Supporting the Specification and Analysis of Timing Constraints*

Lo Ko, Christopher Healy, Emily Ratliff
Robert Arnold, and David Whalley
Computer Science Dept., Florida State Univ.
Tallahassee, FL 32306-4019
e-mail: whalley@cs.fsu.edu, phone: (904) 644-3506

Marion Harmon
Computer and Information Systems Dept.
Florida A&M University
Tallahassee, FL 32307-3101
e-mail: harmon@cis.famu.edu, phone: (904) 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 user is allowed specify timing constraints within the source code of a C program. A user interface for a timing analyzer was developed to depict whether these constraints were violated or met. In addition, the interface allows portions of programs to be quickly selected with the corresponding bounded times, source code lines, and machine instructions automatically displayed. 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

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 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.

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 [2]. Other papers are available for readers interested in how the timing predictions are actually obtained [3], [4].

2. Overview

The design of the environment described in this paper includes 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

*This work was supported in part by the Office of Naval Research under contract number N00014-94-1-0006. A preliminary version of the viewer was described in [1]. Robert Arnold is currently working for Peek Traffic Transyt. Lo Ko is currently working for the Florida Department of Insurance.

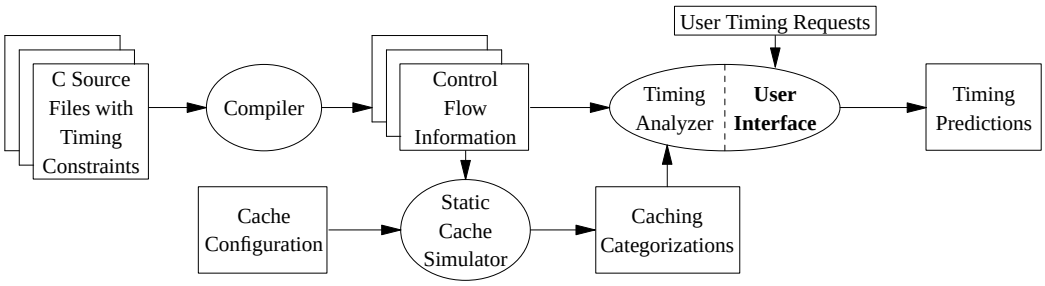

Figure 1: Overview of Obtaining Timing Predictions

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. 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. 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 each basic block, to estimate the best-case and worst-case performance for each loop and function within the program.[1] Finally, a graphical user interface (GUI) is invoked that allows the user to request the status of the constrained sections and timing predictions for other specified portions of the program.

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 [5], C++ [6], and Euclid [7]. In addition, many new real-time languages with features for expressing timing constraints have been proposed and implemented. Unfortunately, there has been little work in the area of providing support for user analysis of timing 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. 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* [9]. This constraint information was passed through the back end of a C compiler called *vpo* [10]. Source lines associated with basic blocks are tracked while performing the optimizations in *vpo*. 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 2 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.

[1] The user is prompted for the minimum and maximum loop iterations of loops when it could not be calculated by the compiler. Note that at this time only pipeline and instruction caching behavior is analyzed [3], [4]. Work is currently proceeding on analyzing data caching behavior.

```
 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 2: Source Code with Timing Constraints

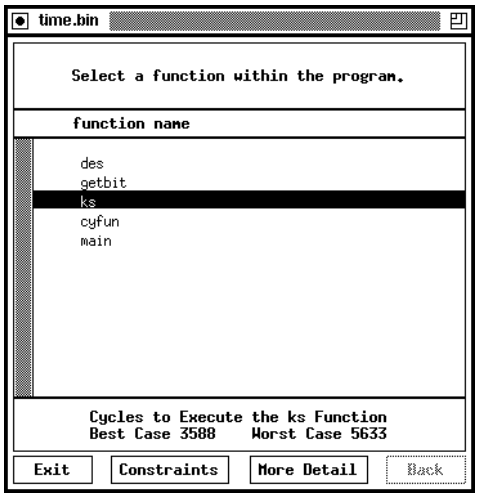

Figure 3: Main Window at Function Level

5. User Interface

The user interface is invoked after the timing analyzer has analyzed the entire program. Figures 3 and 4 depict the three windows that are always displayed for the timing analysis graphical user interface. Figure 3 shows the main window of the user interface. The top section of the main window displays a message 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.

Figure 4 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 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 4 also illustrates a pitfall a user may face with the tool. Source code lines are only tracked to a basic block level. Sometimes optimizations move individual instructions from one basic block to another. For instance, the last instruction in block 5 of Figure 4 corresponds to the assignment of zero to `itmp.l` 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 a function or natural loop instance. A function is analyzed as though it was a natural loop that iterates only once when entered.

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 realized 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.

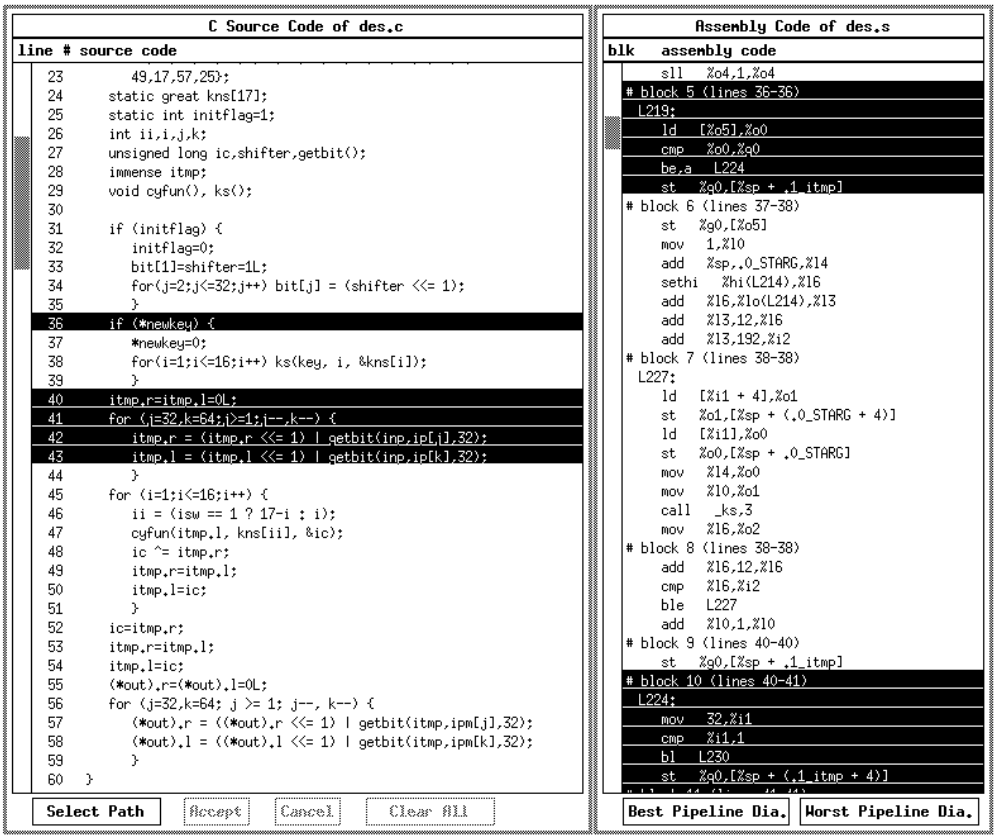

Figure 4: Source Code and Assembly Code Windows

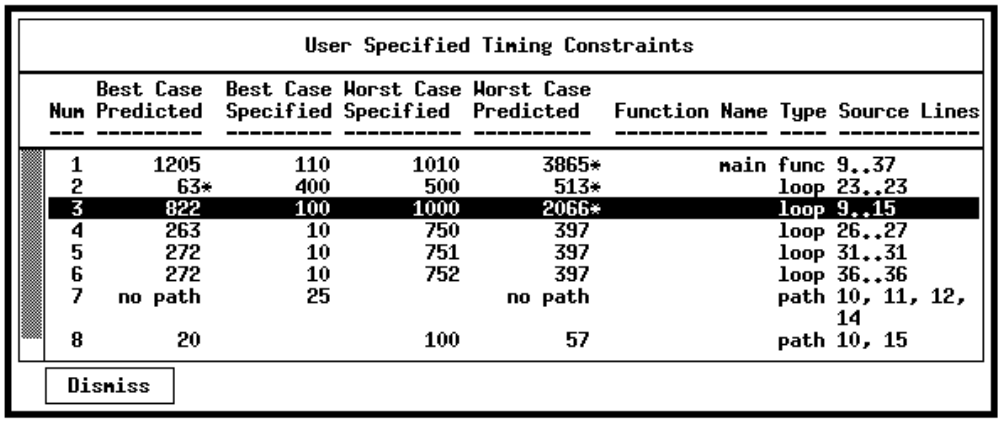

Figure 5: Constraints Window

6. 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 5 shows the constraint window, which contains a scrollable display of the user-specified constraints. A user may choose to have 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.

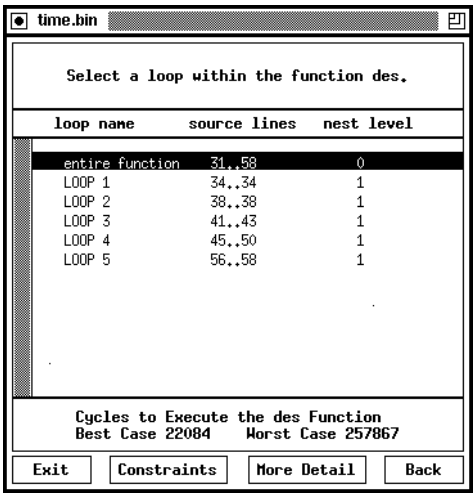

Figure 6: Main Window at Loop Level

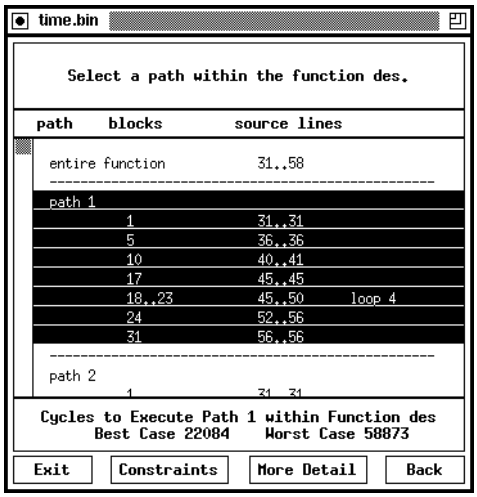

Figure 7: Main Window at Path Level

For each constraint the window displays the specified and predicted best and worst-case times in clock cycles[2] 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 5), such as the `then` and `else` portions of an `if-then-else` construct.

7. 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 five 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 3, 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 four levels are shown in Figures 6 through 9. Selection of a function, loop, 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. 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. 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 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. 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.[3] Hence, the user is allowed to obtain a very fine-grain level of timing predictions.

Thus, there are five levels of detail in a program that a user can view: functions, loops, paths, subpaths, and ranges of machine instructions. At most five selections in the main window are required for a user to quickly choose

[2] 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.

[3] 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.

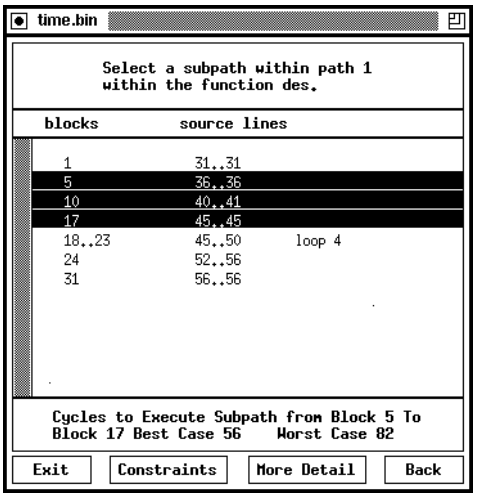

Figure 8: Main Window at Subpath Level

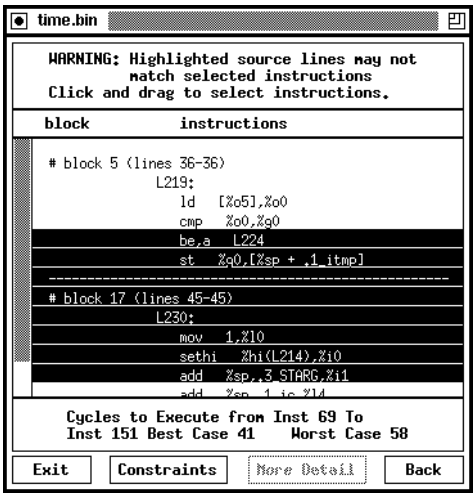

Figure 9: Main Window at Assembly Level

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.

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

The other method for accessing a portion of the program is to select lines of source code directly by using the mouse as depicted in Figure 10. After clicking on the **Select Path** button, the user highlights the source lines within the path to be timed. 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.

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 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 11 and 12 show the best and worst case set of source lines, respectively, that would be displayed associated with the source lines selected in Figure 10. 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. Thus, it must include all source lines associated with a basic block if any source lines in that block are selected.

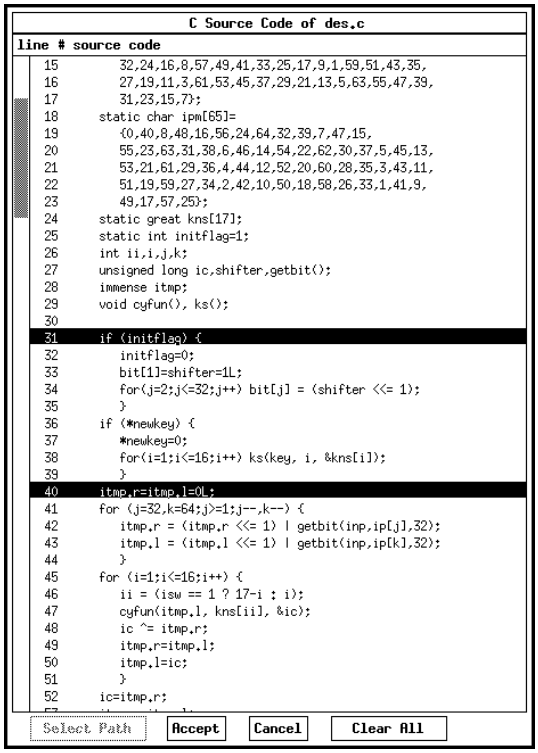

Figure 10: Selecting a Path via the Source Code

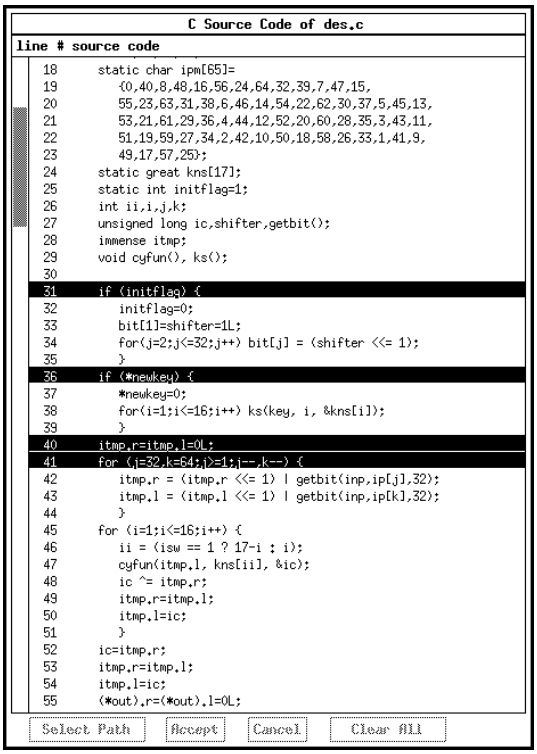

Figure 11: Best Case Path from Source Lines Selected in Figure 10

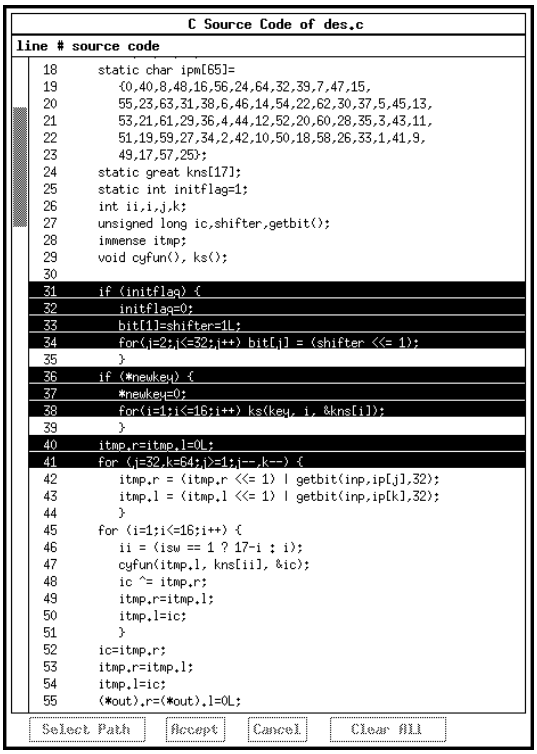

Figure 12: Worst Case Path from Source Lines Selected in Figure 10

9. 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 4 to obtain a pipeline diagram of the best and worst-case performance of a path containing no loops or calls. Figure 13 shows both the best and worst-case pipeline diagrams associated with a path through a loop. In contrast to Figure 4, 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 at cycles 7 (between instructions 146 and 147) and 10 (between instructions 148 and 149) in the best-case diagram of Figure 13. 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.

10. Implementation

The user interface is not invoked until the timing analysis tree is already constructed. Each node within this tree represents a loop or function. 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 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

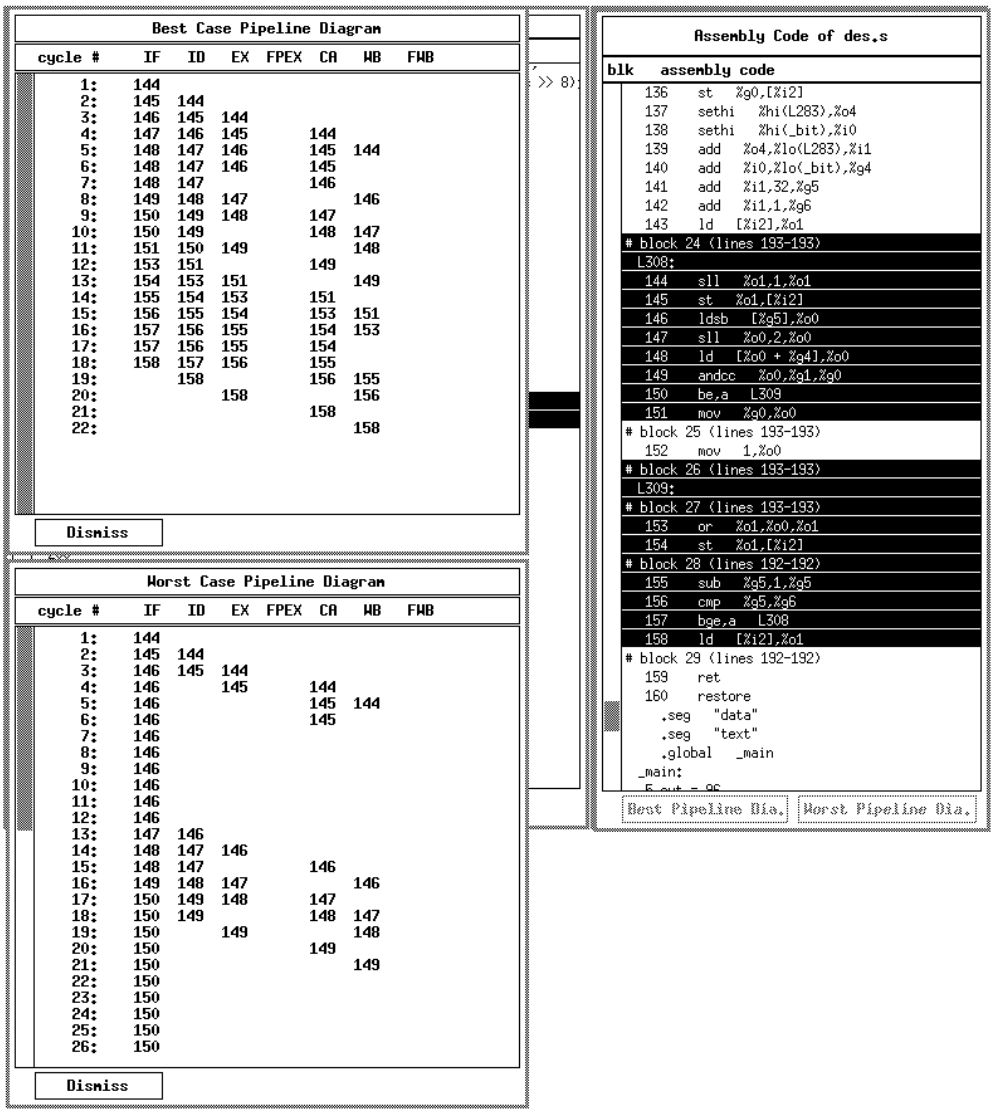

Figure 13: Best and Worst-Case Pipeline Diagrams

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 user interface was implemented using the X Toolkit (Xt) Intrinsics [11] and Xlib [12] libraries. 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.

11. 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 12 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 [13]. Thus, the compiler and timing analyzer would have to track character positions along with source lines to a basic block level.

12. 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.

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.

13. 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. The anonymous referees made several suggestions that improved the quality of the paper.

14. 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) pp. 99-107 (November 1995).

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

[3] 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).

[4] 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).

[5] 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).

[6] 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*, pp. 289-296 (October, 1990).

[7] E. Kligerman and A. Stoyenko, "Real-Time Euclid: A Language for Reliable Real-Time Systems," *IEEE Transactions on Software Engineering* **12**(9) pp. 941-949 (September 1986).

[8] B. Dasarathy, "Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Methods of Validating Them," *IEEE Transactions on Software Engineering* **11**(1) pp. 80-86 (January 1985).

[9] J. W. Davidson and D. B. Whalley, "Quick Compilers Using Peephole Optimizations," *Software—Practice & Experience* **19**(1) pp. 195-203 (January 1989).

[10] 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).

[11] A. Nye and T. O'Reilly, *X Toolkit Intrinsics Programming Manual*, O'Reilly & Associates, Inc. (1990).

[12] A. Nye, *Xlib Programming Manual*, O'Reilly & Associates, Inc. (1990).

[13] S. Hong and R. Gerber, "Compiling Real-Time Programs into Schedulable Code," *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pp. 166-176 (June 1993).