Timing Constraint Specification and Analysis*

Lo Ko, Nagham Al-Yaqoubi, Christopher Healy,
Emily Ratliff, Robert Arnold, and David Whalley
Computer Science Department
Florida State University
Tallahassee, FL 32306-4530
e-mail: whalley@cs.fsu.edu
phone: (850) 644-3506

Marion Harmon
Computer and Information Systems Department
Florida A&M University
Tallahassee, FL 32307-3101
e-mail: harmon@cis.famu.edu
phone: (850) 599-3042

Abstract

Real-time programmers have to deal with the problem of relating timing constraints associated with source code to sequences of machine instructions. This paper describes an environment to assist users in the specification and analysis of timing constraints. A timing analyzer predicts the best and worst case bounds for these constrained portions of code. A user interface for this timing analyzer was developed to depict whether these constraints were violated or met. A user is allowed to specify timing constraints within the source code of a C program. The user interface also provides three different methods for interactively selecting portions of programs. After each selection the corresponding bounded times, source code lines, and machine instructions are automatically displayed. Users are prevented from only selecting portions of the program for which timing bounds cannot be obtained. In addition, a technique is presented that allows the timing analysis to scale efficiently with complex functions and loops. The result is a user-friendly environment that supports the user specification and analysis of timing constraints at a high (source code) level and retains the accuracy of low (machine code) level analysis.

1. Introduction

There are advantages for performing a static timing analysis of a program. Accurate static analysis provides a greater level of assurance that timing constraints will be met. There is always the danger that the test case that drives the best or worst-case times will not be applied with dynamic observations. Determining the best and worst-case test data is particularly difficult with contemporary architectural features, such as a cache. Static analysis can also be performed even before the actual machine is available on which the real-time application will execute.

One controversial aspect of real-time systems is whether timing analysis should be performed at a high (source code) or low (machine code) level. An advantage of high-level analysis is that the results of the timing predictions can be easily related to a user. Timing bounds are obtained for each high-level language construct, which includes statements, loops, and functions. The assumption is that timing bounds for a specific machine can be associated with each of these constructs. Unfortunately, current architectural features, such as pipelines and caches, preclude a single a priori time associated with a high-level language construct. In addition, global compiler optimizations can affect how a specific construct is translated and its associated timing behavior. While much more accurate timing bounds can be obtained by performing the analysis at the machine code level, it is still important to relate these timing predictions in a manner that a user can understand. A user needs to know the correspondence between sequences of machine instructions

*This work was supported in part by the Office of Naval Research under contract number N00014-94-1-0006 and the National Science Foundation under the cooperative agreement HRD-9707076. Preliminary versions of the viewer were described in [1], [2]. Lo Ko is currently working for the Florida Department of Insurance in Tallahassee. Nagham Al-Yaqoubi is in Boston working for Raytheon. Emily Ratliff works for IBM in Fort Lauderdale. Robert Arnold is currently working for Peek Traffic Transyt in Tallahassee.

and lines of source code.

This problem is similar to the one of symbolic debugging of optimized code. Many users are willing to rely on symbolic debugging of unoptimized code given that the behavior of the unoptimized and optimized programs are semantically equivalent. However, correct behavior of real-time programs demands that the results are produced on time. Thus, the timing analysis should be at the level of the optimized machine instructions or the compiler should maintain an accurate mapping between the high-level and low-level representations. There has been much research in the area of symbolic debugging of optimized code to maintain such mappings [3], [4], [5], [6].

This paper describes an environment to support the specification and analysis of timing constraints. The environment allows specification of constraints at the source code level, performs the timing analysis at the machine code level, and provides a graphical display of the relationship between the machine instructions (i.e. assembly code) and the corresponding source code. The timing analysis is performed for the MicroSPARC I processor [7]. Other papers are available for readers interested in how the timing predictions are actually obtained [8], [9].

2. Overview

The design of the environment described in this paper had the following goals:

(1) A user should be able to quickly specify constraints and obtain timing predictions for the specified portions of a program.

(2) The user should only be allowed to select portions of the program for which timing bounds can be obtained.

(3) The ability to specify constraints and obtain timing predictions should not inhibit compiler optimizations from being performed.

(4) The correspondence between source code and machine code of the program selected by the user for timing prediction should be graphically depicted.

Figure 1 gives an overview of the context in which timing predictions are obtained. Control-flow information, which includes timing constraint specifications, is stored as a side effect of the compilation of a file. This control-flow information is passed to a static cache simulator, which constructs the control-flow graph of the program that consists of the call graph and the control flow of each function [10]. 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 [11]. Next, a timing analyzer uses the instruction caching categorizations, machine-dependent information describing the pipeline, and the control-flow information provided by the compiler, which includes the source lines associated with each basic block, to make best-case and worst-case performance predictions. A timing analysis tree is constructed to simplify the process of predicting best and worst-case times. Each node within the tree is considered a natural loop. The outer level of each function instance is treated as a loop that will iterate only once when entered. A best and worst-case time is predicted for each function and loop within the program. The user is prompted for the minimum and maximum loop iterations of loops when it could not be calculated by the compiler. Finally, a

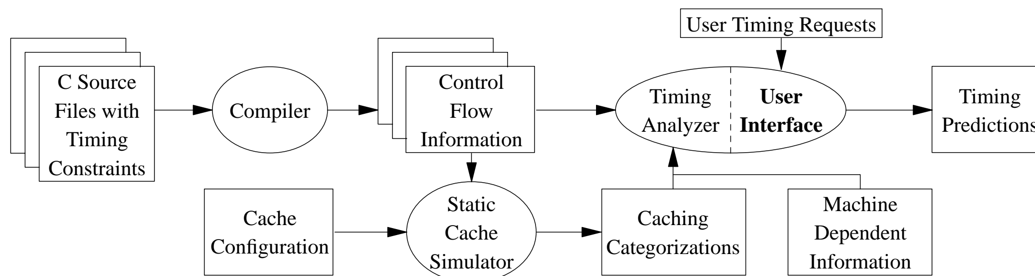

Figure 1: Overview of Obtaining Timing Predictions

graphical user interface (GUI) is invoked that allows the user to request the status of the constrained portions and timing predictions for other specified parts of the program.

The timing analyzer determines the set of possible paths through each loop. A path is a sequence of unique blocks in the loop connected by control-flow transitions. Each path corresponds to a possible sequence of blocks that could be executed during a single loop iteration. If a path within a loop enters a child loop, then the entire child loop is represented as a single block along that path.

Sometimes the control flow within a loop is too complicated to efficiently represent and analyze all possible paths. For instance, consider the function in Figure 2(a), which contains three loops. Figure 2(b) depicts the timing tree that would be constructed for this program. Figure 2(c) shows the control flow for *Loop 1*, which contains n consecutive **if** statements. For each iteration of the loop, there are 2^n possible paths that can be taken since each of the n **if** statements can either be entered or not entered. If there are 20 such **if** statements, then there would be 2^{20} paths, which could not be represented and analyzed on most machines. The timing analyzer was modified to partition the control flow of complex loops and functions into *sections* that are limited to a predefined number of paths. The timing tree is also updated to include each section as a direct descendant to the loop for which it was created as shown in Figure 2(d). Each

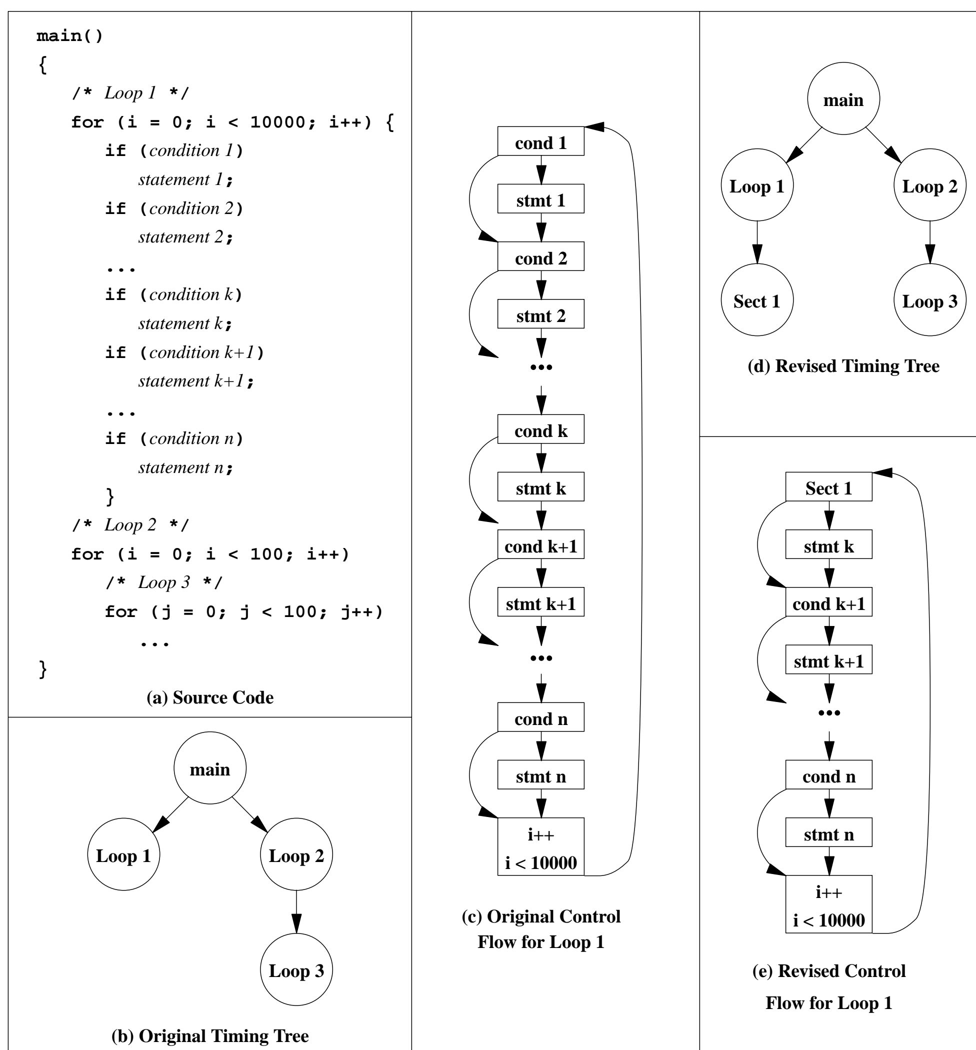

Figure 2: Example Illustrating the Use of Sections

section is treated by the timing analyzer as a loop that iterates only once. Figure 2(e) illustrates the revised control flow for *Loop 1*, where the first *k* conditions and first *k*-1 statements are partitioned into a separate section. With a limit of 16 paths for each loop or function, we found that it was rarely necessary to partition the control flow into sections. Even when partitioning was necessary, there was little or no impact on the accuracy of the timing analysis [12].

3. Related Work

There has been much work on proposing real-time language constructs to express timing constraints. Many authors have added real-time constructs to existing languages, such as C [13], [14], C++ [15], [16], and Euclid [17]. In addition, many new real-time languages with features for expressing timing constraints have been proposed and implemented [18], [19], [20]. There has also been some work on performing code transformations in an attempt to ensure that timing constraints are not violated [20]. Finally, expert systems with friendly user interfaces have also been developed to provide scheduling advice for developers of real-time system applications [21].

Unfortunately, there has been little work in the area of providing support for user analysis of timing constraints. First, a user needs to know which timing constraints could be violated and the extent of the violations. Furthermore, a user may wish to know which timing constraints are close to being potentially violated. Finally, one may wish to obtain some insight as to why a timing constraint may be violated. This insight can aid the user in tuning or rewriting the constrained code to satisfy the constraint. In addition to providing the ability to specify timing constraints, the environment described in this paper also provides GUI support for user-friendly analysis of these constraints.

4. Specification of Timing Constraints

Real-time programs may often have timing constraints on portions of source code, which are sometimes referred to as critical sections. It is desirable to have these timing constraints expressed within the source code and be automatically checked as programs are being developed and later maintained. Ideally, the timing analysis could occur each time the program is linked and the user can be informed of any potential timing constraint violations. In addition, the user may wish to monitor the constrained sections of code to determine how close the predicted worst-case execution time is to violating a timing constraint. Recognizing potential timing constraint violations earlier in the development process can save both time and money. Finally, the ability to obtain timing predictions on constrained code portions should not inhibit the optimizations performed by a compiler.

The ability to capture these constraints was accomplished by modifying the front end of a C compiler called *vpcc* [22]. This constraint information was passed through the back end of a C compiler called *vpo* [23]. Source lines associated with basic blocks are tracked while performing the optimizations in *vpo*. As shown in Figure 1, the back end conveys the constraint information along with the correspondence between source lines and assembly code to the timing analyzer in the control-flow information.

The environment described in this paper allows users to specify timing constraints in the source code on functions, loops, and paths. Figure 3 depicts the three types of constraints that can be specified. The code within this figure contains a function that calculates the sum and count of the nonnegative and negative values of a matrix. The function is constrained to no more than 2 milliseconds. A best-case constraint for the function was not specified. The inner loop within the function has a best-case constraint of 500 nanoseconds and a worst-case constraint of 3 microseconds. A path is specified by annotating source lines, which must be contained within the path. If a source line contains an invocation of a function, then the time required to execute that function (and any other functions that could be invoked from it) is included when the timing analyzer determines if the constraint was satisfied. These annotations are of the form **@*n***, where ***n*** is the path identifier. The annotations require only a few characters to facilitate their placement on the source lines being specified. One of the annotations within a path must have a best and/or worst-case constraint. There are two overlapping paths within the inner loop that have constraints. Path 1 goes through source lines 13 and 15 and path 2 goes through lines 13, 18, and 19. Thus, this simple method of specifying paths is quite flexible. For instance, overlapping paths can be easily specified.

```
 1 functimebnd [:2ms]
 2 void Sum(Array, Nonnegcnt, Negcnt, nonnegsum, Negsum)
 3 matrix Array;
 4 int *Nonnegcnt, *Negcnt, *Nonnegsum, *Negsum;
 5 {
 6    int i, j;
 7    void Addnonneg(), Addneg();
 8
 9    *Nonnegsum = *Negsum = *Nonnegcnt = *Negcnt = 0;
10    for (i=1; i <= MAXSIZE; i++)
11       looptimebnd [500ns:3us]
12       for (j=1; j <= MAXSIZE; j++)
13          if (Array[i][j] >= 0) {      @1[:150ns] @2[10ns:100ns]
14             Addnonneg(Array[i][j], Nonnegsum);
15             (*Nonnegcnt)++;           @1
16             }
17          else {
18             *Negsum += Array[i][j];  @2
19             (*Negcnt)++;             @2
20             }
21 }
```

Figure 3: Source Code with Timing Constraints

5. User Interface

The user interface is invoked after the timing analyzer has analyzed the entire program. Figures 4 and 5 depict the three windows that are always displayed for the timing analysis graphical user interface. Figure 4 shows the main window of the user interface. The top section of the main window displays a message

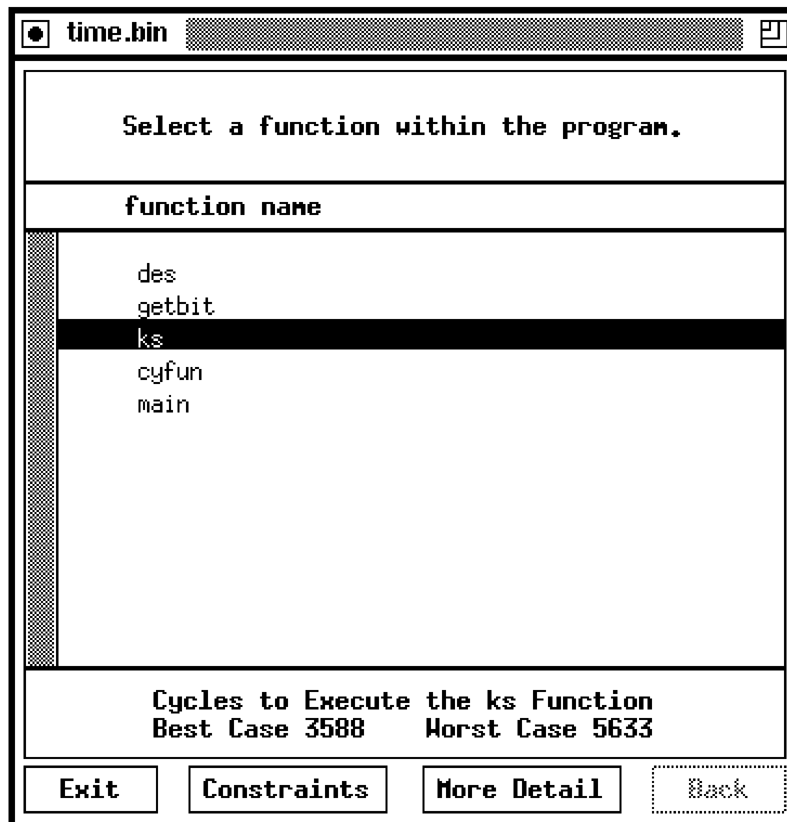

Figure 4: Main Window at Function Level

indicating the current action the user can perform with a mouse selection 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. Portions of the middle section of the window associated with other program constructs can be selected by simply clicking on the appropriate line. 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 coarser level of detail. Examples of selecting these two options will be given shortly. The **Exit** button can always be selected to allow the user to exit the application at any time. The **Constraints** button is clicked when the user wishes to view the timing predictions associated with the source code specified constraints.

Figure 5 shows the two other windows in the user interface that are always displayed. The left window contains a display of the source code of the program being analyzed. The highlighted lines are the executable source lines that correspond to the highlighted construct in the middle section of the main window. Whenever a different construct is selected in the main window, the highlighted lines in the source and assembly windows are automatically updated and scrolled to the appropriate position.

Note that the source lines within the display are numbered. This allows a user to identify constructs that are referenced by line numbers in the main window. The options at the bottom of the source window will

```
C Source Code of des.c
line # source code
 23        49,17,57,25};
 24     static great kns[17];
 25     static int initflag=1;
 26     int ii,i,j,k;
 27     unsigned long ic,shifter,getbit();
 28     immense itmp;
 29     void cyfun(), ks();
 30
 31     if (initflag) {
 32        initflag=0;
 33        bit[1]=shifter=1L;
 34        for(j=2;j<=32;j++) bit[j] = (shifter <<= 1);
 35        }
 36     if (*newkey) {
 37        *newkey=0;
 38        for(i=1;i<=16;i++) ks(key, i, &kns[i]);
 39        }
 40     itmp.r=itmp.l=0L;
 41     for (j=32,k=64;j>=1;j--,k--) {
 42        itmp.r = (itmp.r <<= 1) | getbit(inp,ip[j],32);
 43        itmp.l = (itmp.l <<= 1) | getbit(inp,ip[k],32);
 44        }
 45     for (i=1;i<=16;i++) {
 46        ii = (isw == 1 ? 17-i : i);
 47        cyfun(itmp.l, kns[ii], &ic);
 48        ic ^= itmp.r;
 49        itmp.r=itmp.l;
 50        itmp.l=ic;
 51        }
 52     ic=itmp.r;
 53     itmp.r=itmp.l;
 54     itmp.l=ic;
 55     (*out).r=(*out).l=0L;
 56     for (j=32,k=64; j >= 1; j--, k--) {
 57        (*out).r = ((*out).r <<= 1) | getbit(itmp,ipm[j],32);
 58        (*out).l = ((*out).l <<= 1) | getbit(itmp,ipm[k],32);
 59        }
 60  }
Select Path   Accept   Cancel   Clear All
```

```
Assembly Code of des.s
blk   assembly code
      sll    %o4,1,%o4
# block 5 (lines 36-36)
  L219:
      ld    [%o5],%o0
      cmp   %o0,%g0
      be,a   L224
      st    %g0,[%sp + .1_itmp]
# block 6 (lines 37-38)
      st    %g0,[%o5]
      mov   1,%l0
      add   %sp,.0_STARG,%l4
      sethi   %hi(L214),%l6
      add   %l6,%lo(L214),%l3
      add   %l3,12,%l6
      add   %l3,192,%i2
# block 7 (lines 38-38)
  L227:
      ld    [%i1 + 4],%o1
      st    %o1,[%sp + (.0_STARG + 4)]
      ld    [%i1],%o0
      st    %o0,[%sp + .0_STARG]
      mov   %l4,%o0
      mov   %l0,%o1
      call   _ks,3
      mov   %l6,%o2
# block 8 (lines 38-38)
      add   %l6,12,%l6
      cmp   %l6,%i2
      ble   L227
      add   %l0,1,%l0
# block 9 (lines 40-40)
      st    %g0,[%sp + .1_itmp]
# block 10 (lines 40-41)
  L224:
      mov   32,%i1
      cmp   %i1,1
      bl    L230
      st    %g0,[%sp + (.1_itmp + 4)]
Best Pipeline Dia.   Worst Pipeline Dia.
```

Figure 5: Source Code and Assembly Code Windows

be explained later in the paper. The right window contains a display of the assembly code for the program. The highlighted assembly lines correspond to the code generated for the highlighted source lines. Note that a comment precedes each basic block that identifies the block number and the associated source lines. These comments in the assembly window and the line numbers in the source window allow a user to quickly grasp the relationship between the high-level (source code) and low-level (machine code) representations.

Figure 5 also illustrates a pitfall a user may face with the tool. Source code lines are only tracked to a basic block level. Remember that one goal of the environment was to not inhibit any compiler optimizations from being performed. Sometimes optimizations move individual instructions from one basic block to another. For instance, the last instruction in block 5 of Figure 5 corresponds to the assignment of zero to `itmp.1` at line 40 in the source code. This instruction was copied from block 10 into block 5 when filling the delay slot for the preceding branch. The user has the responsibility to ensure that the selected source lines correspond to the assembly instructions that are examined by the timing analyzer.

The timing analyzer constructs a tree to simplify the process of bounding the timing performance of a program. Each node in the tree corresponds to an instance of a function, natural loop, or section. Functions and sections are analyzed as though they were natural loops that iterate only once when entered. The nodes in the timing analysis tree are processed in a bottom-up fashion [8]. The entire tree is analyzed before a user is allowed to request timing predictions.

The most straightforward approach for allowing one to obtain timing predictions from various portions of the program would be to allow the user to move up or down a single node of the timing tree at a time. The authors perceived that most users would not be interested in traversing a graph representing the combined call graph and loop nesting structure of the program. Instead, users would most likely want the capability of accessing specified portions of the program as quickly as possible. The user interface described in this paper provides three different methods for quickly accessing portions of a program. The first method quickly allows a user to check if each of the specified timing constraints were met or violated. The second method uses a hierarchical approach that provides a very fine level of selection of program portions. The final method permits quick and convenient selection of source code portions for timing predictions.

5.1. Selecting Portions of a Program Using the Constraints Window

The first method for accessing portions of the program involves using the constraints window after clicking the **Constraints** button in the main window. The different portions of the program that can be accessed are the portions specified in the source code timing constraints. Figure 6 shows the constraint window, which contains a scrollable display of the user-specified constraints. A user may choose to have

User Specified Timing Constraints

Num	Best Case Predicted	Best Case Specified	Worst Case Specified	Worst Case Predicted	Function Name	Type	Source Lines
1	1205	110	1010	3865*	main	func	9..37
2	63*	400	500	513*		loop	23..23
3	822	100	1000	2066*		loop	9..15
4	263	10	750	397		loop	26..27
5	272	10	751	397		loop	31..31
6	272	10	752	397		loop	36..36
7	no path	25		no path		path	10, 11, 12, 14
8	20		100	57		path	10, 15

Dismiss

Figure 6: Constraints Window

the source and assembly windows display the code associated with a constraint by simply clicking on the appropriate line within the scrollable section. At that point the associated code portion will be highlighted and scrolled to the appropriate position in both the source and assembly windows. The constraint window will remain until it is dismissed. Thus, the user can quickly view the code associated with a number of constraints by simply clicking on a sequence of lines.

For each constraint the window displays the specified and predicted best and worst-case times in clock cycles[1] and the location of the constrained source code. If the user did not specify a best or worst-case time in the timing constraint, then the corresponding field in the display is left blank. If the best-case predicted time is less than the specified best-case timing constraint, then an asterisk follows the predicted time to indicate that the constraint has been violated. Likewise, an asterisk will follow the worst-case predicted time if it exceeds its corresponding worst-case specified time. It is possible that a user may select a set of lines that cannot be executed in a single path (as in constraint 7 of Figure 6), such as the `then` and `else` portions of an `if-then-else` construct. Constraint number 7 in Figure 6 illustrates such a selection and the display indicates that no path can be associated with the selected source lines.

5.2. Selecting Portions of a Program Using the Main Window

The second method for accessing different portions of the program involves clicking the **More Detail** button after selecting the appropriate 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 top level is depicted in Figure 4, which was discussed earlier in the paper. The function selected by default upon initialization of the interface is the `main` function, which results in displaying the best and worst-case clock cycles representing the execution of the entire program. The remaining five levels are shown in Figures 7 through 11.

The next lower level of detail consists of loops as shown in Figure 7. Selection of a function, loop, section, path, subpath, or range of instructions will cause the corresponding bounded prediction of cycles to be displayed and the appropriate lines to be highlighted in the other two windows. The loops displayed are the loops within the selected function. Note that by each loop number is its range of source lines and nesting level of the loop within the function to facilitate identification by the user. Sections are displayed after loops as shown in Figure 8. The sections displayed are the sections that are automatically created to simplify the control flow within the selected function or loop. If a function contains no loops, then the loop level will be automatically skipped when the **More Detail** is selected. Likewise, the section level will be skipped if a loop contains no sections. Paths, as shown in Figure 9, are the level displayed after sections. A path is defined as a unique sequence of basic blocks connected by control-flow transitions. Each loop path starts with the loop header and is terminated by a block with a transition to the loop header or to an exit block outside the loop. The paths at a function level start with the initial block in the function and are terminated by blocks containing return instructions. Thus, each path is depicted in the main window as a list of blocks and corresponding ranges of source lines. Note that if a path contains a transition to a header of a more deeply nested loop, then the entire child loop is represented as a single step along that path. A maximum of 16 paths is allowed at any given function or loop. Otherwise, the timing analyzer would create a section to reduce the number of paths at that level. The subpath level is depicted in Figure 10. 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 11. Only the instructions within the initial and ending block of the subpath are shown. The user selects a beginning instruction from the initial block by holding down the mouse button and selects an ending instruction from the last block by releasing the button. Hence, the user is allowed to obtain a very fine-grain level of timing predictions.

[1] These specified and predicted times are given in clock cycles as opposed to a time unit (e.g. microseconds). A later section of the paper will describe how the environment supports detailed pipeline analysis of the code portions. This analysis is easily accomplished by presenting performance information based on cycles.

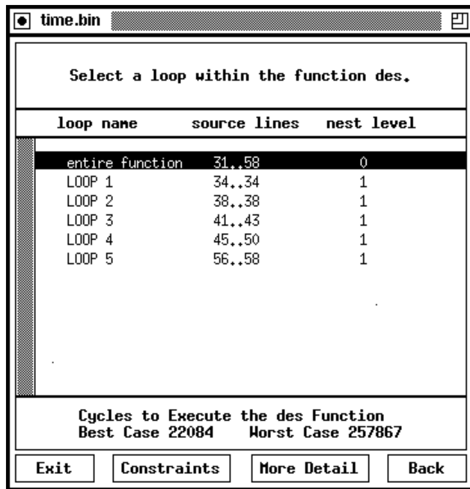

Figure 7: Main Window at Loop Level

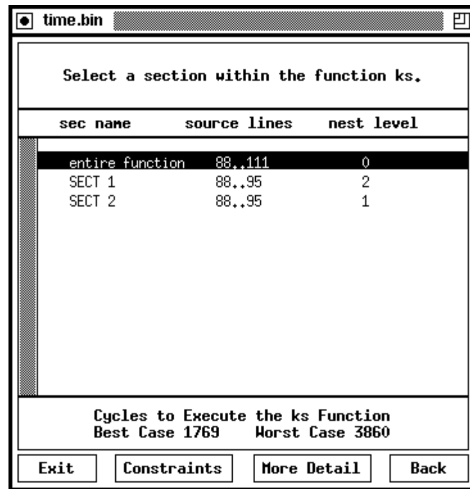

Figure 8: Main Window at Section Level

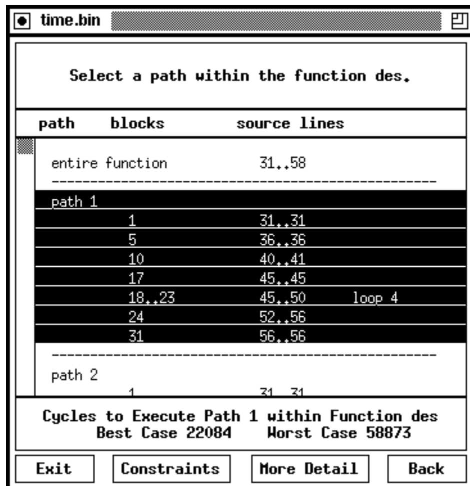

Figure 9: Main Window at Path Level

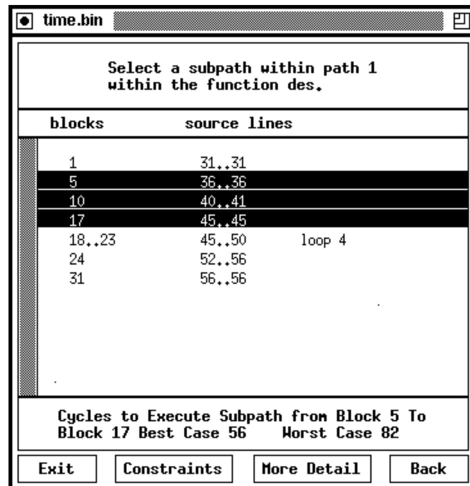

Figure 10: Main Window at Subpath Level

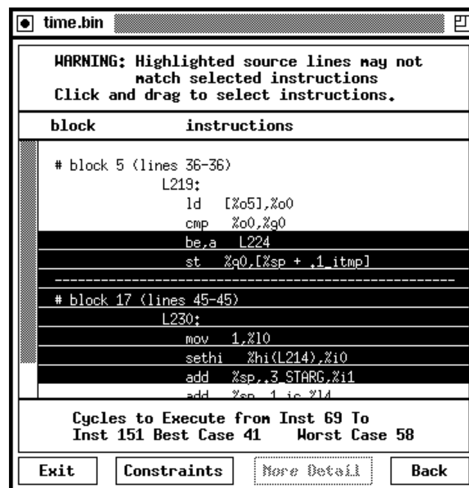

Figure 11: Main Window at Assembly Level

Note that a basic block is the finest level of detail for which timing predictions may be obtained for associated source code selections. The source window is not updated when a range of instructions within a subpath is selected since source code lines are only tracked to the basic block level. Note that while it was not difficult to provide timing bounds at a very fine-grain level by allowing selection of one specific instruction within each of the first and last blocks in the subpath, a corresponding set of source lines would be more difficult to identify. It is possible that a basic block may be associated with multiple source lines or a single source line may be associated with multiple basic blocks.

It was relatively simple to track the source lines associated with basic blocks during compiler optimizations. During the compilation, the source lines for each C statement are identified by the front end of the compiler and are passed to the back end, which associates a range of source lines with each basic block. These ranges of source lines are then maintained whenever basic blocks were moved (i.e. the control flow is adjusted) during optimizations. Note that all the instructions within a specific basic block may not correspond to the associated lines of source code. Various global optimizations may move individual instructions to different blocks (e.g. code motion, filling delay slots, etc.). For instance, the delay slot of the branch in block 5 (source lines 36-36) of Figure 5 was filled with an instruction from block 10 (source lines 40-41). The authors have found that the optimizations performed by our compiler still allow a user to easily understand the correspondence between source code and the generated assembly. Compiler optimizations that perform a greater level of restructuring may make this task for the user more difficult.

The main window accommodates six levels of detail in a program that a user can view: functions, loops, sections, paths, subpaths, and ranges of machine instructions. Thus, at most six selections in the main window are required for a user to quickly choose any specifiable portion of the program. The appropriate timing analysis information is extracted for each user selection. If there is more than one instance of the user selected portion (i.e. multiple instances can occur when the portion of source code can be reached via different sequences of calls), then the fastest of the best-case times and the slowest of the worst-case times of the different instances are displayed.

5.3. Selecting Portions of a Program Directly from the Source Window

The last method for accessing a portion of the program is to select lines of source code directly by using the mouse as depicted in Figure 12. First, the user clicks on the **Select Path** button at the bottom of the source code window. Next, the user highlights the source lines within the path to be timed. This highlighting is accomplished by clicking the left mouse button on the desired source lines as shown in Figure 12. A user may quickly obtain the best-case and worst-case timing predictions for a segment of code by selecting only two source lines, which would indicate the start and the end of the path. The user can clear a specific highlighted line by clicking the right mouse button on that line. The user can also clear all the highlighted lines selected so far by clicking the **Clear All** button. Finally, the user can also cancel the selection of a path by clicking the **Cancel** button.

Once the user has highlighted the source lines of interest, then the timing bounds can be obtained by clicking on the **Accept** button. At this point a popup window is displayed that allows the user to select the best or worst-case path or indicates that no path exists that executes instructions from every selected source line. In addition, the user can select to view the loop or function that most tightly encloses the highlighted lines.

Figures 13 and 14 show the best and worst case set of source lines, respectively, that would be displayed associated with the source lines selected in Figure 12. In contrast to the best case path, both `if` statements are entered in the worst-case path. Note that instructions associated with other source lines may have to be executed as well even in the best case. The basic block associated with source line 36 has to be executed to be able to reach line 40 from line 31. Likewise, other lines may have to be executed since their corresponding machine instructions are in a selected basic block. For instance, the initialization of the for loop at line 41 is in the same basic block as the assignment statement at line 40.

Automatically including line 41 in this example illustrates that the user is restricted to only selecting portions of the program for which timing predictions can be obtained. The timing analyzer only tracks source lines to a basic block level. Thus, it must include all source lines associated with a basic block if any source lines in that block are selected.

```
C Source Code of des.c
line # source code
15       32,24,16,8,57,49,41,33,25,17,9,1,59,51,43,35,
16       27,19,11,3,61,53,45,37,29,21,13,5,63,55,47,39,
17       31,23,15,7};
18    static char ipm[65]=
19       {0,40,8,48,16,56,24,64,32,39,7,47,15,
20       55,23,63,31,38,6,46,14,54,22,62,30,37,5,45,13,
21       53,21,61,29,36,4,44,12,52,20,60,28,35,3,43,11,
22       51,19,59,27,34,2,42,10,50,18,58,26,33,1,41,9,
23       49,17,57,25};
24    static great kns[17];
25    static int initflag=1;
26    int ii,i,j,k;
27    unsigned long ic,shifter,getbit();
28    immense itmp;
29    void cyfun(), ks();
30
31    if (initflag) {
32       initflag=0;
33       bit[1]=shifter=1L;
34       for(j=2;j<=32;j++) bit[j] = (shifter <<= 1);
35       }
36    if (*newkey) {
37       *newkey=0;
38       for(i=1;i<=16;i++) ks(key, i, &kns[i]);
39       }
40    itmp.r=itmp.l=0L;
41    for (j=32,k=64;j>=1;j--,k--) {
42       itmp.r = (itmp.r <<= 1) | getbit(inp,ip[j],32);
43       itmp.l = (itmp.l <<= 1) | getbit(inp,ip[k],32);
44       }
45    for (i=1;i<=16;i++) {
46       ii = (isw == 1 ? 17-i : i);
47       cyfun(itmp.l, kns[ii], &ic);
48       ic ^= itmp.r;
49       itmp.r=itmp.l;
50       itmp.l=ic;
51       }
52    ic=itmp.r;
Select Path   Accept   Cancel   Clear All
```

Figure 12: Selecting a Path via the Source Code

```
C Source Code of des.c
line # source code
18    static char ipm[65]=
19       {0,40,8,48,16,56,24,64,32,39,7,47,15,
20       55,23,63,31,38,6,46,14,54,22,62,30,37,5,45,13,
21       53,21,61,29,36,4,44,12,52,20,60,28,35,3,43,11,
22       51,19,59,27,34,2,42,10,50,18,58,26,33,1,41,9,
23       49,17,57,25};
24    static great kns[17];
25    static int initflag=1;
26    int ii,i,j,k;
27    unsigned long ic,shifter,getbit();
28    immense itmp;
29    void cyfun(), ks();
30
31    if (initflag) {
32       initflag=0;
33       bit[1]=shifter=1L;
34       for(j=2;j<=32;j++) bit[j] = (shifter <<= 1);
35       }
36    if (*newkey) {
37       *newkey=0;
38       for(i=1;i<=16;i++) ks(key, i, &kns[i]);
39       }
40    itmp.r=itmp.l=0L;
41    for (j=32,k=64;j>=1;j--,k--) {
42       itmp.r = (itmp.r <<= 1) | getbit(inp,ip[j],32);
43       itmp.l = (itmp.l <<= 1) | getbit(inp,ip[k],32);
44       }
45    for (i=1;i<=16;i++) {
46       ii = (isw == 1 ? 17-i : i);
47       cyfun(itmp.l, kns[ii], &ic);
48       ic ^= itmp.r;
49       itmp.r=itmp.l;
50       itmp.l=ic;
51       }
52    ic=itmp.r;
53    itmp.r=itmp.l;
54    itmp.l=ic;
55    (*out).r=(*out).l=0L;
Select Path   Accept   Cancel   Clear All
```

Figure 13: Best Case Path from Source Lines Selected in Figure 12

```
C Source Code of des.c
line # source code
18    static char ipm[65]=
19       {0,40,8,48,16,56,24,64,32,39,7,47,15,
20       55,23,63,31,38,6,46,14,54,22,62,30,37,5,45,13,
21       53,21,61,29,36,4,44,12,52,20,60,28,35,3,43,11,
22       51,19,59,27,34,2,42,10,50,18,58,26,33,1,41,9,
23       49,17,57,25};
24    static great kns[17];
25    static int initflag=1;
26    int ii,i,j,k;
27    unsigned long ic,shifter,getbit();
28    immense itmp;
29    void cyfun(), ks();
30
31    if (initflag) {
32       initflag=0;
33       bit[1]=shifter=1L;
34       for(j=2;j<=32;j++) bit[j] = (shifter <<= 1);
35       }
36    if (*newkey) {
37       *newkey=0;
38       for(i=1;i<=16;i++) ks(key, i, &kns[i]);
39       }
40    itmp.r=itmp.l=0L;
41    for (j=32,k=64;j>=1;j--,k--) {
42       itmp.r = (itmp.r <<= 1) | getbit(inp,ip[j],32);
43       itmp.l = (itmp.l <<= 1) | getbit(inp,ip[k],32);
44       }
45    for (i=1;i<=16;i++) {
46       ii = (isw == 1 ? 17-i : i);
47       cyfun(itmp.l, kns[ii], &ic);
48       ic ^= itmp.r;
49       itmp.r=itmp.l;
50       itmp.l=ic;
51       }
52    ic=itmp.r;
53    itmp.r=itmp.l;
54    itmp.l=ic;
55    (*out).r=(*out).l=0L;
Select Path   Accept   Cancel   Clear All
```

Figure 14: Worst Case Path from Source Lines Selected in Figure 12

```
C Source Code of des.c
line # source code
48          ic ^= itmp.r;
49          itmp.r=itmp.l;
50          itmp.l=ic;
51          }
52       ic=itmp.r;
53       itmp.r=itmp.l;
54       itmp.l=ic;
55       (*out).r=(*out).l=0L;
56       for (j=32,k=64; j >= 1; j--, k--) {
57          (*out).r = ((*out).r <<= 1) | getbit(itmp,ipm[j],32);
58          (*out).l = ((*out).l <<= 1) | getbit(itmp,ipm[k],32);
59          }
60    }
61
62    unsigned long getbit(source,bitno,nbits)
63    immense source;
64    int bitno,nbits;
65    {
66       if (bitno <= nbits)
67          return bit[bitno] & source.r ? 1L : 0L;
68       else
69          return bit[bitno-nbits] & source.l ? 1L : 0L;
70    }
71
72    void ks(key,n,kn)
73    immense key;
74    great *kn;
75    int n;
76    {
77       static immense icd;
78       static char ipc1[57]={0,57,49,41,33,25,17,9,1,58,50,
79          42,34,26,18,10,2,59,51,43,35,27,19,11,3,60,
80          52,44,36,63,55,47,39,31,23,15,7,62,54,46,38,
81          30,22,14,6,61,53,45,37,29,21,13,5,28,20,12,4};
82       static char ipc2[49]={0,14,17,11,24,1,5,3,28,15,6,21,
83          10,23,19,12,4,26,8,16,7,27,20,13,2,41,52,31,
84          37,47,55,30,40,51,45,33,48,44,49,39,56,34,
85          53,46,42,50,36,29,32};
Select Path   Accept   Cancel   Clear All
```

Figure 15: Selecting an Infeasible Path

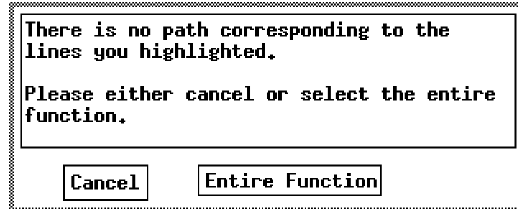

Figure 16: Popup Window after Selecting an Infeasible Path

Figure 15 contains a source window selection of an `if-then-else` construct. Figure 16 shows the popup window that is displayed when no path can be associated with the selected source lines. As the popup indicates in Figure 16, there is no single path that can execute both the `then` and `else` portions. Note in this case the user is given the option of selecting the entire function, which immediately encloses the selected source lines.

Figure 17 shows another selection by the user. Notice that the user has selected lines partially within a loop and outside of the loop. This selection illustrates another restriction imposed by the timing analyzer. Any path that contains a loop along with code outside a loop is assumed to execute the entire loop. Figure 18 shows that additional highlighted lines are included after the path has been selected. Lines 166-168 are automatically included since the entire loop is assumed to have been executed. As illustrated previously in Figures 12 through 14, only selections that correspond to entire basic blocks are allowed. Lines 172-176 are included since its corresponding instructions are in the same basic block as the instructions associated with line 171.

5.4. Supporting Detailed Analysis of Timing Constraints

The user interface can also be used to display information to the user about how the timing prediction was obtained. This information may aid a user in rewriting constrained code to satisfy a violated constraint. The user can select buttons at the bottom of the assembly window shown in Figure 5 to obtain a pipeline diagram of the best and worst-case performance of a path containing no loops or calls. Figure 19 shows the pipeline diagrams for both the best and worst-case performance associated with a path through a loop. In contrast to Figure 5, the assembly window has been redrawn to include numbers with each assembly instruction. These numbers are referenced in the scrollable pipeline diagrams to indicate when each instruction enters a given stage of the pipeline. The source code window is also covered by the pipeline diagram windows since the user may confuse the source line numbers with the instruction numbers in the pipeline diagram. Pipeline diagrams are useful since a user may wish to understand why a sequence of instructions required a given number of cycles. For instance, a user can determine that load stalls occurred

```
C Source Code of des.c
line # source code
153       0,9,0,4,12,0,7,10,0,0,5,9,11,10,9,11,15,14,
154       0,10,3,10,2,3,13,5,3,0,0,5,5,7,4,0,2,5,
155       0,0,5,2,4,14,5,6,12,0,3,11,15,14,8,3,8,9,
156       0,5,2,14,8,0,11,9,5,0,6,14,2,2,5,8,3,6,
157       0,7,10,8,15,9,11,1,7,0,8,5,1,9,6,8,6,2,
158       0,0,15,7,4,14,6,2,8,0,13,9,12,14,3,13,12,11};
159    static char ibin[16]={0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15};
160    great ie;
161    unsigned long itmp,ietmp1,ietmp2;
162    char iec[9];
163    int jj,irow,icol,iss,j,l,m;
164
165    ie.r=ie.c=ie.l=0;
166    for (j=16,l=32,m=48;j>=1;j--,l--,m--) {
167       ie.r = (ie.r <<=1) | (bit[iet[j]] & ir ? 1 : 0);
168       ie.c = (ie.c <<=1) | (bit[iet[l]] & ir ? 1 : 0);
169       ie.l = (ie.l <<=1) | (bit[iet[m]] & ir ? 1 : 0);
170       }
171    ie.r ^= k.r;
172    ie.c ^= k.c;
173    ie.l ^= k.l;
174    ietmp1=((unsigned long) ie.c << 16)+(unsigned long) ie.r;
175    ietmp2=((unsigned long) ie.l << 8)+((unsigned long) ie.c >> 8)
176    for (j=1,m=5;j<=4;j++,m++) {
177       iec[j]=ietmp1 & 0x3fL;
178       iec[m]=ietmp2 & 0x3fL;
179       ietmp1 >>= 6;
180       ietmp2 >>= 6;
181       }
182    itmp=0L;
183    for (jj=8;jj>=1;jj--) {
184       j =iec[jj];
185       irow=((j & 0x1) << 1)+((j & 0x20) >> 5);
186       icol=((j & 0x2) << 2)+(j & 0x4)
187          +((j & 0x8) >> 2)+((j & 0x10) >> 4);
188       iss=is[icol][irow][jj];
189       itmp = (itmp <<= 4) | ibin[iss];
190       }

Select Path   Accept   Cancel   Clear All
```

Figure 17: Selecting a Single Path


```
C Source Code of des.c
line # source code
153       0,9,0,4,12,0,7,10,0,0,5,9,11,10,9,11,15,14,
154       0,10,3,10,2,3,13,5,3,0,0,5,5,7,4,0,2,5,
155       0,0,5,2,4,14,5,6,12,0,3,11,15,14,8,3,8,9,
156       0,5,2,14,8,0,11,9,5,0,6,14,2,2,5,8,3,6,
157       0,7,10,8,15,9,11,1,7,0,8,5,1,9,6,8,6,2,
158       0,0,15,7,4,14,6,2,8,0,13,9,12,14,3,13,12,11};
159    static char ibin[16]={0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15};
160    great ie;
161    unsigned long itmp,ietmp1,ietmp2;
162    char iec[9];
163    int jj,irow,icol,iss,j,l,m;
164
165    ie.r=ie.c=ie.l=0;
166    for (j=16,l=32,m=48;j>=1;j--,l--,m--) {
167       ie.r = (ie.r <<=1) | (bit[iet[j]] & ir ? 1 : 0);
168       ie.c = (ie.c <<=1) | (bit[iet[l]] & ir ? 1 : 0);
169       ie.l = (ie.l <<=1) | (bit[iet[m]] & ir ? 1 : 0);
170       }
171    ie.r ^= k.r;
172    ie.c ^= k.c;
173    ie.l ^= k.l;
174    ietmp1=((unsigned long) ie.c << 16)+(unsigned long) ie.r;
175    ietmp2=((unsigned long) ie.l << 8)+((unsigned long) ie.c >> 8);
176    for (j=1,m=5;j<=4;j++,m++) {
177       iec[j]=ietmp1 & 0x3fL;
178       iec[m]=ietmp2 & 0x3fL;
179       ietmp1 >>= 6;
180       ietmp2 >>= 6;
181       }
182    itmp=0L;
183    for (jj=8;jj>=1;jj--) {
184       j =iec[jj];
185       irow=((j & 0x1) << 1)+((j & 0x20) >> 5);
186       icol=((j & 0x2) << 2)+(j & 0x4)
187          +((j & 0x8) >> 2)+((j & 0x10) >> 4);
188       iss=is[icol][irow][jj];
189       itmp = (itmp <<= 4) | ibin[iss];
190       }

Select Path   Accept   Cancel   Clear All
```


Figure 18: Expanded Selected Path

at cycles 7 (between instructions 146 and 147) and 10 (between instructions 148 and 149) in the best-case diagram of Figure 19. In addition, a user may wish to know why there is a difference between best and worst-case times. In this example, the worst-case time requires 36 more cycles than the best-case time due to four instruction cache misses. Other potential pipeline stalls due to structural or data hazards can also be quickly analyzed by a user.

6. Implementation of the User Interface

The user interface is not invoked until the timing analysis tree (examples were shown in Figure 2) is already constructed. Each node within this tree represents a function, loop, or section. Each of these nodes is distinguished by function instances, where a function is uniquely identified by the sequence of call sites required for its invocation. If the user requests a timing prediction for a function, loop, section, or path, then this information can be obtained directly from the timing tree. If a function containing the selected code portion has more than one instance, then the best-case timing prediction is the fastest one of the best-case predictions among all instances. Likewise, the worst-case timing prediction would be the slowest of the worst-case predictions.

Timing predictions for subpaths and ranges of instructions are not stored in the timing analysis tree since there are many combinations of subpaths and ranges of instructions within a single path. If a user requests information for a subpath or a range of instructions, then the appropriate function within the timing analyzer is reinvoked for each instance of the loop or function in which the subpath or range is contained. The authors have found the user interface quite responsive. Even timing prediction requests for subpaths or ranges of instructions can be quickly determined and displayed to the user.

The user interface was implemented using the X Toolkit (Xt) Intrinsics [24] and Xlib [25] libraries. Xt is a library of user-interface objects called widgets and gadgets, which provides a convenient interface for creating and manipulating windows, events, colormaps, and other attributes of the display. Some of the widgets of the timing user interface were implemented using Xlib, the layer on which Xt is based, for

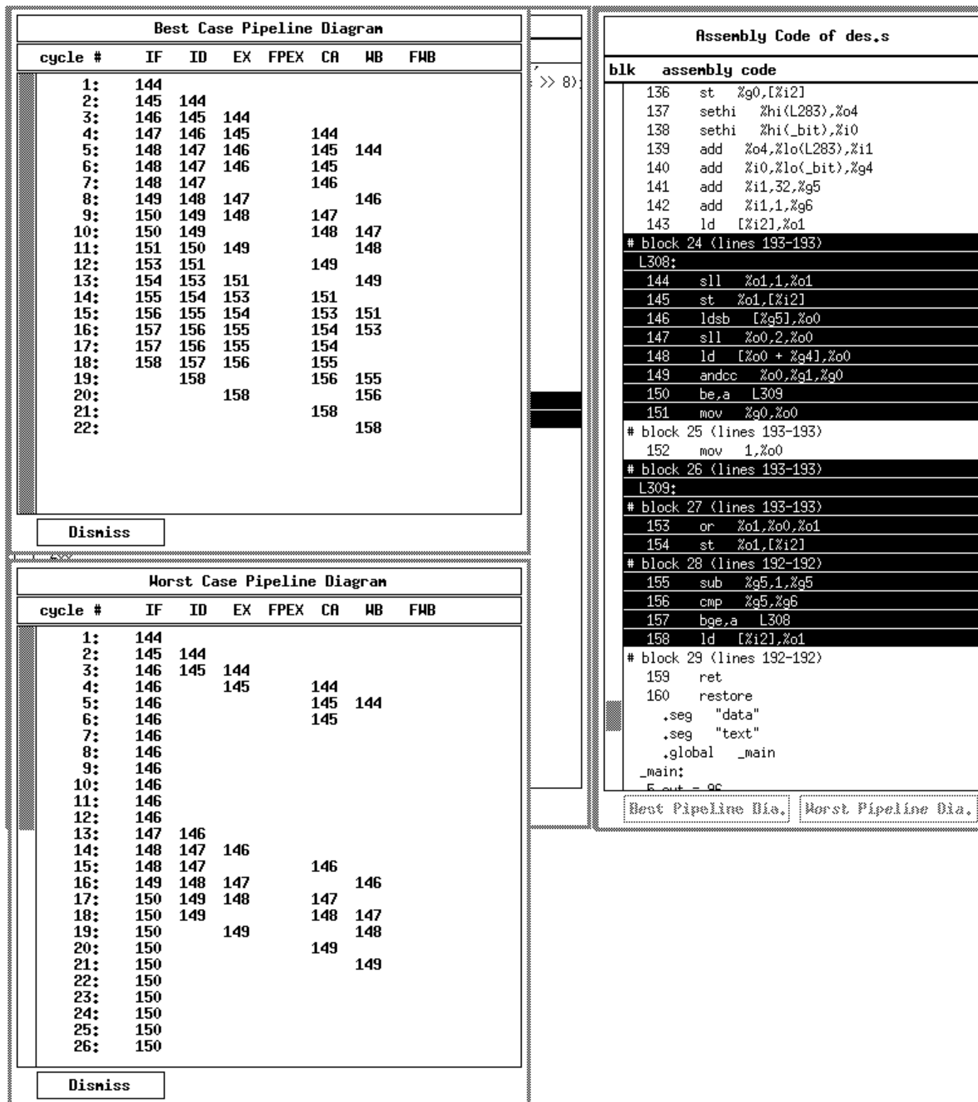

Figure 19: Best and Worst-Case Pipeline Diagrams

manipulating specific attributes. Both libraries come with each distribution of X-Windows. Thus, use of these libraries and the proliferation of X-Windows should enhance the portability of the interface.

7. Future Work

One area in which the user interface could be enhanced is to allow highlighting and selection of portions of a source line. For instance, Figure 14 shows a subpath that includes the initialization of a `for` loop. Yet, the entire first line of the `for` statement is highlighted, which inappropriately includes the test condition and increment as well. Likewise, the selection of this loop for timing predictions should not include the initialization portion of the `for` statment. In addition, consider the `for` loop from source lines 45-51 in the same figure. There are two paths through this loop. However, both paths would be highlighted identically in the source window since the conditional control flow within the loop is entirely contained in line 46, which consists of an assignment statement containing a conditional expression. Yet, the user interface would allow both paths to be selected via the main window and the appropriate assembly instructions would be highlighted.

The user interface could also support selecting portions of source code that includes portions of a line. The character position within the lines where the mouse is pressed and released would affect the corresponding assembly code selected. For instance, if a user wished to select a `for` loop with the mouse in the source window, then one could select a character on the `for` statement that was after the initialization of the loop. The user could also select a portion of an arithmetic expression. For instance, a function call associated with some observable event could be selected [20]. Thus, the compiler and timing analyzer would have to track character positions along with source lines to a basic block level.

At this time the only target for our timing analyzer has been the MicroSPARC I processor. As shown in Figure 1, we have isolated machine-dependent information in an easily editable file to facilitate retargetability of the timing analyzer. However, any additional architectural features used in a new processor and not available in the MicroSPARC I, such as branch prediction buffers, would have to be addressed.

8. Conclusions

The user interface described in this paper provides three methods to allow a user to quickly select a portion of a program for timing prediction. The first method allows a user to quickly inspect whether or not the timing constraints specified in the source code were violated. The second method uses a menu selection approach, which permits a very fine level of selection. For instance, consider that C conditional expressions (i.e. `a > b ? a : b`), logical operators (i.e. `||`, `&&`, and `!`), and assignment of boolean expressions (e.g. `v = i == j;`) often are expressed on a single source line. Yet, the resulting assembly instructions will consist of multiple basic blocks. Likewise, macro calls may be expanded to also generate multiple basic blocks. The menu selection approach allows selection of subpaths down to the machine instruction level.

The third method allows a user to directly select paths from the source window. This method is functionally equivalent to specifying a path constraint in the source code using the first method. While a user may find the third method faster than the second method, some selections of paths or subpaths may not be possible when a single source line has multiple basic blocks. Futhermore, selections with the third method are restricted to only those portions of the program for which the timing analyzer can provide timing predictions. Selection of portions of a program with either of these methods results in the corresponding source lines and assembly instructions being highlighted.

This paper describes a solution for resolving the controversy of whether timing analysis should be performed at a high or low level. This controversy is a result of the desire to relate timing constraints to the source code and to obtain as accurate timing predictions as possible. A user-friendly interface has been presented that assists real-time programmers in relating the analysis of timing constraints associated with source code lines to sequences of machine instructions. Thus, specifying and presenting timing predictions at a high (source code) level can be achieved while retaining the accuracy of low-level (machine code) analysis.

9. Acknowledgements

Mickey Boyd answered many questions about interfacing with the X Toolkit (Xt) Intrinsics and Xlib libraries. Frank Mueller and Randy White proposed several alternatives that resulted in a friendlier user interface.

10. References

1. L. Ko, D. B. Whalley, and M. G. Harmon, “Supporting User-Friendly Analysis of Timing Constraints,” *Proceedings of the ACM SIGPLAN Notices 1995 Workshop on Languages, Compilers, and Tools for Real-Time Systems* **30**(11)(November 1995) pp. 99-107.
2. 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*, (June 1996) pp. 170-178.
3. J. L. Hennessy, “Symbolic Debugging of Optimized Code,” *ACM Transactions on Programming Languages and Systems* **4**(3)(July 1982) pp. 323-344.

4. D. S. Coutant, S. Meloy, and M. Rsucetta, “A Practical Approach to Source-Level Debugging of Globally Optimized Code,” *Proceedings of the SIGPLAN ’88 Symposium on Programming Language Design and Implementation*, (June 1988) pp. 125-134.
5. G. Brooks, G. Hansen, and S. Simmons, “A New Approach to Debugging Optimized Code,” *Proceedings of the SIGPLAN ’92 Conference on Programming Language Design and Implementation*, (June 1992) pp. 1-11.
6. A. Adl-Tabatabai and T. Gross, “Detection and Recovery of Endangered Variables Caused by Instruction Scheduling,” *Proceedings of the SIGPLAN ’93 Conference on Programming Language Design and Implementation*, (June 1993) pp. 13-25.
7. Texas Instruments, Inc., *Product Preview of the TMS390S10 Integrated SPARC Processor*, 1993.
8. R. Arnold, F. Mueller, D. Whalley, and M. Harmon, “Bounding Worst-Case Instruction Cache Performance,” *Proceedings of the Fifteenth IEEE Real-Time Systems Symposium*, (December 1994) pp. 172-181.
9. C. A. Healy, D. B. Whalley, and M. G. Harmon, “Integrating the Timing Analysis of Pipelining and Instruction Caching,” *Proceedings of the Sixteenth IEEE Real-Time Systems Symposium*, (December 1995) pp. 288-297.
10. F. Mueller, *Static Cache Simulation and Its Applications,* PhD Dissertation, Florida State University, Tallahassee, FL (August 1994).
11. F. Mueller, D. B. Whalley, and M. G. Harmon, “Predicting Instruction Cache Behavior,” *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, (June 1994)
12. Nagham M. Al-Yaqoubi, *Reducing Timing Analysis Complexity by Partitioning Control Flow,* Masters Project, Florida State University, Tallahassee, FL (1997).
13. C. Y. Park and A. C. Shaw, “Experiments with a Program Timing Tool Based on a Source-Level Timing Schema,” *Computer* **24**(5)(May 1991) pp. 48-57.
14. I. Lee and V. Gehlot, “Language Constructs for Distributing Real-Time Programming,” *Proceedings of the IEEE Real-Time Systems Symposium*, (December 1985) pp. 57-66.
15. Y. Ishikawa, H. Tokuda, and C. Mercer, “Object-Oriented Real-Time Language Design: Constructs for Timing Constraints,” *Proceedings of the ACM Object-Oriented Programming: Systems, Languages, and Applications*, (October, 1990) pp. 289-296.
16. V. Nirkhe, S. Tripathi, and A. Agrawala, “Language Support for the Maruti Real-Time System,” *Proceedings of the IEEE Real-Time Systems Symposium*, (December 1990) pp. 257-266.
17. E. Kligerman and A. Stoyenko, “Real-Time Euclid: A Language for Reliable Real-Time Systems,” *IEEE Transactions on Software Engineering* **12**(9)(September 1986) pp. 941-949.
18. K. Kenny and K. Lin, “Building Flexible Real-Time Systems Using the Flex Language,” *IEEE Computer* **24**(5)(May 1991) pp. 70-78.
19. B. Dasarathy, “Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Methods of Validating Them,” *IEEE Transactions on Software Engineering* **11**(1)(January 1985) pp. 80-86.
20. S. Hong and R. Gerber, “Compiling Real-Time Programs into Schedulable Code,” *Proceedings of the SIGPLAN ’93 Conference on Programming Language Design and Implementation*, (June 1993) pp. 166-176.
21. M. Humphrey and J. Stankovic, “CAISARTS: A Tool for Real-Time Scheduling Assistance,” *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, (June 1996) pp. 150-159.
22. J. W. Davidson and D. B. Whalley, “Quick Compilers Using Peephole Optimizations,” *Software—Practice & Experience* **19**(1)(January 1989) pp. 195-203.
23. 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*, (June 1988) pp. 329-338.
24. A. Nye and T. O’Reilly, *X Toolkit Intrinsics Programming Manual,* O’Reilly & Associates, Inc. (1990).
25. A. Nye, *Xlib Programming Manual,* O’Reilly & Associates, Inc. (1990).