arise when the loop is entered that affect which paths can initially execute. The steps taken by the timing analyzer related to preheader constraints are as follows.

1. use data-flow analysis to determine the initial constraints
2. determine the first iteration on which each path in the loop can execute
3. update the range of possible iterations for the paths
4. update the minimum and maximum number of iterations of the loop

These steps are described in this section.

The timing analyzer uses data-flow analysis [39] to calculate *ins* and *outs* for each block in a function. The algorithm for accomplishing this is given in Figure 28. The data-flow equations 10 through 15 that determine the ins and outs are based on truth tables given in Table 16. The implementation uses six bit vectors for each block: in.jump, in.fallthru, in.unknown, out.jump, out.fallthru and out.unknown. The jump,

```
FOR each function in the program DO
   DO
      change = FALSE
      FOR each block in the function DO
         in.j = NULL
         in.f = NULL
         in.u = NULL
         IF the block has at least one predecessor (pred) THEN
            in.j = pred.out.j
            in.f = pred.out.f
            in.u = pred.out.u
            FOR each other predecessor block (pred) DO
               in.j ∪= pred.out.j
               in.f ∪= pred.out.f
               in.u ∪= pred.out.u ∪ (in.j ∩ pred.out.f) ∪
                  (in.f ∩ pred.out.j)

         Initialize this.e, this.u and this.j based on the branch
            constraints contained in this block.
         out.j = this.j ∪ (in.j - this.f - this.u)
         out.f = this.f ∪ (in.f - this.j - this.u)
         out.u = this.u ∪ (in.u - this.j - this.f)
         IF any in or out bit vector changed THEN
            change = TRUE
   WHILE change
```

Figure 28: Calculating Ins and Outs

$$B.in.j = \bigcap_{p \in preds(B)} p.out.j \tag{10}$$

$$B.in.f = \bigcap_{p \in preds(B)} p.out.f \tag{11}$$

$$B.in.u = (B.in.j \cup B.in.f)' \tag{12}$$

$$B.out.j = B.j \cup (B.in.j - B.f - B.u) \tag{13}$$

$$B.out.f = B.f \cup (B.in.f - B.j - B.u) \tag{14}$$

$$B.out.u = B.u \cup (B.in.u - B.j - B.f) \tag{15}$$

fallthru and unknown bit vectors indicate which branches are made to jump, fall through or become unknown, respectively, based on this block. For determining the ins and outs of a block, exactly one of the three corresponding bit vectors must be set, since a branch must be in either a jump, fall through or unknown state. Each block also contains bit vectors indicating if it causes a branch to jump, fall through or become unknown. However, the current block may have no effect on the branch in question, so it is possible that the bit vectors representing the effect from the current block may all be zero.

The first part of Table 16 enumerates the cases to calculate the ins, based on the outs of the predecessor blocks. As an illustration, consider line 7. Predecessor block p1 makes a branch unknown, and predecessor block p2 makes the same respective branch jump. The combination of these effects is to make that branch unknown. Since the

Table 16: Truth Tables for Ins and Outs

line	INPUT 1			INPUT 2			RESULT		
	p1.j	p1.f	p1.u	p2.j	p2.f	p2.u	in.j	in.f	in.u
1	1	0	0	1	0	0	1	0	0
2	1	0	0	0	1	0	0	0	1
3	1	0	0	0	0	1	0	0	1
4	0	1	0	1	0	0	0	0	1
5	0	1	0	0	1	0	0	1	0
6	0	1	0	0	0	1	0	0	1
7	0	0	1	1	0	0	0	0	1
8	0	0	1	0	1	0	0	0	1
9	0	0	1	0	0	1	0	0	1
	this.j	this.f	this.u	in.j	in.f	in.u	out.j	out.f	out.u
10	0	0	0	0	0	0	0	0	0
11	0	0	0	1	0	0	1	0	0
12	0	0	0	0	1	0	0	1	0
13	0	0	0	0	0	1	0	0	1
14	1	0	0	0	0	0	1	0	0
15	1	0	0	1	0	0	1	0	0
16	1	0	0	0	1	0	1	0	0
17	1	0	0	0	0	1	1	0	0
18	0	1	0	0	0	0	0	1	0
19	0	1	0	1	0	0	0	1	0
20	0	1	0	0	1	0	0	1	0
21	0	1	0	0	0	1	0	1	0
22	0	0	1	0	0	0	0	0	1
23	0	0	1	1	0	0	0	0	1
24	0	0	1	0	1	0	0	0	1
25	0	0	1	0	0	1	0	0	1

timing analyzer cannot assume which predecessor block will always precede the block in question, it has to intersect the information from all the predecessors. The inner FOR-loop in Figure 28 combines the effects from each predecessor block one at a time.

Equations 10 and 11 show that the current block's ins for the jump (fall through) branches are simply the intersection of the jump (fall through) bit vectors of the predecessors' outs. Equation 12 states that the ins for the unknown branches are the complement of the union of the ins for the jump and fall through branches. For example, if one predecessor out says that a certain branch will fall through, but another predecessor out says the same branch will jump, then the in of the current block will show that that branch is unknown due to the conflict between the predecessors.

The second part of Table 16 shows the cases that determine the outs, based on the effects of the block in question combined with its ins. If the current block has no effect on a branch, then the out bit vectors will be assigned the value of the ins. Otherwise, the effect of this block will override the ins to determine the outs of this block. For example, consider line 16 in Table 16. It depicts a situation where the block in question makes a particular branch jump, while the effect of the ins is to make that jump fall through. In this case, since this block has an effect, it overrides the ins, so the value of the out bit vectors will represent that the branch will jump. The equations 13 through 15 to compute the outs are straightforward and follow directly from the truth table. In Figure 28, the outs are calculated after the ins. However, the algorithm is a typical data-flow calculation in which the ins and outs depend on each other, so the algorithm continues until there is no change to the bit vectors.

After the ins and outs of every block are calculated, the timing analyzer uses the outs of the preheader to see which paths can execute on the first iteration. The algorithm in

Figure 29 sets `p.on_first` to true (false) if it determines path `p` can (cannot) execute on the first iteration. The cases in which `p.on_first` is false correspond to situations where a branch in the path contradicts the information from the preheader outs. If a path is found not able to execute on the first iteration as a result of this algorithm, then in some cases it may be assigned fewer maximum iterations, and a more accurate timing bound can be obtained. The preheader's *out* constraints are propagated through each path. Any path that does not obey the preheader constraints cannot execute on the first iteration. For example, consider the loop in Figure 21. The application of the algorithm in Figure 29 to the paths of this loop is depicted in Figure 30. This figure shows the propagation of the preheader constraints to determine which paths can execute on the first iteration. The solid arrows indicate transitions that occur between blocks inside the loop, while dashed arrows indicate transitions to or from a block outside the loop. Block 1 is the preheader of the loop, and block 10 is the block to which the loop exits. The value of `odd` is initialized to 0 in block 1, which is in the outs of the preheader of the loop, so the associated branch constraint is **{4J}**. Thus, on the first iteration of the loop, the branch in block 4 must be taken. Path 4 contains a transition from block 4 to block 5, which is a fall through situation, contradicting the preheader constraint. The timing analyzer detects that path 4 cannot execute on the first iteration.

The algorithm in Figure 29 also detects if a loop exit transition in a path causes it to be ineligible to execute on the first iteration. Consider exit paths 1 and 2 from the loop in Figure 21. Path 1 consists only of block 8, so this block is considered the last block in

```
Initialize pre.j, pre.f and pre.u to be the union of the
   respective bit vectors of all the header's immediate predecessors.
FOR each path (p) in the loop DO
   IF we already know the path cannot execute on
      first iteration THEN
      CONTINUE
   FOR each block (b) in path p DO
      p.on_first = TRUE
      IF there is no branch in this block THEN
         CONTINUE
      IF all three bit vectors at bit b are zero THEN
         CONTINUE
      succ = number of immediate successor block that lies outside
         the loop
      /* if the preheader says this branch must jump */
      pre.j[b] THEN
         IF this is not the last block in path THEN
            IF number of next block in path == b + 1
               p.on_first = FALSE
         ELSE
            IF succ == b + 1 THEN
               p.on_first = FALSE
      /* if the preheader says this branch must fall through */
      ELSIF pre.f[b] THEN
         IF this is not last block in path THEN
            IF number of next block in path != b + 1
               p.on_first = FALSE
         ELSE
            IF succ != b + 1 THEN
               p.on_first = FALSE
```

Figure 29: Which Paths Can Execute on First Iteration

the path. The timing analyzer determines the successor block to block 8 that is located outside the loop, which is block 10. The exit transition from block 8 to block 10 is a jump, however the preheader constraint is for the branch in block 8 to fall through (see **8F** constraint shown for path 1 in Figure 30). This contradiction means that path 1 cannot execute on the first iteration. Path 2 has a similar situation. Its last block is block 9, and its successor that is located outside the loop is block 10. To exit the loop by taking path 2 implies that the branch in block 9 must fall through, but the preheader constraint says that it must jump (see **9J** constraint shown for path 2 in Figure 30). So

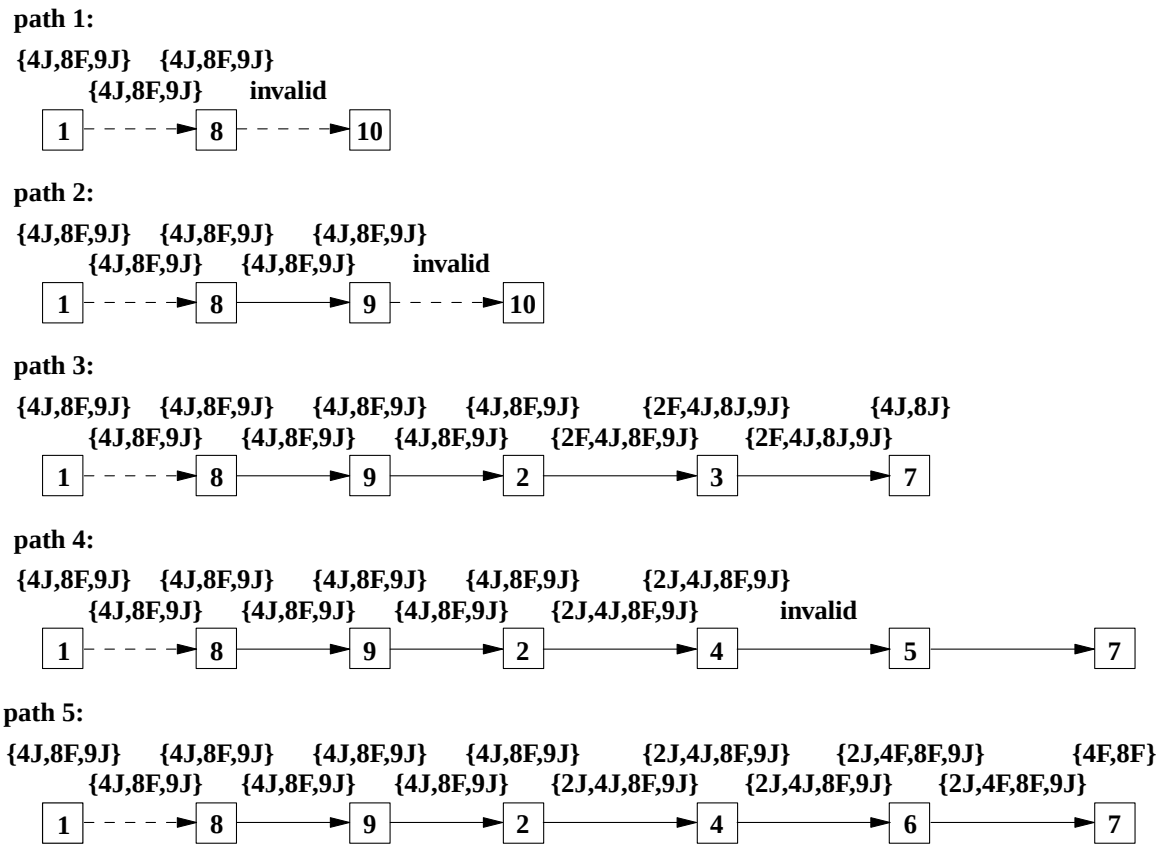


Figure 30: Propagating Preheader Constraints for Figure 21

the timing analyzer concludes that path 2 cannot execute on the first iteration as well.

For those paths that cannot execute on the first iteration, the next step is to determine on which iteration it can first be taken. Table 17 shows a Path Distance matrix for the example loop in Figure 21 that is derived from the Can Follow matrix given in Table 15. The table entries containing ∞ indicate that it is impossible for one path to reach the other path. For paths that cannot execute on the first iteration, the timing analyzer determines on which iteration it can execute as follows. Let *P* be the set of paths that

Table 17: Path Distance Matrix for Figure 21

Current Path in Loop	How Many Iterations to Reach Path				
	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	∞	∞	∞	∞	∞
3	1	∞	∞	∞	∞
4	2	1	1	2	1
5	2	1	1	1	2

can execute on the first iteration, and let *Q* be the set of paths that cannot. For each path *q* in *Q*, the timing analyzer finds the shortest number of iterations to reach *q* from any path in *P*. This shortest distance plus 1 represents the first iteration on which path *q* can execute. Continuing with the example from Figure 21, path 4 belongs to the set *Q*. Path 5 is a path in *P*, and according to Table 17 the path distance from path 5 to path 4 is one iteration. So the timing analyzer concludes that path 4 can first execute on the second iteration, and the range of possible iterations becomes [2..1000]. Similarly, the timing analyzer determined that exit paths 1 and 2 could not execute on the first iteration. However, the path distances from path 3 to path 1 and from path 5 to path 2 are both one iteration as indicated in Table 17. Since both path 3 and path 5 can execute on the first iteration, paths 1 and 2 can first execute on the second iteration of the loop. For best case analysis, their ranges of possible iterations are adjusted to [2..2] as shown in Table 14. Their worst-case possible iterations are not updated since they had already been determined to be [1001..1001] in Table 13.

The timing analyzer enforces a rule that if any exit path can execute on the first

iteration, then it must allow all exit paths to be chosen for the first iteration. The reason for this rule is that in best case, the BCET is assumed to occur for the minimum number of iterations. Consider a loop having two paths, where only path 1 can execute on the first iteration, but path 2 is significantly shorter. Then the loop may take less time to execute path 2 for two iterations than to execute path 1 for just one iteration. The author believes that requiring the best-case loop analysis algorithm to repeatedly examine a loop for varying numbers of iterations would be overly inefficient. Specifying the minimum number of iterations before starting loop analysis makes the algorithm much simpler and only slightly more conservative in this highly unlikely scenario. In the above scenario, the timing analyzer will make the conservative assumption that path 2 can execute on the first iteration, and that the minimum number of iterations is still one.

If it turns out that no exit path can execute on the first iteration, then the timing analyzer updates the number of iterations of the loop based on when the exit paths can execute. In the example from Figure 21, both exit paths can only execute on the second iteration, so the timing analyzer sets the minimum number of iterations to 2, even though the compiler had previously determined, before this path analysis was performed, that the minimum number of iterations would have been 1 [31].

The total number of iterations of the loop may also be updated in the case where the user is prompted to enter information from which the timing analyzer computes the number of iterations. If the user provides unrealistic values, then the number of iterations based on the user's information may be too small, and updating the number of

iterations would be appropriate. But the situation of having to updating the number of iterations is quite rare, only occurring when the number of paths exceeds the original number of loop iterations.

5.2.4 Using Iteration-Based Constraints

The maximum number of iterations can sometimes be constrained by analyzing iteration-based constraints. The header block is assigned a range that spans all iterations of the loop. This range is propagated through each path. When a transition is encountered that has an iteration-based constraint, the range in the constraint is intersected with the range in the current block in the path. Figure 31 illustrates how iteration-based constraints are propagated through path 4 in Figure 23(d). The transition from block 3 (`i<=249`) to block 4 results in the range [1..1000] being intersected with [251..1000], which is the range specified in constraint 5 of Figure 23(c). The transition from block 4 (`i>=750`) to block 5 results in the current range of [251..1000] being intersected with [1..750]. Thus, path 4 can only possibly execute in iterations [251..750].

If a path can only be executed in a given range of iterations, then the maximum iterations in which that path can execute cannot be greater than the number of iterations

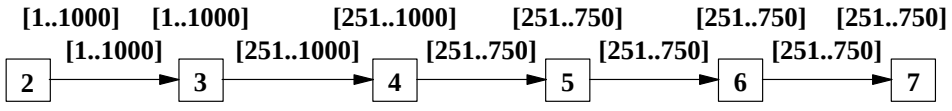


Figure 31: Iteration-Based Constraints Propagated Through Path 4 in Figure 23

in the range. A path with no possible iterations is infeasible and is removed from the list of paths by the timing analyzer. Note that the range of a path that only exits is always the last iteration of the loop, which is the case for paths 1 and 2 of Figure 21(d). Likewise, if path A cannot reach itself and can only be immediately followed by a different path B, which has a range [Bmin..Bmax], then path A's range cannot span more than [Bmin−1..Bmax−1]. For instance, Table 15 shows that path 3 of Figure 21(d) always leads to path 1, which has an iteration range of [1001..1001]. Thus, path 3's possible range of iterations is [1001−1..1001−1] or [1000..1000] for WCET analysis.

The minimum number of iterations of a path is calculated by simply subtracting the possible range of iterations of all other paths in the loop from the possible range of iterations for the current path. The result is the unique set of iterations for the current path, which is the minimum number of times that the path has to execute. There is one exception to this rule. Consider path 1 in Figure 23(d). Its maximum number of iterations is one due to constraint 3 (**2J once**) in Figure 23(c). The timing analyzer does not reduce the range of unique iterations of the other paths, but does indicate that one iteration in these paths may not be unique.

5.2.5 Using the Constraints in Loop Analysis

The author decided to use the minimum and maximum iterations associated with each loop path to obtain tighter loop predictions without restricting the order in which these paths are evaluated. There were several reasons why this approach was used. First, the approach supports paths that can execute at most once, but in any iteration. Consider

path 1 of the loop in Figure 23. This situation may occur frequently in numerical applications. For instance, special conditions are often checked for the diagonal elements of a matrix (diagonal systems). Second, the approach deals with paths that have dependencies on other paths, such as paths 4 and 5 in Figure 21. Finally, the timing analyzer often calculates an average WCET and BCET for a loop using an average number of iterations when the number of iterations can vary depending on the value of a outer loop counter variable [31]. Using this approach allows the calculation of a safe average WCET (BCET) since the longest (shortest) paths are selected first in the respective loop analysis algorithms.

In addition, the timing analyzer determines sets of paths, where the range of iterations of the paths in one set do not overlap with other sets. Each path is assigned to a single set of paths. The timing analyzer uses the maximum number of iterations that can be executed by a set of paths, which is the number of iterations in the set's range. Table 18 depicts an example with 4 paths and 2 sets. Each set of paths can only execute a maximum of 50 iterations. If only the maximum iterations of each path was used, then two paths from a single set could be selected and a significant overestimation may occur

Table 18: Example Illustrating Use of Path Sets

Path	Possible Iterations	Min Iters	Max Iters	Set
1	[1..50]	0	50	1
2	[1..50]	0	50	1
3	[51..100]	0	50	2
4	[51..100]	0	50	2

when the paths in one set require many more cycles than the paths in the other set. This approach has limitations. Consider if a fifth path existed in this example which could execute in any iteration of the loop. All of the loop paths would be assigned to a single set, which could result in an conservative timing prediction. Fortunately, inequality tests (<, <=, >=, >) on loop induction variables do not occur frequently. The two subsections that follow describe the worst-case and best-case loop analysis algorithms that employ the path constraint information.

5.2.6 Worst Case Loop Analysis

Figure 32 shows how the WCET loop analysis algorithm uses the path constraint information. Let N be the maximum number of iterations and P be the number of paths in a loop. The DO-WHILE will process at most the minimum of N or $2P$ total iterations since the first misses and first hits in each path can miss or hit at most once, respectively.[6]

The algorithm selects the longest path on each iteration of the loop from the set of paths that can still possibly execute. In order to demonstrate the correctness of the algorithm, one must show that no other path for a given iteration of the loop will produce a longer worst-case time than that path selected by the algorithm. Descriptions of how the caching categorizations and pipeline information are used in the loop analysis and correctness arguments about selecting the longest path using these categorizations and information have been given in previous work [25, 26]. Thus, it remains to be shown

[6] If the number of paths within a loop exceeds a reasonable limit, then the loop control flow is partitioned to reduce the timing analysis complexity [30].

```
/* calculate required and non-required path information */
req_iters = 0.
FOR P = each path in the loop DO
   P->req_iters = P->min_iters.
   P->nonreq_iters = P->max_iters - P->min_iters.
   req_iters += P->min_iters.
nonreq_iters = N - req_iters.

/* process all iterations of the loop */
iters_handled = 0.
pipeline_info = NULL.
WHILE iters_handled < N DO

   /* process iters while longest path has a first miss or first hit */
   DO
      IF req_iters < N - iters_handled THEN
         Find longest path P where
            P->req_iters+P->nonreq_iters > 0 &&
            P->set.maxiters > 0.
      ELSE
         Find longest path P where
            P->req_iters > 0 &&
            P->set.maxiters > 0.
      Concatenate pipeline_info with the current
         worst-case union of executable paths.
      iters_handled += 1.
      IF P->req_iters > 0 THEN
         P->req_iters -= 1.
         req_iters -= 1.
      ELSE
         P->nonreq_iters -= 1.
         nonreq_iters -= 1.
      P->set.maxiters -= 1.
   WHILE encountered a first miss or first hit
         AND iters_handled < N

   /* Efficiently process iterations for the current longest path */
   IF iters_handled < N THEN
      nonreq_iters_to_do =
         min(nonreq_iters, P->nonreq_iters,
             P->set.maxiters - P->req_iters).
      iters_to_do = P->req_iters + nonreq_iters_to_do.
      req_iters -= P->req_iters.
      nonreq_iters -= nonreq_iters_to_do.
      P->set.maxiters -= iters_to_do.
      P->req_iters = 0.
      P->nonreq_iters -= nonreq_iters_to_do.
      Concatenate pipeline_info iters_to_do
         times with current worst-case union.
      iters_handled += iters_to_do.
```

Figure 32: WCET Loop Analysis Algorithm

that each time a path is selected, it is in fact chosen from the paths that can still possibly execute given that the minimum and maximum number of iterations for each path and

set were accurately estimated. A path's number of required iterations is its minimum iterations to be performed. The non-required iterations of a path is the difference between its maximum and minimum number of iterations. A path is initially chosen in the IF-THEN-ELSE construct at the beginning of the DO-WHILE loop in Figure 32. If the iterations remaining is greater than the required iterations left to be processed (sum of each path's minimum iterations not yet processed), then the path selected is chosen from any path that has any iterations that can be performed. Otherwise, the iterations remaining must be equal to the required loop iterations remaining and the path must be selected only from paths that have remaining required iterations left. The code after the DO-WHILE in the algorithm efficiently uses repeated instances of a path that has no first misses or first hits and thus will remain the longest path since its worst-case behavior cannot change. This code processes the remaining required iterations of the path and the minimum of the remaining non-required iterations of the path, the set of paths to which the path belongs, or the entire loop. Therefore, the paths that can still possibly execute is accurate since a given path's required iterations are always processed before its non-required iterations and the number of non-required iterations to be processed for a path is never allowed to exceed the number of non-required iterations remaining in the loop.

Table 19 illustrates the worst-case loop analysis algorithm using the example loop given in Figure 21. The iteration information pertaining to the five paths was given in Table 13. None of the paths has any required iterations, so the number of non-required iterations is the same as their number of maximum iterations available. Table 19 outlines

Table 19: Example for Worst-Case Loop Analysis

Iteration	P 1	P 2	P 3	P 4	P 5	Longest	Time
1	16	28	44	**56**	54	4	56
2	7	10	17	**20**	18	4	72
3-500	*7*	*10*	*17*	**20**	*18*	4	8040
501	7	10	17		**18**	5	8054
502-1000	*7*	*10*	*17*		**18**	5	15040
1001	7	**10**				2	15046

the progress of the loop analysis algorithm. It selects the longest path for each iteration. Note that the iteration numbers in the first column are accounting for the iterations of the loop, but not necessarily in the order in which they take place. To actually identify which path is the longest on each sequential iteration would make the loop analysis algorithm more complex, with little or no benefit in tightening the execution time bound. However, the use of path sets is used to determine whether a particular path is eligible to execute during a particular range of iterations, and this feature will also be illustrated shortly. For the remainder of this illustration of the algorithm, and also for the subsequent example for best case, the iterations will be referred to ordinally, but the reader should note that these iteration numbers are only used for accounting all the loop iterations and does imply the temporal order of paths actually taken.

Columns 2 through 6 in the table indicate the path execution times for a particular iteration. All five paths are eligible to execute for the first iteration, and path 4 is the longest, taking 56 cycles. It turns out in this example that all the first misses encountered during the first iteration, so that the instruction cache behavior does not change starting

with the second iteration. For the second iteration, all first misses are now treated as hits. Once again path 4 is the longest path. Its execution time is 20 cycles, but starting with the second iteration it is no longer necessary to fill the pipeline, so 16 (20−4) cycles are added to the total time for the loop after two iterations.

Since there is no change in the instruction cache behavior during the second iteration, the algorithm proceeds to the second phase where it efficiently replicates path 4 until it has exhausted its 500 available iterations. For iteration 501 the algorithm returns to the first phase, where the individual paths are re-evaluated. At this point path 4 is ineligible for consideration since it has exhausted its iterations. The longest path available is path 5, whose execution time is 18 cycles, which includes the pipeline filling time that is not included in the accumulation of the total time for the loop. So the algorithm adds 14 cycles to the loop's execution time. Since there is no change to the cache behavior, path 5 is efficiently replicated starting at iteration 502, for its remaining 499 iterations.

For iteration 1001, which is the last iteration of the loop, the algorithm cannot consider paths 4 or 5 because they have exhausted their number of iterations. Path 3 cannot be considered either, since the previous analysis determined that it is in the same set of paths as paths 4 and 5 that share iterations [1..1000], and this set has exhausted all 1000 of its maximum iterations. So only paths 1 and 2 are in contention for the final iteration and path 2 is the longer path. The total execution time for the loop is predicted to be 15,046 cycles, which is exact. If the timing analyzer did not generate path constraints, then path 4 would have been selected for 1000 iterations rather than 500, and

the loop's WCET would have been overestimated by about 7%.

5.2.7 Best Case Loop Analysis

Figure 33 depicts the best-case loop analysis algorithm, which is for the most part analogous to the worst-case algorithm described in the previous subsection. As a preliminary step, the algorithm computes the number of required and non-required iterations for each path, as was done in worst case. The rest of the algorithm consists of two phases. The first phase finds the shortest path *P* for the first iteration. For the first iteration only, the timing analyzer treats all first misses as misses and all first hits as hits when analyzing the cache behavior of all the paths' instructions. The major issue for selecting the shortest path *P* is determining which paths are eligible to be selected. If the loop has at least one non-required iteration, then *P* may be chosen from any of the continue paths. However, if the loop has no non-required iterations, then *P* may only be selected from those continue paths that have required iterations.

The `WHILE-DO` loop in Figure 33 represents the second phase of the best-case algorithm, which processes all the remaining iterations of the loop after the first. Note that the timing analyzer treats a function as a loop with a single iteration, so its best case analysis will only perform the first phase of this algorithm. In the second phase, all first misses are treated as hits and all first hits are treated as misses. In other words, the instruction cache behavior is assumed not to change during the last $n - 1$ iterations. The reason for the difference in how the worst-case and best-case loop analysis algorithms are organized is described in the next section. The method of selecting the shortest path

```
/* calculate required and non-required path information */
req_iters = 0.
FOR P = each path in the loop DO
   P->req_iters = P->min_iters.
   P->nonreq_iters = P->max_iters - P->min_iters.
   req_iters += P->min_iters.
nonreq_iters = N - req_iters.
pipeline_info = NULL.

/* process the first iteration of the loop */
first_miss_treatment = miss.
first_hit_treatment = hit.
IF req_iters < N THEN
   Find shortest path P among the paths in which
      P->req_iters + P->nonreq_iters > 0  &&  P->set.maxiters > 0.
ELSE
   Find shortest path P among the paths in which
      P->req_iters > 0  &&  P->set.maxiters > 0.
Concatenate pipeline_info with the current
   best-case union of executable paths.
iters_handled = 1.
IF P->req_iters > 0 THEN
   P->req_iters -= 1.
   req_iters -= 1.
ELSE
   P->nonreq_iters -= 1.
   nonreq_iters -= 1.
P->set.maxiters -= 1.

/* process the remaining iterations */
WHILE iters_handled < N DO
   first_miss_treatment = hit.
   first_hit_treatment = miss.
   IF req_iters < N THEN
      Find shortest path P among the paths in which
         P->req_iters + P->nonreq_iters > 0  &&  P->set.maxiters > 0.
   ELSE
      Find shortest path P among the paths in which
         P->req_iters > 0  &&  P->set.maxiters > 0.
   nonreq_iters_to_do = min (nonreq_iters, P->nonreq_iters,
                             P->set.maxiters - P->req_iters).
   iters_to_do = P->req_iters + nonreq_iters_to_do.
   req_iters -= P->req_iters.
   nonreq_iters -= nonreq_iters_to_do.
   P->req_iters = 0.
   P->set.max_iters -= iters_to_do.
   P->nonreq_iters -= nonreq_iters_to_do.
   Concatenate pipeline_info with the current
      best-case union of executable paths.
   iters_handled += iters_to_do.
```

Figure 33: BCET Loop Analysis Algorithm

P is the same as in the first phase. Once *P* is selected, it is necessary to calculate the number of iterations to account for path *P*, which is done in the same manner as in worst

case. The timing analyzer will use *P* for all of its required iterations, plus the minimum of *P*'s non-required iterations, *P*'s set's maximum iterations remaining and the remaining non-required iterations of the loop. Since the method of selecting the shortest path for the best-case algorithm is analogous to selecting the longest path in the worst-case algorithm, the correctness argument for best case would also follow analogously from the worst-case explanation given in the previous section.

Table 20 illustrates the best-case loop analysis algorithm, using the same example loop (from Figure 21) that was described for worst case. This example's best case iteration information was given in Table 14. This loop only has two iterations instead of 1001 for worst case. The first phase of the algorithm selects the shortest path for the first iteration. The only paths that are eligible for iteration 1 are paths 3 and 5, and path 3 is shorter. The second phase of the algorithm examines the remaining iterations in the loop, and in this case there is only iteration 2 to consider. For this iteration, only paths 1 and 2 are eligible. Path 4 is a continue path and its range of possible iterations is [2..2], but this iteration is the last iteration of the loop. Path 4 cannot serve as a continue path, and it is not an exit path either, so path 4 is not eligible to be chosen for the second iteration. Among paths 1 and 2 for the second iteration, path 1 is the shorter path. Its

Table 20: Example for Best-Case Loop Analysis

Iteration	P 1	P 2	P 3	P 4	P 5	Shortest	Time
1			**44**		54	3	44
2	7	10				1	47

execution time of 7 cycles includes 4 cycles for pipeline filling, which is not used after the first iteration. The best-case execution time for the loop is computed to be 47 cycles, which is an exact prediction.

5.2.8 Reason for Different Algorithms

It is important to note that the worst-case and best-case loop analysis algorithms are not perfectly analogous with respect to the effect of first misses [2]. Consider a loop having three paths with information depicted in Table 21. Paths 1 and 2 each have a distinct first miss instruction, while path 3 has no first misses. According to the worst-case loop analysis algorithm, the timing analyzer selects path 1 for the first iteration, path 2 for the second iteration, and path 3 for all other iterations. For this example, the worst-case algorithm computes the WCET exactly for any number of loop iterations.

For best case, path 3 will be chosen for the first iteration. But starting with the second iteration, all first misses will be treated as hits, so path 2 will be selected for all iterations after the first. Thus, the timing analyzer will compute a BCET of 13 + 9*(n – 1) cycles for this loop, where n is the minimum number of loop iterations. However, the true BCET of this loop can be slightly greater. If the loop has just one iteration, the

Table 21: Information on Three Paths in Hypothetical Loop

How Path is Evaluated	Path 1	Path 2	Path 3
Treat first misses as misses	19	18	13
Treat first misses as hits	10	9	13

timing analyzer correctly predicts that path 3 should be taken, and there is no underestimation in the BCET. If the loop has two iterations, then path 3 should be taken for both iterations, yielding 26 cycles for the loop. The timing analyzer would compute 22 cycles if there are two iterations, a BCET underestimation of four cycles. On the other hand, if there are three or more iterations, the BCET is realized if the loop takes path 2 for every iteration. In this case, the timing analyzer will underestimate the BCET of the loop by five cycles, and this underestimation is due to the incorrect prediction of which path had been chosen for the first iteration. In order to make an exact prediction in best case, it becomes necessary to re-examine path choices for prior iterations. The author believes that having to re-examine all combinations of path choices for prior iterations to compute the BCET of a current iteration is overly inefficient, and thus the slightly more conservative approach described in Figure 33 is used.

5.3 Results

The results of the test programs are shown in Table 22. This table shows the benefit of automatically addressing branch constraints within the timing analyzer. The lines that are printed in boldface indicate which timing predictions became tighter as a result of this additional analysis. As in previous result tables, the *Observed Cycles* represent the cycles required for an execution with worst-case and best-case input data, as appropriate.[7] The last two columns indicate the results when the analysis includes the

[7] The author modified the desired relative error of the *Expint* and *Gaujac* programs so they would not converge early in worst case, which made it possible to obtain an accurate maximum iterations for a loop and worst-case input data for the *Observed Cycles* in Table 22.

Table 22: Results After Adding Branch Constraint Analysis

Worst-Case Results					
Name	Observed Cycles	+ Iter. Count Cycles	+ Iter. Count Ratio	+ Br. Constr. Cycles	+ Br. Constr. Ratio
Des	**149,706**	**172,509**	**1.152**	**167,165**	**1.117**
Expint	**58,217**	**1,293,290**	**22.215**	**58,289**	**1.001**
Fresnel	**47,749**	**48,887**	**1.024**	**47,783**	**1.001**
Gaujac	**786,786**	**790,116**	**1.004**	**787,134**	**1.000**
Hes	55,834,609	56,739,136	1.016	56,739,136	1.016
Integ	22,538,082	22,553,163	1.001	22,553,163	1.001
Interp	25,469,403	25,478,409	1.000	25,478,409	1.000
LU	**23,055,832**	**23,572,337**	**1.022**	**23,444,562**	**1.017**
Matcnt	1,769,321	1,861,150	1.052	1,861,150	1.052
Matmul	4,444,911	4,448,212	1.001	4,448,212	1.001
Matsum	1,277,465	1,279,322	1.001	1,279,322	1.001
Sort	7,672,281	7,672,292	1.000	7,672,292	1.000
Sprsin	**28,339**	**28,664**	**1.011**	**28,404**	**1.002**
Stats	1,016,048	1,016,128	1.000	1,016,128	1.000
Summidall	**15,340**	**18,090**	**1.179**	**15,341**	**1.000**
Summinmax	**16,080**	**17,080**	**1.062**	**16,080**	**1.000**
Sumnegpos	**11,067**	**13,068**	**1.181**	**11,068**	**1.000**
Sumoddeven	**15,093**	**16,112**	**1.068**	**15,102**	**1.001**
Sym	2,747,654	2,747,708	1.000	2,747,708	1.000
Average	7,734,420	7,882,403	2.157	7,815,352	1.011

Best-Case Results					
Name	Observed Cycles	+ Iter. Count Cycles	+ Iter. Count Ratio	+ Br. Constr. Cycles	+ Br. Constr. Ratio
Des	**65,615**	**22,247**	**0.339**	**57,920**	**0.883**
Expint	125	118	0.944	118	0.944
Fresnel	181	172	0.950	172	0.950
Gaujac	**45,270**	**44,566**	**0.984**	**45,127**	**0.997**
Hes	306,733	258,908	0.844	258,908	0.844
Integ	19,160,842	19,135,118	0.999	19,135,118	0.999
Interp	6,485,878	6,479,865	0.999	6,479,865	0.999
LU	**12,883,939**	**637,365**	**0.049**	**11,847,472**	**0.920**
Matcnt	1,549,095	1,548,798	1.000	1,548,798	1.000
Matmul	4,444,666	4,420,068	0.994	4,420,068	0.994
Matsum	1,257,239	1,167,140	0.923	1,167,140	0.923
Sort	19,966	19,950	0.999	19,950	0.999
Sprsin	17,436	17,379	0.997	17,379	0.997
Stats	607,399	601,406	0.990	601,406	0.990
Summidall	**15,340**	**8,072**	**0.526**	**15,312**	**0.998**
Summinmax	13,080	13,062	0.999	13,062	0.999
Sumnegpos	9,067	9,049	0.998	9,049	0.998
Sumoddeven	**94**	**63**	**0.670**	**94**	**1.000**
Sym	160	160	1.000	160	1.000
Average	2,467,480	1,809,658	0.853	2,395,748	0.970

automatic detection and exploitation of branch constraints.

Several of the test programs exhibit branch constraints that have been described in this chapter. The *Sumoddeven*, *Sumnegpos*, and *Summidall* programs correspond to the examples illustrated in Figures 21, 22, and 23, respectively. The *Des* program contains a loop in which the index variable is being compared to constants, giving rise to iteration-based constraints. The *Expint* program performs more computation when a loop variable is equal to a loop-invariant value on a single loop iteration. *Fresnel* takes different paths on the odd and even steps in the evaluation of the series. *Gaujac* executes different paths depending upon the specified iteration of a loop. The *LU* program contains some nested loops in which the the body of the inner loop may or may not be entered based on a condition in the outer loop. The *Sprsin* program does not perform a computation for a single column (the diagonal element) of each row of a matrix. The *Summinmax* program determines the minimum and maximum of each corresponding pair of elements in two vectors and these two tests are logically correlated.

The results show that exploiting value-dependent constraint information in a timing analyzer can significantly tighten WCET and BCET predictions. The programs *Fresnel* and *Sumoddeven* execute alternating paths in a loop depending upon a flag variable. One of the alternating paths has a slightly longer WCET than the other path in both of these programs. The timing analyzer was able to determine that longer path of each program could only be executed for one half of the iterations, which reduced the overestimations. In the case of *Sumoddeven* in best case, the compiler originally determined that the loop

had a minimum number of iterations of 1, but the timing analyzer was able to predict that the loop was required to iterate twice, using the methods described in Section 5.2.3. The result of this analysis was an exact BCET prediction. *LU* also showed a dramatic tightening in its BCET prediction. There were three nested loops in which the timing analyzer was able to exploit iteration-based constraints. The previous version of the timing analyzer assumed that the inner loop in these three nests would always be avoided along the best-case path of their respective surrounding loops. But in fact these loops execute on all but one iteration of the surrounding loops. The *Summinmax* and *Sumnegpos* programs have logically correlated branches and the timing analyzer was able to detect for each program that the longest path was infeasible due to this correlation. The compiler detected iteration-based constraints for the *Gaujac* and *Summidall* programs indicating that certain paths could only be executed in specific iterations. There was little overestimation in the previous version of the timing analyzer for *Gaujac* since these iteration-based constraints were associated with paths that were not in the most deeply nested loop of the program. However, *Summidall*'s iteration-based constraints were for the most frequently executed portion of that program and a significant overestimation of WCET was avoided. In best case, the timing analyzer was able to determine that the loop's shortest path in *Summidall* could execute at most once, and its second shortest path could execute for at most 250 of the 1,000 iterations. Even the longest path was required to execute for at least 499 iterations. These iteration-based constraints significantly tightened *Summidall*'s BCET prediction. Finally, the compiler

detected an iteration-based constraint in *Sprsin* and *Expint* that was associated with an equality test between a loop variable and a value that was invariant for that loop. This means that the loop could only execute a path associated with the equality transition from the block containing the test for a single iteration of the loop. For *Sprsin* this path required a smaller WCET than when the loop variable was not equal to the loop-invariant value. Thus, the overestimation by the previous version of the analyzer was quite small and would decrease when applied to arrays with larger dimensions. However, the opposite situation occurs in *Expint*, which has a higher WCET associated with the path where the loop variable is equal to the loop-invariant value. Thus, exploiting this branch constraint significantly reduces the WCET overestimation of *Expint*.

The slight remaining WCET overestimations and BCET underestimations for several of the programs in the current version of the timing analyzer were due to a few reasons. First, the *Des* program in particular had several arrays in which the elements are hard-coded in the data segment, and these array element values affect various comparisons. These branch constraints were not detected in the compiler. Second, in worst case some instructions conservatively categorized as misses actually hit in cache due to the order in which paths were executed because of dependences on data values. *Matcnt* had about a 90,000 cycle (about 5%) overestimation in worst case due to this conservative categorization. Similarly, in best case some instructions were conservatively classified as hits even though they actually miss in cache. *Hes* had a 44,541 cycle (about 15%)

underestimation in best case for this reason. Third, there were some minor limitations to the timing analysis that result in conservative predictions. For instance, the programs *Hes*, *Integ* and *LU* had non-rectangular loop nests where the number of iterations is rounded to an integer, and this effect was described in the previous chapter. Also, the underestimation in *LU* was partially due to the fact that an iteration-based constraint was not generated by the compiler for a condition containing a complex expression that needed to be expanded. Finally, there were slightly conservative predictions that resulted from instruction caching categorizations that change between loop levels and their interaction with the pipeline analysis, affecting both WCET and BCET [26]. In particular, the 8% underestimation of the BCET of *Summatrix* was due to this interaction.

5.4 Conclusions

This chapter has described how branch constraints were automatically detected by a compiler and exploited by a timing analyzer. This chapter described techniques to efficiently detect constraints from effects causing the outcome of a branch to become known and from ranges of iterations associated with branch outcomes. This constraint information could be used by a variety of timing analyzers, including those that use an ILP solver. These branch constraints were used in a non-ILP based timing analyzer to constrain the minimum and maximum iterations associated with each path in a loop and how these path constraints were used in WCET loop analysis. The results indicate that detection and exploitation of branch constraints can significantly tighten WCET timing

predictions. Furthermore, the approaches used for detection and exploitation of branch constraints were shown to be quite efficient and are fully automated, requiring no interaction from the user.

CHAPTER 6

SUMMARY RESULTS

Table 23 summarizes all of the WCET and BCET ratios for the various levels of analysis. The Naive Ratio column refers to no analysis being performed. Each of the remaining ratio columns shows the result of the analysis adding one feature to the column to its left. The ratios in the rightmost column represent all of the analysis described in this dissertation, taking into account instruction caching, pipelining, as well as automatic iteration calculation and branch constraint analysis. The results show that, on average, the WCET is predicted to within 1.1% of the observed worst-case time, and the BCET is predicted to within 3.0% of the observed best-case time.

Table 24 shows the response time of the timing analysis environment. All modules in the timing analysis environment have been compiled with optimizations. The first three columns give the percentage share of the total analysis time divided among the compiler, static cache simulator and the timing analyzer. In some rows of the table, the percentages do not total 100% due to rounding. The last column gives the execution time in seconds required for the timing analyzer to make the WCET and BCET predictions. The times were obtained by calculating for each program the average of the elapsed times of ten executions of the timing analyzer on an UltraSPARC. These response time measurements show that on average, the timing analysis takes only

Table 23: Estimated Ratios for Levels of Analysis

Worst-Case Results					
Name	Naive Ratio	Cache Only Ratio	+ Pipelining Ratio	+ Iter. Count Ratio	+ Br. Constr. Ratio
Des	5.144	2.663	1.152	1.152	1.117
Expint	50.384	34.426	22.215	22.215	1.001
Fresnel	2.222	1.533	1.024	1.024	1.001
Gaujac	2.006	1.466	1.004	1.004	1.000
Hes	12.302	7.251	2.339	1.016	1.016
Integ	4.419	2.420	1.332	1.001	1.001
Interp	4.208	2.966	1.991	1.000	1.000
LU	34.338	18.257	5.403	1.022	1.017
Matcnt	3.688	1.844	1.052	1.052	1.052
Matmul	4.977	2.108	1.001	1.001	1.001
Matsum	4.082	1.880	1.001	1.001	1.001
Sort	10.546	4.982	1.988	1.000	1.000
Sprsin	6.644	2.711	1.011	1.011	1.002
Stats	3.118	1.823	1.000	1.000	1.000
Summidall	13.834	6.787	1.179	1.179	1.000
Summinmax	12.511	6.349	1.062	1.062	1.000
Sumnegpos	14.379	7.054	1.181	1.181	1.000
Sumoddeven	12.943	6.373	1.068	1.068	1.001
Sym	26.114	8.980	1.995	1.000	1.000
Average	11.993	6.414	2.631	2.157	1.011
Best-Case Results					
Name	Naive Ratio	Cache Only Ratio	+ Pipelining Ratio	+ Iter. Count Ratio	+ Br. Constr. Ratio
Des	0.191	0.292	0.339	0.339	0.883
Expint	0.232	0.816	0.944	0.944	0.944
Fresnel	0.238	0.834	0.950	0.950	0.950
Gaujac	0.268	0.753	0.984	0.984	0.997
Hes	0.014	0.043	0.046	0.844	0.844
Integ	0.131	0.132	0.668	0.999	0.999
Interp	0.007	0.018	0.022	0.999	0.999
LU	0.017	0.018	0.022	0.049	0.920
Matcnt	0.247	0.659	1.000	1.000	1.000
Matmul	0.322	0.399	0.994	0.994	0.994
Matsum	0.257	0.761	0.923	0.923	0.923
Sort	0.481	0.495	0.999	0.999	0.999
Sprsin	0.419	0.900	0.997	0.997	0.997
Stats	0.300	0.687	0.990	0.990	0.990
Summidall	0.457	0.461	0.526	0.526	0.998
Summinmax	0.918	0.922	0.999	0.999	0.999
Sumnegpos	0.883	0.886	0.998	0.998	0.998
Sumoddeven	0.628	0.670	0.670	0.670	1.000
Sym	0.238	0.856	1.000	1.000	1.000
Average	0.329	0.558	0.741	0.853	0.970

Table 24: Response Time Measurements

Name	Compiler Percent	Static Cache Simulator Percent	Timing Analyzer Percent	Timing Analyzer Seconds
Des	32	5	63	1.35
Expint	38	4	58	.31
Fresnel	52	4	46	.22
Gaujac	11	1	88	3.08
Hes	81	1	18	.71
Integ	27	4	69	.15
Interp	36	3	61	.33
LU	70	1	29	1.17
Matcnt	43	6	50	.16
Matmul	40	5	55	.21
Matsum	41	6	53	.15
Sort	32	5	64	.28
Sprsin	44	5	51	.11
Stats	36	6	58	.36
Summidall	24	5	49	.06
Summinmax	59	5	36	.05
Sumnegpos	62	4	34	.04
Sumoddeven	60	4	37	.04
Sym	21	3	77	.21
Average	43	4	52	.47

slightly longer than compilation. However, there were a few anomalies. The programs *Hes* and *LU* took proportionally longer to compile since their source files were over 1,000 lines long, mostly due to the initialization of array elements. The timing analysis of *Gaujac* took significantly longer than the compilation since this program contains a loop with twelve long paths with many floating-point instructions. In addition, the timing analyzer examines each function instance separately, which adds to the response time whenever a function is called from multiple sites. The programs *Des* and *Stats* contain functions that are called from four or more sites.

The timing analyzer is now faster than the version that was used prior to the constraint research described in this dissertation [1, 2]. The decrease in elapsed time for the analysis was due to two reasons. First, the timing analyzer was modified to avoid

redundant analysis of a path when its caching behavior has not changed. Second, the new approach does not analyze a path in a given iteration when the path was infeasible, its maximum iterations had been exhausted, or only required iterations of other paths were available. Thus, the timing analyzer implemented for this dissertation remains a highly efficient tool.

CHAPTER 7

FUTURE WORK

There are additional aspects of using constraints in timing analysis that can be investigated. Many branch constraints were not detected due to function calls separating effects and the branches affected. These branch constraints could be detected using inter-procedural analysis. Similarly, inter-procedural analysis could also detect more loop iteration constraints, in the case where one loop contains a call to a function and another loop is in the called function. As was mentioned in Section 5.4, further branch constraints could also be obtained from analyzing values assigned to global variables and arrays.

Another goal is to make the tool more 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 Figure 1). At present, it is straightforward to retarget the timing analyzer to a similar type machine, a single issue RISC with a direct-mapped instruction cache. Retargeting to more varied machines, containing additional hardware features such as a secondary cache, non-blocking caches, register windows, multiple issue, etc. will require more work.

While the performance of the timing analyzer described in this dissertation 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. A logic analyzer inspects addresses sent on the bus to main memory. Unfortunately, it is difficult to measure execution time using a logic analyzer on a modern machine because many references to instructions and data are obtained from cache instead of main memory. It is also difficult to obtain accurate timing measurements due to the complexity of virtual memory and operating system overhead. Another disadvantage with only using execution time measurements is that capturing just the total execution time does not verify the correctness of the timing analyzer. It is possible that a faulty implementation for the timing analyzer could have an overestimation and an underestimation that cancel each other out, resulting in what initially appears to be a tight bound on the execution time.

Rather than focusing on the specific details of one machine, the author chose to use an existing retargetable hardware simulator. Assumptions were built into the simulator based on published documentation from the manufacturer. However, many details about the hardware were not specified, and the manufacturer's technical staff were not particularly forthcoming in responding to the author's questions via telephone or electronic mail. One advantage to using a simulator is that it is trivial to change certain aspects of the hardware such as the cache configuration. This feature is especially beneficial since the MicroSPARC I, designed in 1993, is becoming obsolete. Another reason the simulator was used was to allow one to examine lower levels of detail, such as pipeline stages of an individual instruction. With such fine-grain details, it is possible to test and debug the timing analyzer using simulator output.

On many embedded machines the measurement of the execution time can take place in the absence of virtual memory and operating system overhead. Some recent

architectures support hardware performance counters that are automatically updated during execution and introduce no additional overhead. One such counter can keep track of the number of machine cycles. Thus, in the future, the ideal testing environment would be on an embedded machine with both a simulator and hardware performance counters. A simulator of the architecture can be used to validate the timing analyzer. Next, hardware performance counters on an embedded machine can be used to validate the simulator. Using both testing components, one can then be more confident that the timing analyzer generates accurate WCET and BCET predictions for the actual machine.

CHAPTER 8

CONCLUSION

This dissertation has presented an extension to an earlier timing tool [1, 2] that now bounds execution time based on automatically generated constraints about the program. The two type of constraints are the calculation of the number of loop iterations discussed in Chapter 4 and the constraints related to branch outcomes described in Chapter 5.

The compiler calculates the number of loop iterations for loops having a single exit or multiple exits. In cases where this number of loop iterations depends on a non-constant loop invariant expression, the user can enter the minimum and maximum values for each variable in the expression interactively or through the use of assertions to bound the number of loop iterations. If the number of loop iterations depends on an outer loop's index variable, then the timing analyzer formulates a summation expression to be evaluated by an algebraic simplifier [47].

Constraints that can affect branch outcomes are automatically detected by the compiler, and a set of branch constraints is associated with each basic block. The timing analyzer propagates these branch constraints along each possible path of execution to determine path constraints. The analysis of path constraints can determine if certain paths are infeasible or if they can only execute on a certain set of iterations. These path constraints are used in the worst-case and best-case loop analysis algorithms to more tightly bound the execution time. For instance, a loop's worst case (best case) time can

be more tightly predicted if is determined that the longest (shortest) path is infeasible.

The implementation of the timing analysis environment includes the four modules depicted in Figure 1. The compiler, static cache simulator and the algebraic solver that support the timing analysis were implemented by other researchers at FSU. The total response time including the compilation, static cache simulation and timing analysis, takes only a few seconds for the benchmark programs used in this dissertation.

The major contribution of this dissertation to timing analysis research is the nature in which constraints are made known and used within the timing analyzer. In the past, such constraints were manually entered by the user, which is quite tedious and error prone. Using the techniques described in this dissertation, it has been shown that this process can be automated. It is much more likely that a real-time programmer would use an automated tool as opposed to an unautomated one to obtain timing predictions in order to relieve the tedium and get a quicker response. The new version of the timing analyzer bounds the WCET and BCET much more tightly than before. In addition, this analysis can still be performed in a small amount of time.

APPENDIX

The following context-free grammar describes the syntax of the file that is created by the compiler and used as input into the timing analyzer. The file contains information about the control flow of the program and the branch constraints. The grammar follows the following syntax conventions.

1. The definition operator is "::=".
2. Nonterminals are enclosed in angle brackets.
3. A superscript asterisk indicates that an expression may have zero or more instances. A superscript plus indicates one or more instances.
4. Optional expressions are enclosed in square brackets.
5. Parentheses are used as grouping symbols.
6. A boldface token represents a terminal whose value is not specified. A token appearing in the standard font is a terminal that appears verbatim in the information file.
7. Concatenation implies that the expressions appear in the specified order.
8. The vertical bar (|) separates alternatives within an expression.

<Inf_file> ::= <Function>*

<Function> ::= **func_identifier** **new_line** <Func_name> **new_line** <loop>* <block>*

<Func_name> ::= <letter> (<letter> | <digit> | _)*

<loop> ::= **loop_identifier** <loop_number> <nesting_level> [<induction_var>] [<iteration_info>] [<user_provided>]

<min_iterations> <max_iterations> <lower_bound> <upper_bound> <block_list> **new_line**

<loop_number> ::= <nonnegative_integer>

<nesting_level> ::= <nonnegative_integer>

<induction_var> ::= **induction_var_identifier** <register> <init_value_rtl> <limit_rtl> <increment>

<init_value_rtl> ::= <rtl>

<limit_rtl> ::= <rtl>

<increment> ::= <integer>

<iteration_info> ::= **iteration_info_identifier** <iter_mode> <inner_initial> <inner_limit> <relop> <inner_incr> <outer_number> ((c <outer_initial_value>) | (r | <outer_initial_rtl>)) ((c <outer_limit_value) | (r <outer_limit_rtl>)) <outer_increment>

<iter_mode> ::= 0 | 1

<inner_initial> ::= <integer> | <rtl>

<inner_limit> ::= <integer> | <rtl>

<relop> ::= **equal_id** | **not_equal_id** | **greater_id** | **less_id** | **greater_or_equal_id** | **less_or_equal_id**

<inner_incr> ::= <integer>

<outer_number> ::= <positive_integer>

<outer_initial_value> ::= <integer>

<outer_initial_rtl> ::= <rtl>

<outer_limit_value> ::= <integer>

<outer_limit_rtl> ::= <rtl>

<outer_increment> ::= <integer>

<user_provided> ::= **user_provided_identifier** (<integer> | <variable_name>) [(+ | –) (<integer> | <variable_name>)]*

<min_iterations> ::= <positive_integer>

<max_iterations> ::= <positive_integer>

<lower_bound> ::= <nonnegative_integer> [s | m | u | n]

<upper_bound> ::= <nonnegative_integer> [s | m | u | n]

<block_list> ::= <integer_list>

<block> ::= <block_intro> <effects> <doms> <inst_list>

<block_intro> ::= **block_identifier** <block_number> lines <begin_line> – <end_line> preds <pred_list> succs <succ_list> **new_line**

<block_number> ::= <positive_integer>

<begin_line> ::= <positive_integer>

<end_line> ::= <positive_integer>

<pred_list> ::= <integer_list>

<succ_list> ::= <integer_list>

<effects> ::= [makes_unknown <integer_list> **new_line**]

[makes_fallthru <integer_list> **new_line**]

[makes_branch <integer_list> **new_line**]

[makes_unknown_if_not_fellthru <integer_list> **new_line**]

[makes_unknown_if_not_branched <integer_list> **new_line**]

[fallthru_causes_branch <integer_list> **new_line**]

[fallthru_causes_fallthru <integer_list> **new_line**]

[branch_causes_branch <integer_list> **new_line**]

[branch_causes_fallthru <integer_list> **new_line**]

[(<iters_range> | <iters_once>) **new_line**]

<iters_range> ::= iters_range <range> <range>

<range> ::= **[**<positive_integer> .. <positive_integer>**]**

<iters_once> ::= iters_once (R | L) (<positive_integer> |

non_guarantee_id | **will_occur_id**)

<doms> ::= doms <integer_list> **new_line**

<inst_list> ::= <instruction>*

<instruction> ::= <block_number> <opcode> <data_type>

<condition_code> <operand_info> <operand_info>

<operand_info> **new_line**

<opcode> ::= <nonnegative_integer>

<data_type> ::= <nonnegative_integer>

<condition_code> ::= <nonnegative_integer>

<operand_info> ::= <operand_addr> <operand_data_type> <operand>

<operand_addr> ::= <nonnegative_integer>

<operand_data_type> ::= <nonnegative_integer>

<operand> ::= **(** [<integer> | (<SPARC_register> [<SPARC_register>])] **)**

<SPARC_register> ::= % (g | o | l | i | f) <nonnegative_integer>

<integer_list> ::= <positive_integer>$^{+}$ **list_terminator**

<rtl> ::= <register> [(+ | –) <nonnegative_integer>]

<register> ::= r **[** <nonnegative_integer> **]**

REFERENCES

[1] C. A. Healy, *Predicting Pipeline and Instruction Cache Performance,* Masters Thesis, Florida State University, Tallahassee, FL (1995).

[2] C. Healy, R. Arnold, F. Mueller, D. Whalley, and M. Harmon, "Bounding Pipeline and Instruction Cache Performance," *IEEE Transactions on Computers* **48**(1) pp. 53-70 (January 1999).

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

[4] 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).

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

[6] 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).

[7] 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).

[8] Y. Hur, Y. H. Bae, S. S. Lim, S. K. Kim, B. D. Rhee, S. L. Min, C. Y. Park, H. Shin, and C. S. Kim, "Worst Case Timing Analysis of RISC Processors 1995: R3000/R3010 Case Study," *Proceedings of the Sixteenth IEEE Real-Time Systems Symposium*, pp. 308-321 (December 1995).

[9] Y. S. Li, S. Malik, and A. Wolfe, "Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software," *Proceedings of the Sixteenth IEEE Real-Time Systems Symposium*, pp. 298-307 (December 1995).

[10] R. T. White, F. Mueller, C. A. Healy, D. B. Whalley, and M. G. Harmon, "Timing Analysis for Data Caches and Wrap-Around Fill Caches," *Real-Time Systems*, (accepted April 1999).

[11] G. Ottosson and M. Sjödin, "Worst Case Execution Time Analysis for Modern Hardware Architectures," *ACM SIGPLAN Workshop on Language, Compiler, and Tools for Real-Time Systems*, pp. 47-55 (June 1997).

[12] H. Theiling and C. Ferdinand, “Combining Abstract Interpretation and ILP for Microarchitecture Modeling and Program Path Analysis,” *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pp. 144-153 (December 1998).

[13] C. Y. Park and A. C. Shaw, “Experiments with a Program Timing Tool Based on a Source-Level Timing Schema,” *Computer* **24**(5) pp. 48-57 (May 1991).

[14] R. Chapman, A. Wellings, and A. Burns, “Integrated Program Proof and Worst Case Timing Analysis of SPARK Ada,” *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, (June 1994).

[15] 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).

[16] A. Ermedahl and J. Gustafsson, “Deriving Annotations for Tight Calculation of Execution Time,” *Proceedings of European Conference on Parallel Processing*, pp. 1298-1307 (August 1997).

[17] T. Lundqvist and P. Stenström, “Integrating Path and Timing Analysis using Instruction-Level Simulation Techniques,” *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pp. 1-15 (June 1998).

[18] Y. Liu and G. Gomez, “Automatic Accurate Time-Bound Analysis for High-Level Languages,” *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pp. 31-40 (June 1998).

[19] C. Ferdinand, *Cache Behavior Prediction for Real-Time Systems*, PhD Dissertation, Universität des Saarlandes, Saarbrücken, Germany (September 1997).

[20] C. Ferdinand, F. Martin, and R. Wilhelm, “Applying Compiler Techniques to Cache Behavior Prediction,” *ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, pp. 37-46 (June 1997).

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

[22] 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).

[23] 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).

[24] 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).

[25] 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).

[26] 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).

[27] L. Ko, N. Al-Yaqoubi, C. Healy, E. Ratliff, R. Arnold, D. Whalley, and M. Harmon, "Timing Constraint Specification and Analysis," *Software Practice & Experience,* pp. 77-98 (January 1999).

[28] R. White, *Bounding Worst-Case Data Cache Performance,* PhD Dissertation, Florida State University, Tallahassee, FL (April 1997).

[29] R. T. White, F. Mueller, C. A. Healy, D. B. Whalley, and M. G. Harmon, "Timing Analysis for Data Caches and Set-Associative Caches," *Proceedings of the IEEE Real-Time Technology and Applications Symposium,* pp. 192-202 (June 1997).

[30] Nagham M. Al-Yaqoubi, *Reducing Timing Analysis Complexity by Partitioning Control Flow,* Masters Project, Florida State University, Tallahassee, FL (1997).

[31] C. A. Healy, M. Sjodin, V. Rustagi, and D. B. Whalley, "Bounding Loop Iterations for Timing Analysis," *Proceedings of the IEEE Real-Time Technology and Applications Symposium,* pp. 12-21 (June 1998).

[32] C. A. Healy and D. B. Whalley, "Tighter Timing Predictions by Automatic Detection and Exploitation of Value-Dependent Constraints," *Proceedings of the IEEE Real-Time Technology and Applications Symposium,* pp. 79-88 (June 1999).

[33] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing,* Cambridge University Press, New York, NY (1988).

[34] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing, Second Edition,* Cambridge University Press, New York, NY (1992).

[35] 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).

[36] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach, Second Edition,* Morgan Kaufmann, San Francisco, CA (1996).

[37] M. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," *Proceedings of the SIGPLAN '88 Symposium on Programming Language Design and Implementation,* pp. 318-328 (June 1988).

[38] H. S. Stone, *High-Performance Computer Architecture, Second Edition,* Addison Wesley, Reading, MA (1990).

[39] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA (1986).

[40] L. Ko, C. Healy, E. Ratliff, R. Arnold, D. Whalley, and M. Harmon, "Supporting the Specification and Analysis of Timing Constraints," *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pp. 170-178 (June 1996).

[41] R. Sakellariou, *Symbolic Evaluation of Sums for Parallelising Compilers*, Wissenschaft & Technik Verlag, Proceedings of the 15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics (1997).

[42] R. Sakellariou, *On the Quest for Perfect Load Balance in Loop-Based Parallel Computations*, PhD Dissertation, Department of Computer Science, University of Manchester, Manchester, England (October 1996).

[43] R. van Engelen, L. Wolters, and G. Cats, "Ctadel: A Generator of Multi-Platform High Performance Codes for PDE-based Scientific Applications," *Proceedings of the 10th ACM International Conference on Supercomputing*, pp. 86-93 (May 1996).

[44] R. van Engelen, L. Wolters, and G. Cats, "Tomorrow's Weather Forecast: Automatic Code Generation for Atmospheric Modeling," *IEEE Journal of Computational Science and Engineering* **4**(3) pp. 22-31 (September 1997).

[45] M. J. Wolfe, *High Performance Compilers for Parallel Computers*, Addison-Wesley, Redwood City, CA (1996).

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

[47] C. A. Healy, R. van Engelen, and D. B. Whalley, "A General Approach for Tight Timing Predictions of Non-Rectangular Loops," *WIP Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pp. 11-14 (June 1999).

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 and the Master of Science degree in computer science from F.S.U. in 1995. During his six years in the graduate program, he was a part-time teaching assistant for four years and a research assistant for two. Starting in fall 1999, he will be employed as an assistant professor of computer science at Furman University.

ACKNOWLEDGEMENTS

I wish to thank my major professor, Dr. David Whalley, for his patience, guidance and support during my research, and for making the needed changes to the compiler to support the latest implementation of the timing analyzer. I am also grateful for Dr. van Engelen's assistance in integrating his Ctadel environment with the timing analyzer. He implemented the algebraic solver used by the timing analyzer to compute summations representing the number of iterations of non-rectangular loops. I also thank Dr. Lacher, Dr. Baker, Dr. van Engelen, Dr. Gallivan and Dr. Bellenot for the helpful suggestions they had during the writing of the prospectus and this dissertation. The timing analyzer described in this dissertation 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. Viresh Rustagi and Mikael Sjödin also provided valuable assistance on calculating loop iterations for the timing analysis environment. The research upon which this dissertation is based was supported in part by the Office of Naval Research under contract number N00014-94-1-0006 and the National Science Foundation under grant number EIA-9806525.

TABLE OF CONTENTS

Page

LIST OF TABLES

TABLE NUMBER AND DESCRIPTION **PAGE**

LIST OF FIGURES

FIGURE NUMBER AND DESCRIPTION **PAGE**

ABSTRACT

Predicting the worst-case execution time (WCET) and best-case execution time (BCET) of a real-time program is a challenging task. Though much progress has been made in obtaining tighter timing predictions by using techniques that model the architectural features of a machine, significant overestimations of WCET and underestimations of BCET can still occur. It is essential to accurately calculate the number of loop iterations for all loops in order to tightly bound the program's execution time since most of a typical program's execution takes place inside of loops. In addition, dependences on data values can constrain the outcome of conditional branches and the corresponding set of paths that can be taken in a program. This dissertation describes how timing analysis can be improved when these two types of constraints are addressed. First, the minimum and maximum number of iterations are automatically calculated. Loops with multiple exit conditions or a varying number of iterations are also addressed. Second, constraints on branches are automatically detected during compilation. These branch constraints are then used to determine how many times each path in a loop or function can be taken. Finally, this iteration and branch constraint information is automatically utilized to obtain tighter bounds on the execution time. Not only does the timing analysis provide significantly tighter WCET and BCET predictions, the analysis response time is typically faster as well.