THE FLORIDA STATE UNIVERSITY
COLLEGE OF ARTS AND SCIENCES

Reducing Timing Analysis Complexity by Partitioning Control Flow

By
NAGHAM M. AL-YAQOUBI

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

Degree Awarded:
Summer Semester, 1997

The members of the Committee approve the project of Nagham Al-Yaqoubi defended on July 24, 1997.

David B. Whalley
Professor Directing Project

Theodore P. Baker
Committee Member

Stephen P. Leach
Committee Member

Approved:

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

To Yasir, Noor, Amar, Sumia, and Arwa

Acknowledgements

I wish to thank my major professor, Dr. David Whalley for his patience, guidance and support during my project. I am also grateful for the helpful suggestions given by Christopher Healy. I am also grateful for the assistance Gang-Ryung Uh and Randy White offered in typesetting this document. The timing analyzer upon which this project is based was created by Robert Arnold and Christopher Healy. Lo Ko and Emily Ratliff implemented the graphical user interface.

Contents

List of Tables

List of Figures

Chapter 1
INTRODUCTION

To assist real-time programmers in the specifications and analysis of timing constraints, a timing analyzer and graphical user interface were developed. The timing analyzer analyzes the paths through each loop and function associated with the control flow in a C program. The analyzer uses this path analysis information to estimate the execution time of each loop and function. Afterwards, a user interface is invoked to allow the user to request timing predictions on portions of the program.

However, program functions and loops may have many control statements that can cause exponential growth in the number of possible paths through a loop or function. To illustrate this concept, consider the function and the four possible paths within this function in Figure 1.1. The T's represent the conditions taken and the NT's the conditions not taken. Each row represents different possible paths. If there is a loop containing six *if* statements at the same level, then there will be 64 paths through that loop. Likewise, a loop with ten *if* statements will have 1024 paths. So for a loop with n *if* statements, there would be 2^n paths through that loop.

Attempting to obtain timing predictions for a function or loop with complex control flow poses problems for both the timing analyzer and the user interface. The timing analyzer will abort if it is unable to dynamically allocate enough

```
main ()
{
  if (condition 1);
    statement;
  if (condition 2);
    statement;
}
```

PATH NO.	COND. 1	COND. 2
path 1	T	T
path2	T	NT
path3	NT	T
path4	NT	NT

Figure 1.1: Code Segment with Two Control Statments

space to represent all of the paths. Furthermore, even if enough space can be allocated to represent all of the paths in each loop and function, a large number of paths will result in a significant increase in the execution time of the timing analyzer. Likewise, the user interface may be unable to allocate enough space to graphically represent the paths at each loop and function level. The user interface allows a user to select a path for timing prediction from the list of possible paths. Few users would bother searching through a very long list of paths.

This document describes a technique to simplify the control flow of complex functions or loops by partitioning the control flow into sections that are limited to a predefined number of paths. Each section is treated by the timing analyzer as a loop that iterates only once (similar to a function). After the timing analyzer was modified to partition complex loops and functions into sections and to analyze these sections, the user interface was updated to display the sections as a new selection level between loops and paths.

Chapter 2

OVERVIEW OF OBTAINING TIMING PREDICTIONS

Figure 2.1 gives an overview of the context in which timing predictions are obtained. Control-flow information is stored as the side effect of the compilation of a file. This control-flow information is passed to a static cache simulator. It constructs the control-flow graph of the program that consists of the call graph and the control flow of each function. The program control-flow graph is then analyzed for a given cache configuration and a categorization of each instruction's potential caching behavior is produced.

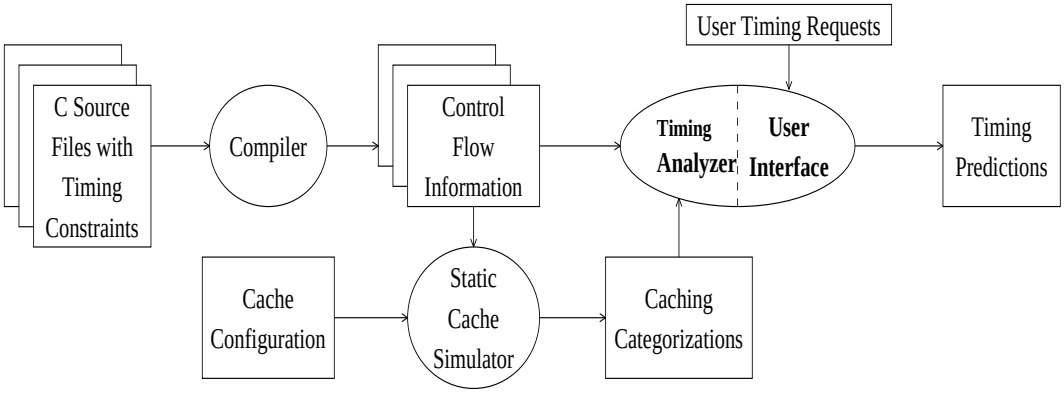


Figure 2.1: Overview of Obtaining Timing Predictions

Next, a timing analyzer uses the instruction caching categorizations along with the control-flow information provided by the compiler, which includes the

source lines associated with basic blocks, to estimate the worst-case instruction caching performance for each loop within the program. Once the timing analyzer has evaluated all functions within the program, a user interface is invoked to allow the user to request timing bounds for specific code segments within the program.

Chapter 3
IMPORTANT CONCEPTS AND DEFINITIONS

The timing analyzer constructs a *timing tree* to simplify the process of determining the execution bounds of a program. Each node in the tree represents a function or natural loop in the program. Functions are analyzed as though they are a natural loop that iterates only once when entered.

The creation of the timing tree requires the analysis of the program's code in order to determine information regarding the loops within each function. The optimizing compiler initiates this analysis by identifying for each loop: the nesting level, all the blocks contained within the loop, all exit blocks from the loop, the minimum number of loop iterations, and the maximum number of loop iterations. The timing tool extends this analysis by determining all possible paths through the loop.

A *basic block* is defined as a sequence of consecutive instructions in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end. A *loop header block* is defined as the unique entry block into the loop. Blocks outside the loop that are reached by control-flow transitions from blocks within the loop are defined as *loop exit blocks*. A *function header block* is defined as the unique entry block into the function. A *function exit block* is the block containing a return instruction. A *path* is a sequence of unique blocks in the loop connected by control-flow

transitions. Each path in the loop must start with the loop header block and terminate with a block containing a transition to the header block (*continue path*) or to an exit block (*exit path.*) The path through a function is defined to start with the function header block and end with the function exit block. If a path within a loop or a function contains a nested loop, then the entire nested loop is represented in the path by only the header block of the nested loop. Associated with each loop is the set of exit blocks for that specific loop.

To illustrate these definitions and concepts consider a loop depicted by the block diagram in Figure 3.1. Each block is represented by a numbered box. This loop has block 2 as a header block. Blocks 5 and 7 are not part of the loop, but rather exit blocks from the loop. This loop is nested in an outer loop (actually a function) that has block 1 as its loop header block, and it contains two paths, one containing block 5 but not block 7, and the other containing block 7 but not block 5.

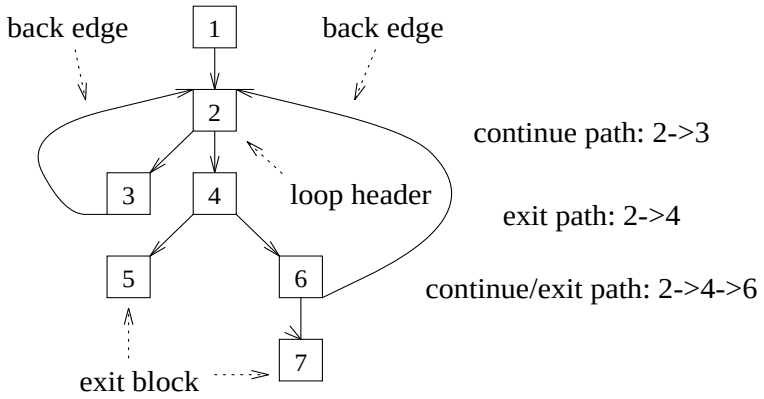


Figure 3.1: Examble Introducing Loop Terminology

Chapter 4

SECTION APPROACH IN THE TIMING ANALYZER AND THE USER INTERFACE

While the main purpose of partitioning loops into sections is to help the user obtain an accurate timing prediction in an efficient manner regardless of the complexity of the analyzed program, there are some implementation issues that have to be considered when implementing sections. First, the approach for creating sections, when necessary, and modifying the timing analyzer to process sections, should not increase the execution time of the timing analyzer. In fact, the use of sections should decrease the execution time since the total number of paths that need to be analyzed after section creation will be less than before sections. Second, the timing analyzer should not recognize any differences between sections and loops. The sections should analyzed as if they are loops that only iterate once, which is similar to the way functions are processed. This approach will minimize the changes to the timing analyzer and other portions of the timing analysis process. Finally, updating the timing analyzer to create the sections and analyze them should not significantly affect the accuracy of the timing prediction results.

4.1 Section Approach in The Timing Analyzer

The timing analyzer reads the control-flow information of the program from the INF file. This file contains information identifying the loops within each

function, the blocks within each loop, and the instructions within each block. This file also includes the lists of the predecessor, successor, and dominator blocks for every block in each function. The timing analyzer uses this static information from the INF file to find all possible paths through each loop and function. Once the static analysis is complete, the timing analyzer begins to construct the timing analysis tree, where each node of the tree represents either a loop or a function in the function instance graph. Finally, the timing analyzer determines the execution time of the program by analyzing each node in the tree starting from the innermost loops and functions, and proceeding to higher level loops until it reaches the *main()* function.

The section approach in the timing analyzer consists of the following steps. First, the loops whose number of paths exceeds the specified threshold are identified. Second, sections are represented using the same data structure that is used to represent loops in the timing analyzer. Third, the available utilities for analyzing paths, building the timing tree, and calculating the execution time are updated to handle sections as if they were loops that iterated only once.

4.1.1 Find the Maximum Number of Paths Through a Loop

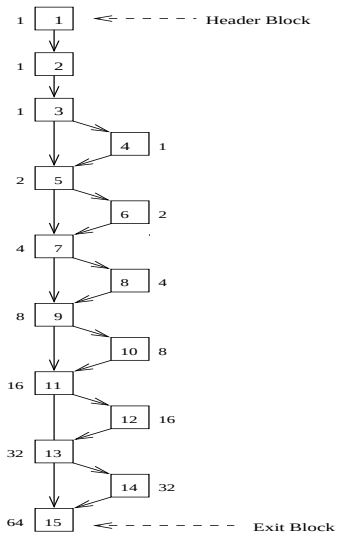
The maximum number of paths through each loop and function should be known before deciding to create sections since only loops with a number of paths that exceeds a given threshold will be partitioned into sections. The algorithm shown in Figure 4.1 was used to find the maximum number of paths in each loop of a program. The path number associated with each block is used to represent the number of paths that will pass through that block. The path number also serves

```
(1) Set the current loop header block's path number to one.  Set the path
    number of all other blocks in the current loop at that level to zero.
    Remember that a loop that is directly nested in the current loop will
    be represented as a single block at the current loop level.

(2) DO
    {
      FOR( each basic block in the loop at that level)
        {
        IF the current block has not been processed (path number == 0),
THEN:

         (a) Determine the list of predecessor blocks to the current basic
             block.

         (b) IF all the blocks in the predecessor block list has been
             processed, THEN:

             -  Calculate the path number for the current block to be
                the sum of the path numbers associated with each block in
                the predecessor list.
        }
    }WHILE (more blocks to  process )

(3) The maximum number of paths through the loop is equal to the sum of
    the values of the path numbers associated with each block within the
    loop that have a transition to blocks outside the loop.
```

Figure 4.1: Find the Maximum Number of Path Algorithm

```
#include <stdio.h>
main ()
{
   int i, j;

   printf("please enter a number: ");
   scanf("%d", &i);
   if ( i == 1 )
      i = i + 1;
   if ( i == 2 )
      i = j;
   if ( i == 3 )
      i = i + 3;
   if ( i == 4 )
      ++i;
   if ( i == 5 )
      printf(" i = %d\n", i);
   if ( i == 6 )
      printf(" j = %d\n", j);
}
```

Source Code of Toy6



Control Flow of Toy6

Figure 4.2: Source Code and Block Diagram for *Toy6*

as a sign that this block has been processed. The central idea of the algorithm is that the path number of a block is calculated as the sum of the path numbers of its immediate predecessors. To illustrate the algorithm, consider the source code of *Toy6* and the corresponding control flow diagram show in Figure 4.2. The algorithm first initializes the path number associated with each block to zero, except for the header block that is initialized to one. The *for* loop in Figure 4.1 iterates for each block in the loop, *main()* at that level. The header block (block 1) has already been marked as processed, so the first block examined will be block 2. It inherits its path number from its only predecessor. Block 5 receives a path number of 2, which is the sum of the path numbers in blocks 3 and 4. After all of the blocks have been processed, block 15 has a path number of 64. Thus, the number of distinct paths through the loop will be 64.

4.1.2 Splitting a Loop into Sections

As stated previously, sections are implemented to decrease the execution time of the timing analyzer and to reduce the amount of dynamically allocated space needed to represent all of the paths when analyzing a program with complex control flow. Therefore, the number of paths through each loop is calculated and the sections are created before the timing analyzer begins to allocate space for the paths. The algorithm to find and create the sections is given in Figure 4.3. The central idea of the algorithm is to find the first block where the path number (as used in Figures 4.1 and 4.2) exceeds the specified threshold and to replace the blocks collected up to that point with a section. Each section is then treated like a single block at that loop level. The algorithm iterates until the

```
(1)  Allocate an empty block_list structure sub_list and set a pointer to
     it.

DO
{

   (2)  IF sub_list is not empty THEN:

    a.  Create a section, copy the sub_list to the section's block list.
    b.  Increment the section number, the new header number, and
        the loops nesting level to include the new section.
    c.  Remove the blocks from the loop's block list that are in the
        sub_list.
    d.  Create a new block for the new header and link it at the
        beginning of the loop's main  block list.

   (3)  Set the loop header block's path number to one and set the other
        blocks' path numbers to zero in the loop's block list.

   (4)  Clear sub_list. Copy the loop's header block to it.

   DO
   {
    (5)  FOR (each basic block in the loop's main block list)
         {
            IF this is the first time to visit the current block THEN:

            (a) Determine the list of predecessor blocks to the current
                basic block.

            (b) IF all the predecessor blocks have been visited AND
                   all the predecessor blocks exist in the sub_list THEN

            .  Append the current block to the sub_list.

            .  Calculate the path number for the current block to be
               equal to the sum of the path numbers associated with each
               predecessor block.

            .  IF the path number is larger than threshold THEN
                 - Delete the current block from the sub_list.
                 - CONTINUE.

            .  Find the total number of paths through the sub_list by
               adding the path numbers associated with the blocks that
               have transitions out of the sub_list.

            .  IF the total number of paths through the sub_list is
                  larger than the threshold THEN
                 - Delete the current block from the sub_list.

         }

   }(6)  WHILE there are more blocks in the loop's block list to proces.

}(7)  WHILE loop's block list is NOT the same as sub_list.

(8)  IF there are now sections within the loop THEN
   a. Create a new block for the header and link it at the beginning of
      the loop's block list.
   b. Calculate the new exit block list.
   c. Update the block list of the loop to include the new header blocks.
```

Figure 4.3: Find and Create Sections Algorithm

number of paths within the loop is within the specified threshold. To illustrate the algorithm, consider the control flow diagram in Figure 4.4 and the variables value in Table 4.1, assuming the threshold is set to 4. The algorithm collects

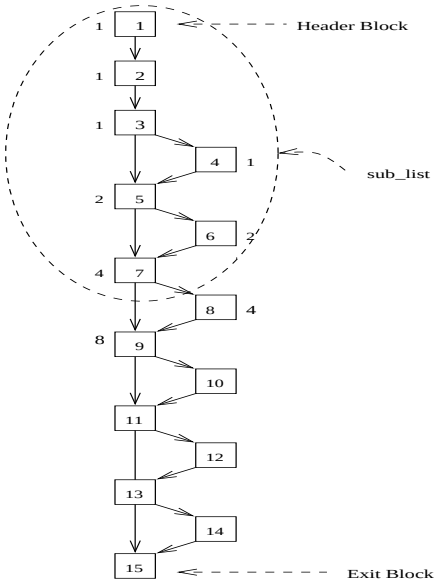


Figure 4.4: Control Flow after the First Pass

iteration no.	current block	sub_list	num. of paths through current block	blocks no. that sub_list exits from	total number of paths through sub_list
1st	1	1	1	1	1
2nd	2	1,2	1	2	1
3rd	3	1,2,3	1	3	1
4th	4	1,2,3,4	1	3,4	1
5th	5	1,2,3,4,5	2	5	2
6th	6	1,2,3,4,5,6	2	5,6	4
7th	**7**	**1,2,,3,4,5,6,7**	**4**	**7**	**4**
8th	**8**	**1,2,3,4,5,6,7,8**	**8**	**7,8**	**8**
9th	**9**	**1,2,3,4,5,6,7,9,**	**8**	**9**	**8**
10th	10	1,2,3,4,5,6,7	NA	7	4
11th	11	1,2,3,4,5,6,7	NA	7	4
12th	12	1,2,3,4,5,6,7	NA	7	4
13th	13	1,2,3,4,5,6,7	NA	7	4
14th	14	1,2,3,4,5,6,7	NA	7	4
15th	15	1,2,3,4,5,6,7	NA	7	4

Table 4.1: Paths Numbers and the Sublist during the First Pass

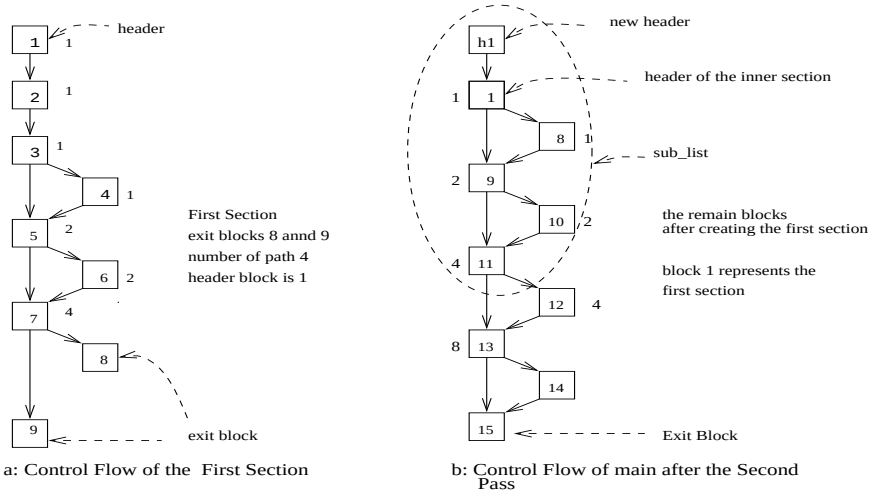


Figure 4.5: Control Flow after the Second Pass

iteration no.	current block	sub_list	num. of paths through current block	blocks no. that sub_list exit from	total number of paths through sub_list
1st	h1	h1	1	h1	1
2nd	1	h1,1	1	1	1
3rd	8	h1,1,8	1	1,8	1
4th	9	h1,1,8,9	2	9	2
5th	10	h1,1,8,9,10	2	9,10	4
6th	11	h1,1,8,9,10,11	4	11	4
7th	12	h1,1,8,9,10,11,12	4	11,12	8
8th	13	h1,1,8,9,10,11,13	8	13	8
9th	14	h1,1,8,9,10,11	NA	11	4
10th	15	h1,1,8,9,10,11	NA	11	4

Table 4.2: Paths Numbers and the Sublist during the Second Pass

blocks 1 through 8 in a sublist. At this point there are two transitions out of the sublist from blocks 7 and 8. Both of these two blocks have a path number of 4, which results in a total of 8 paths for the sublist. Thus, block 8 is deleted from the sublist and a section is created for blocks 1 through 7 as shown in Figure 4.5a. during the second pass, an empty block with an unique number is added as a new header for the loop and the first section is treated as single block using the section's header (block 1) as shown in Figure 4.5b and Table 4.2. A similar scenario takes place through the last pass of the algorithm and a second section is created for the new block, block 1, and blocks 8 through 11 as shown in Figure 4.6 and Table 4.3.

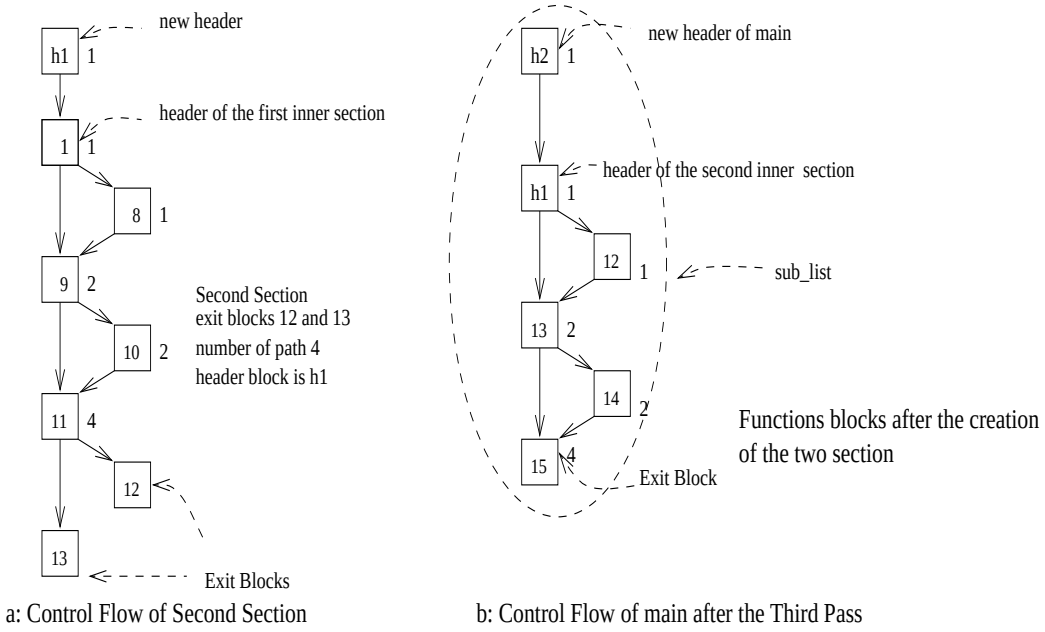


Figure 4.6: Block Diagram after the Third Pass

iteration no.	current block	sub_list	num. of paths through current block	blocks no. that sub_list exit from	total number of paths through sub_list
1st	h1	h1	1	h1	1
2nd	1	h1,1	1	1	1
3rd	8	h1,1,8	1	1,8	1
4th	9	h1,1,8,9	2	9	2
5th	10	h1,1,8,9,10	2	9,10	4
6th	11	h1,1,8,9,10,11	4	11	4
7th	12	h1,1,8,9,10,11,12	4	11,12	8
8th	13	h1,1,8,9,10,11,13	8	13	8
9th	14	h1,1,8,9,10,11	NA	11	4
10th	15	h1,1,8,9,10,11	NA	11	4

Table 4.3: Paths Numbers and the Sublist during the Last Pass

4.1.3 Timing Tree

The timing analyzer constructs a timing tree to calculate the worst and best case bounds of a program. The timing analyzer uses the function call graph and the control flow information from each function in the program to construct a timing tree. Figure 4.8 depicts the timing tree before and after section creation for the program in Figure 4.7.

The root node at nesting level 0 represents the *main* function. The two immediate children of *main* are Loop 1 and Loop 2 and are at nesting level 1. Loop 3 is represented with the leaf node at nesting level 2.

The timing tree after creating sections for the same program for a *threshold* equal to 4 is depicted in Figure 4.8. There were originally 32 paths through Loop 1. Therefore, the complex control flow was simplified by creating Sections

```
main()
{
  int i, j, a[10][10];

  for (i = 0; i < 10; ++i)        /* loop_1: outer loop */
  {
     if ( i == 0 )
        a[i][0] = i ;
     if ( i == 1 )
        a[0][i] = i ;
     if ( i <= 4 )
        for ( j = 1; j < 10; ++j) /* loop_3: inner loop */
           a[i][j] = i - j;
     if ( i == 4 )
        j = 0;
     if ( i > 4 )
        a[i][j] = j;
  }
  for (i = 10; j < 10; ++j)       /* loop_2: outer loop */
     a[i][j] = i - j;
}
```

Figure 4.7: Program Containing Three Loops

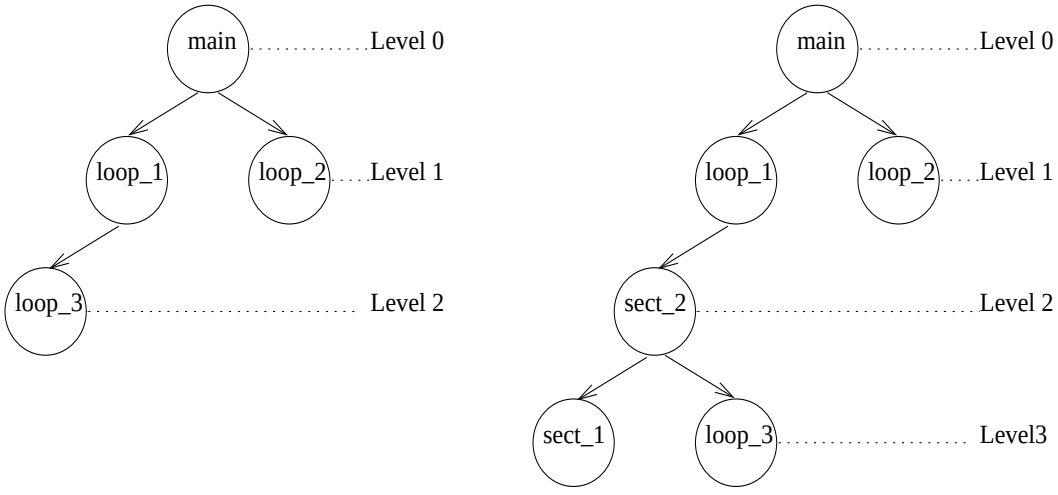


Figure 4.8: Timing Tree

1 and 2 within Loop 1 using the algorithm described in Figure 4.3. Section 2 becomes the parent for Loop 3 since it contains Loop 3's header block. Section 2 is also the parent of Section 1 for the same reason.

4.2 Section Approach in the User Interface

The timing prediction for each loop, section, and function is stored within the node that represents it in the timing tree. Each node contains the predicted time for the execution of the instructions at that loop level plus the time bound of its immediate children. The user interface uses the timing tree to obtain timing predictions for the user-requested program portions.

After the timing analyzer has analyzed the entire program, the user interface is invoked to provide a tool that allows a user to quickly obtain best-case and worst-case timing predictions for selected portions of the program. Figure 4.9 and Figure 4.10 depict three windows that are always displyed when the user interface is executing. Figure 4.9 shows the main window, which describes the portion of the program that is currently selected. The top section of the main window displays a message indicating the current action the user can perform in the middle section. The middle section of the main window has a specific portion highlighted, which indicates the current program construct for which best-case and worst-case timing predictions are displayed in the lower part of this section. The bottom section of the main window contains buttons that allow the user to select the level of information displayed. Selection of the More Detail button permits the user to view the current program portion in finer detail. The Back button is selected when the user desires to back up to a previous level of detail.

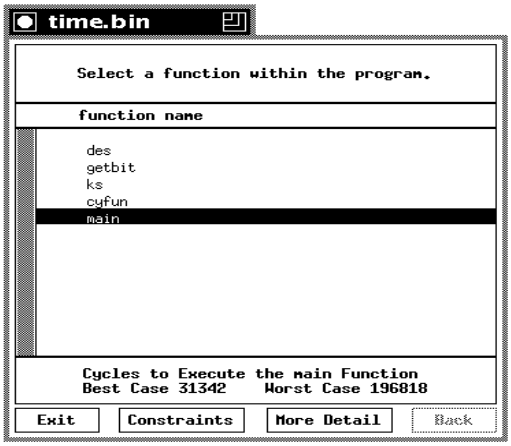


Figure 4.9: Main Window at Function Level

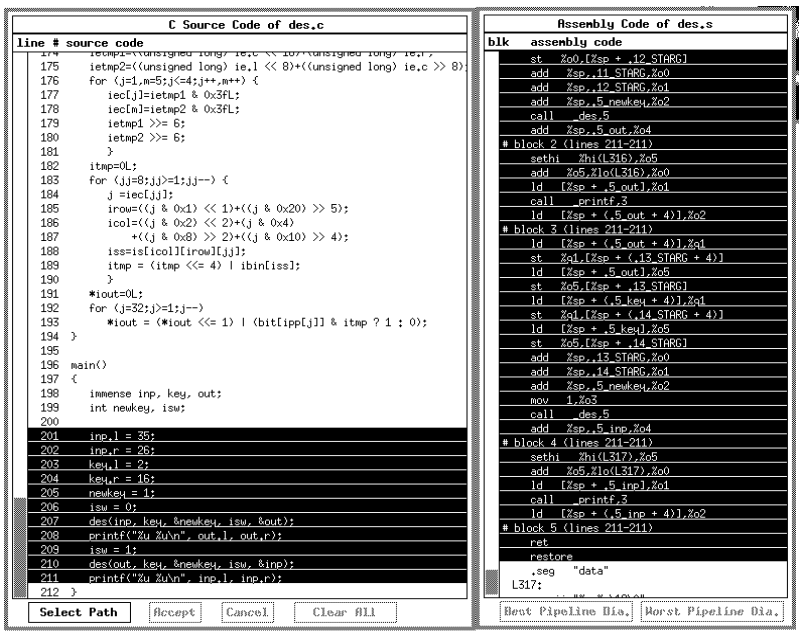


Figure 4.10: Source Code and Assembly Code Window

Figure 4.10 shows the two other windows in the user interface that are always displayed. The left window depicts the C source code and the right window shows the corresponding assembly code. Whenever the user selects a different construct from the middle part of the main window, the corresponding lines in the source and assembly windows are highlighted simultaneously.

The user can obtain timing predictions either by using the main window to access different portions of the program, or by highlighting the desired path in the source window using the mouse. For the first method, the user can click the More Detail button to view a lower level of detail for the selected construct in the middle section of the main window.

There are six levels of detail a user is allowed to view. The top level and initial display for the middle section of the main window is the list of functions within the program. This level is depicted in Figure 4.9.

The next lower level of detail consists of loops as shown in Figure 4.11. The entire function and each loop within the function are listed in the display. Note that if there are no loops within the selected function then the user interface will skip the loop level and proceed automatically to the next level.

The next lower level of detail displays sections as shown in Figure 4.12. The entire loop and each section within the loop are listed in the display. Note that if there are no sections within the selected function or loop, then the user interface will skip the section level when the More Detail or Back button is selected.

The next lower level of detail displays paths as shown in Figure 4.13. Each path is depicted in the main display as a list of blocks and corresponding source

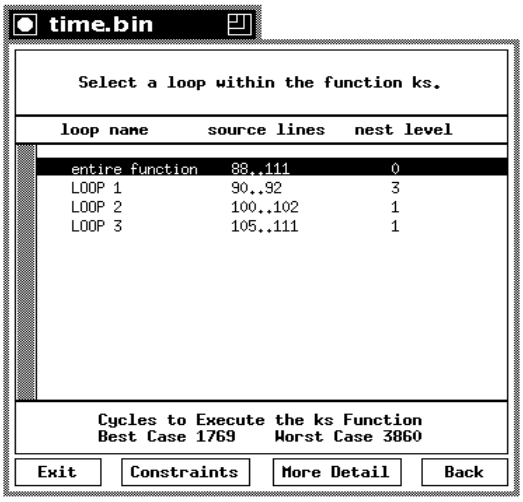


Figure 4.11: Main Window at Loop Level

Figure 4.12: Main Window at Section Level

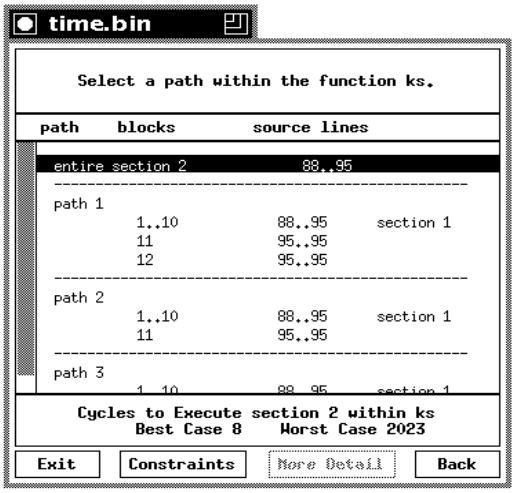


Figure 4.13: Main Window at Path Level

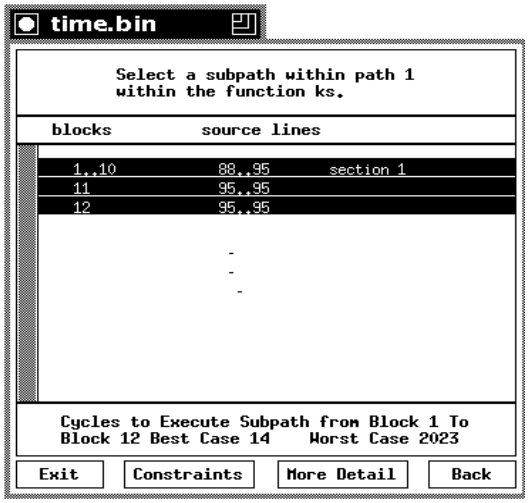


Figure 4.14: Main Window at Subpath Level

line ranges. Note that if a path contains a transition to a header of a more deeply nested loop or section, then the entire child loop or section is represented as a single step along that path.

The next level of detail consists of subpaths as shown in Figure 4.14. A subpath is a subset of the blocks within a path that are connected by control-flow transitions. A subpath is selected by pressing the mouse button with the cursor on the subpath starting block and releasing it on the ending block. The final level of detail consists of machine instructions as shown in Figure 4.15. Only the instructions within the initial and ending block of the subpath are shown.

Thus, there are six levels of detail in the program that the user can view: function, loops, sections, paths, subpaths, and ranges of machine instructions. The loop level or the section level is not shown if there are no loops within the selected function or no sections within the selected loop, respectively.

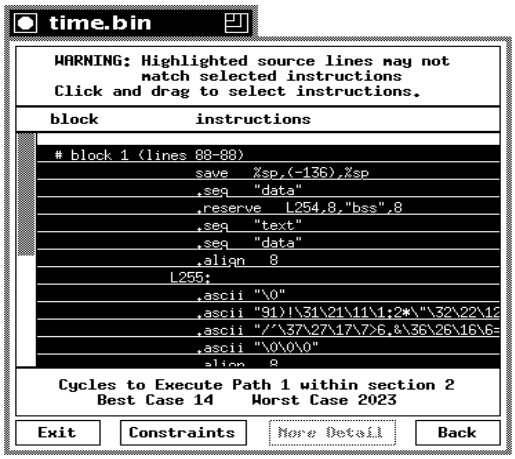


Figure 4.15: Main Window at Instruction Level

Chapter 5
PERFORMANCE

A real-time program with simple control flow will not cause performance problems for the timing analyzer. Therefore, the threshold for the maximum number of paths has been selected to be sixteen, which is a fairly high number of paths. We found that the timing analyzer is responsive when the number of paths at any loop level is less than or equal to sixteen. When the number of paths through a loop is larger than sixteen, then it would be awkward for the user to find a specific path using the main window method in the user interface.

To assess the performance of the section approach in the timing analyzer, four programs with different levels of complexity have been selected. The first test program, *Toy4*, with sixteen paths through *main()* has been chosen to represent the case where no section is needed to be created. It may be the case that most loop levels in a program will have simple control flow and no sections will be needed for these levels. *Toy7* with seven control statments is used to represent a case of moderately complex control flow. *Toy10* and *Toy15* with 10 and 15 control statments are used to test the cases of complicated control flow.

As stated previously, the goal of the section approach is to increase the speed of the timing analyzer and reduce the dynamic space needed for analyzing programs with complex control flow. At the same time this modification to the timing analyzer should not significantly affect the timing predictions results.

Therefore, the dynamically allocated space needed by the timing analyzer, the WCET (Worst Case Execution Time), the BCET (Best Case Execution Time), and the user CPU time spent in the timing analyzer were measured for each test program to assess the performance of the section approach.

Table 5.1 shows the performance results of these metrics when executing the timing analyzer without using the section approach. Table 5.2 shows the value of the same metrics when running the modified timing analyzer for the same test programs. Note that the timing analyzer without the section approach runs out of memory and abortes before finishing the analysis of *Toy15*. Thus, the reported memory allocated for *Toy15* is the amount before the program aborted. The user CPU time spent in the timing analyzer to analyze *Toy4*, a program with simple control flow, is the same before and after sections since no sections were required to be created. It appears that the CPU time is correlated to the number of paths. Thus, as number of paths grow, the complexity and CPU time both increase exponentially. Partitioning the control flow into sections effectively reduce the timing analysis complexity.

The use of the section approach slightly affected the accuracy of the timing prediction results. A slightly less accurate timing analyzer seems to be a good compromise for a faster timing analyzer that is capable of analyzing programs with complex control flow.

	Memory allocated KB	CPU user time sec	WCET cycle	BCET cycle
Toy4	116.4	1.0	43	27
Toy7	856.4	15.8	61	33
Toy10	7,945.4	146.3	85	43
Toy15	66,851.6	NA	NO results	NO results

Table 5.1: Timing Analyzer Performance before Modification

	Memory allocated KB	CPU user time sec	WCET cycle	BCET cycle
Toy4	120.9	0.9	43	27
Toy7	223.8	1.8	61	31
Toy10	357.1	3.8	86	40
Toy15	1,202.3	4.1	107	45

Table 5.2: Timing Analyzer Performance after Modification

Chapter 6
CONCLUSION

This document has described a technique to simplify complex control flow by partitioning loops and functions into sections. First, the loops and the functions with a number of paths exceeding a predefined threshold are detected. Second, sections are created for these loops and functions to reduce the number of paths at those nesting levels. All the sections within the same loop start from the loop header and have a subset of the loop's paths. The timing analysis tree is updated to include the sections as direct descendants to the loop for which they were created. Third, a new level of detail to be viewed by the user has been added to the already existing five levels of the user interface. The user is also now not forced to view the loop or section level of detail if there are no loops or sections within the function or the loop.

Updating the timing tools to create sections when needed significantly decreased the execution time of the timing analyzer and reduced the dynamically allocated space needed by the timing analyzer. Furthermore, the user can use the user interface to search through any number of paths and obtain timing prediction results in more convenient manner.