Supporting User-Friendly Analysis of Timing Constraints*

LO KO AND DAVID B. WHALLEY

Department of Computer Science, Florida State University, Tallahassee, FL 32306-4019, U.S.A.
e-mail: whalley@cs.fsu.edu, phone: (904) 644-3506

MARION G. HARMON

Dept. of Computer and Information Systems, Florida A&M University, Tallahassee, FL 32307-3101, U.S.A.
e-mail: harmon@cis.famu.edu, phone: (904) 599-3042

SUMMARY

Real-time programmers have to deal with the problem of relating timing constraints associated with source code lines to sequences of machine instructions. This paper describes an interface that was developed to assist users in this task. Portions of programs can be quickly selected and the corresponding bounded times, source code lines, and machine instructions are automatically displayed. In addition, users are restricted to only selecting portions of the program for which timing bounds can be obtained. The result is a user-friendly interface that assists programmers in the analysis of timing constraints within a program.

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 more 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 bounds. 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 very 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 includes whether the results are produced on time. Thus, the timing analysis should be at the level of the 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 [Hen82, CMR88, BHS92, AdG93].

This paper describes the implementation of a user interface to support analysis of timing constraints. The approach that was used involves performing the timing analysis on the machine code of a program and depicting the relationship between the machine instructions (i.e. assembly code) and the corresponding source code lines.

OVERVIEW

The design of the timing analysis user interface described in this paper included the following goals:

(1) A user should be able to quickly select a portion of the program for timing prediction.

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

(3) The corresponding portions of the source code and machine code levels of the program selected by the user for timing prediction should be depicted.

Figure 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. The 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.[1] 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 best-case and worst-case instruction caching performance for each loop and function within the program. Finally, user interface windows are displayed allowing one to request the timing bounds for

*This work was supported in part by the Office of Naval Research under contract number N00014-94-1-0006.

[1] Note that at this time only instruction caching behavior is analyzed. Work is currently proceeding on analyzing pipelining and data caching behavior.

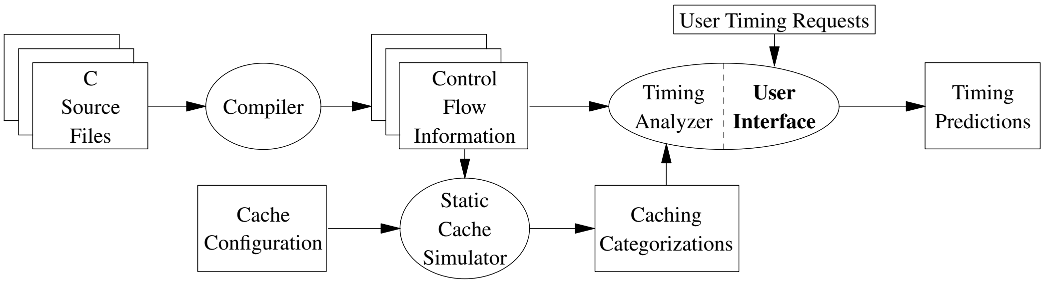

Figure 1: Overview of Obtaining Timing Predictions

portions of the program.

A timing tree is constructed 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 [AMW94]. A function is analyzed as though it was a natural loop that iterates only once when entered. The loops in the timing analysis tree are processed in a bottom-up fashion. The entire tree is analyzed before a user is allowed to request timing predictions for portions of the program.

USER INTERFACE

Figures 2 and 3 depict the three windows that are always displayed for the timing analysis user interface. Figure 2 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 in 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 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.

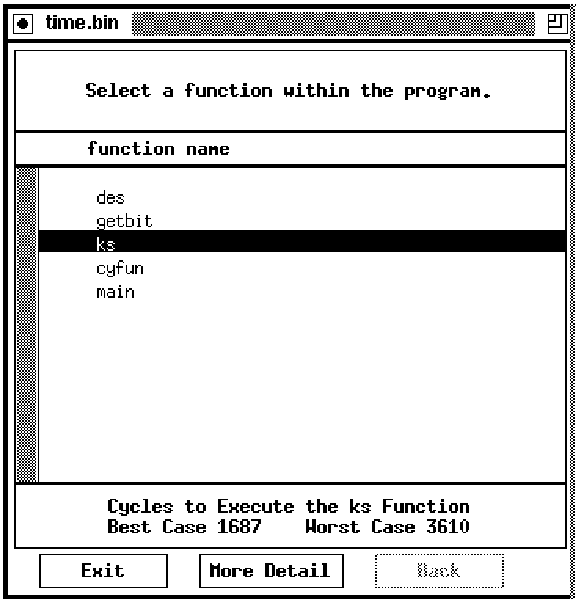

Figure 2: Main Window at Function Level

Figure 3 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 be explained shortly. 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.

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

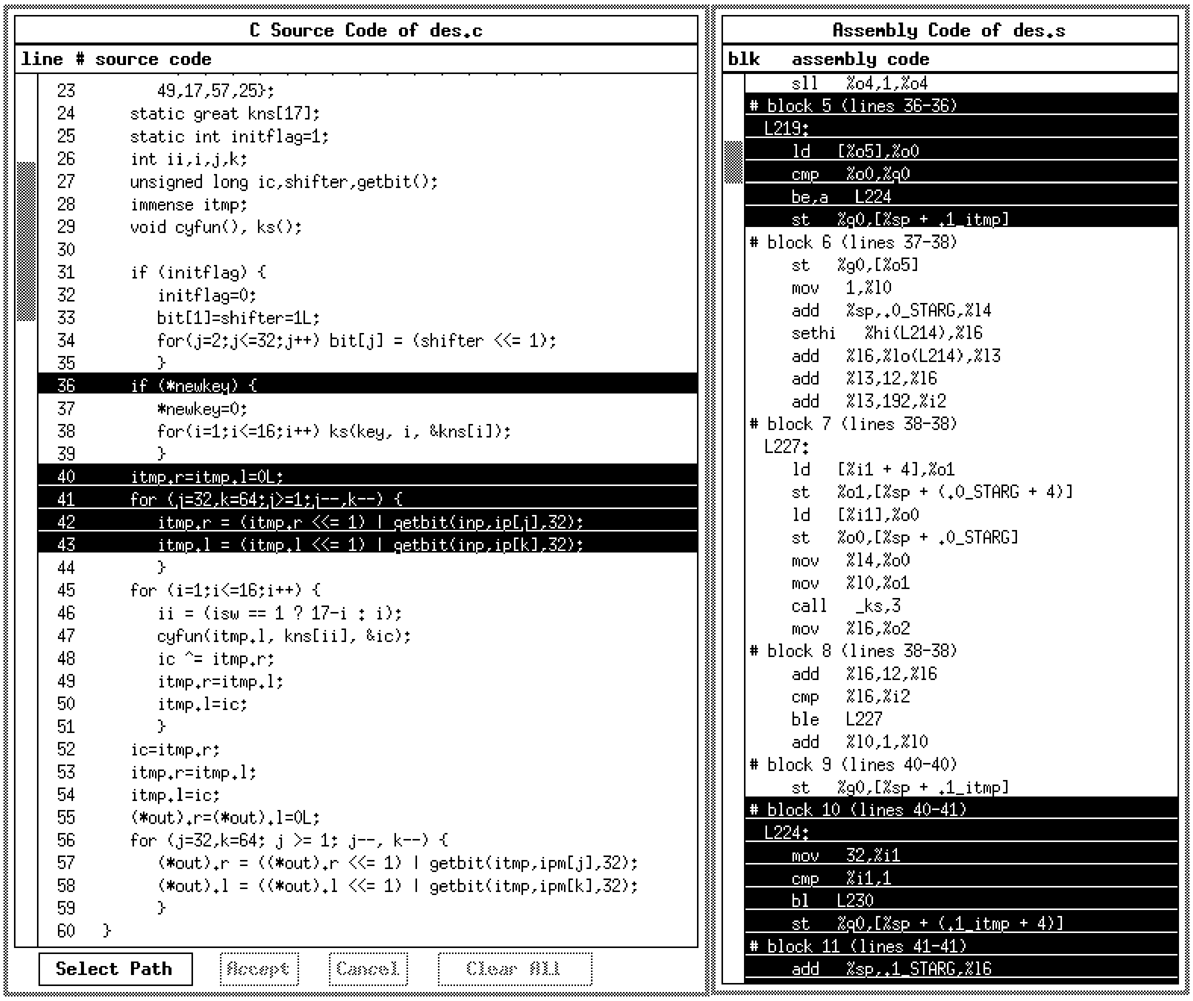

Figure 3: Source Code and Assembly Code Windows

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 simply want the capability of accessing specified portions of the program as quickly as possible. The user interface described in this paper has two different methods for accessing portions of a program.

Selecting Portions of a Program Using the Main Window

The first 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 four 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 2, which was shown previously 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 cycles representing the execution of the entire program.

The next lower level of detail consists of loops as shown in Figure 4. The entire function and each loop within the function are listed in the display. Again selection of the function or a loop will cause the corresponding bounded number of cycles to be displayed and the appropriate lines to be highlighted in the other two windows. Note that by each loop number is its range of source lines and nesting level within the function to facilitate identification by the user.

The next lower level of detail displays paths as shown in Figure 5. A path is defined as a unique sequence of basic blocks connected by control-flow transitions. Thus, each path is depicted in the main display as a list of blocks and corresponding source line ranges. Each loop path starts with the loop header and is terminated by a block with a transition to the loop header or 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.

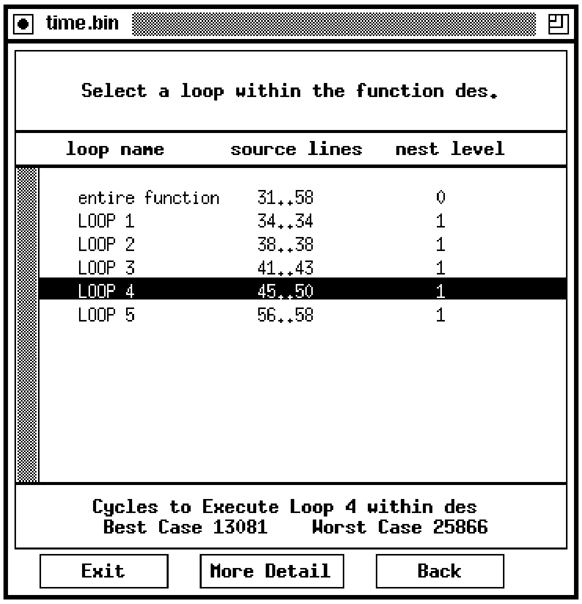

Figure 4: Main Window at Loop Level

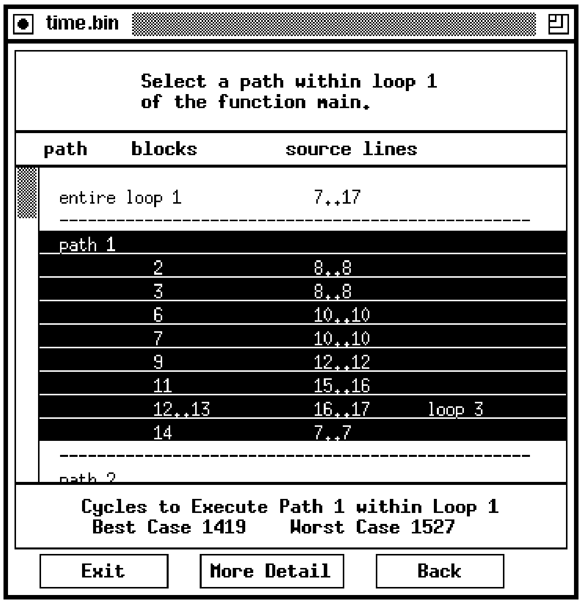

Figure 5: Main Window at Path Level

The final level of detail consists of subpaths as shown in Figure 6. 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. Note that a basic block is the finest level of detail for which timing predictions may be obtained. 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 optimizations. The source lines for each C statement were identified by the front end of the compiler and are passed to the back end, which associated a source line range with each basic block. These ranges of source lines were then maintained whenever basic blocks were moved (i.e. the control flow was adjusted) during optimizations.[2] Tracking individual instructions is much more challenging and has been the target of much research associated with debugging optimized code. Note that while it would not be difficult to provide timing bounds at a more 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.

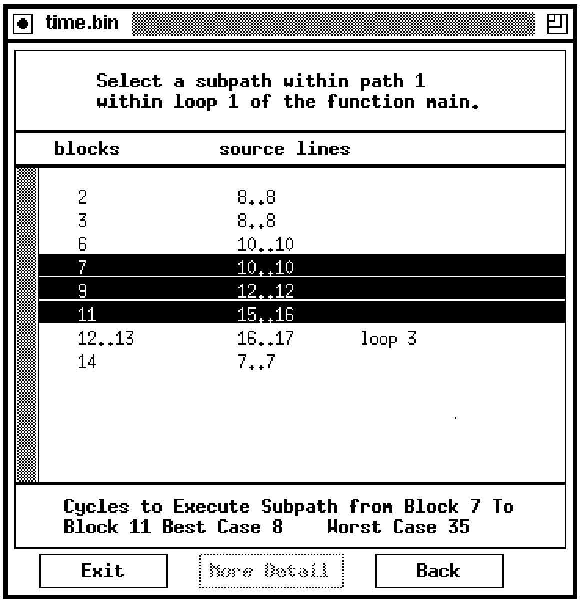

Figure 6: Main Window at Subpath Level

Thus, there are four levels of detail of a program that a user can view: functions, loops, paths, and subpaths. A user can quickly choose any specifiable portion of the program in only four selections in the main window. The appropriate timing analysis information is extracted for

[2] 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) was filled with an instruction from block 10 (source lines 40-41).

each selection by the user. If there is more than one instance of the portion selected by the user (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.

Selecting Portions of a Program Using the Mouse on the Source Window

The other method for accessing a portion of the program is to directly select lines of source code using the mouse as depicted in Figure 7. First, the user clicks on the **Select Path** button at the bottom of the source code window. Next, the user highlights the lines which must be 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 7. A user may quickly obtain the best-case and worst-case timing predictions for a segment of code by selecting only two source lines. 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.

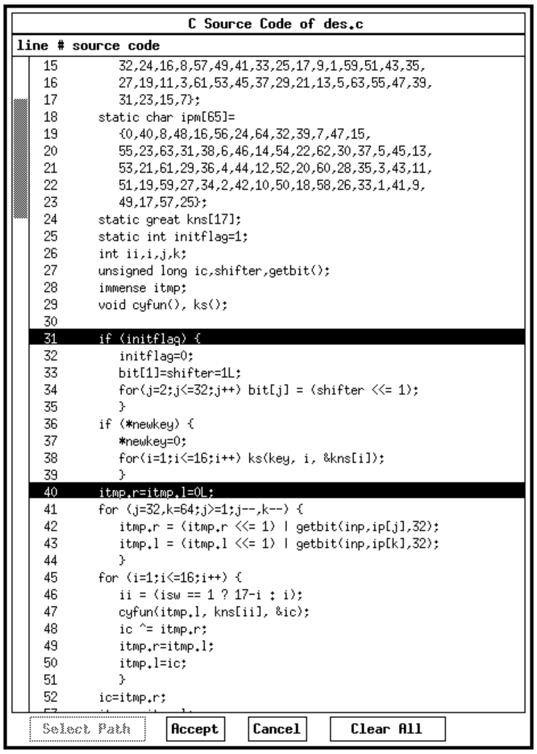

Figure 7: Selecting a Path via the Source Code Window

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 one of three popups is displayed depending upon the number of subpaths that are associated with the highlighted source lines (zero, one, or more than one). The popup shown in Figure 8 is displayed when there are multiple subpaths corresponding to the selected path, as in Figure 7. Note that if there is no subpath or only one subpath that is associated with the highlighted lines, then the user is given the option of selecting the loop or function that most tightly encloses the highlighted lines.

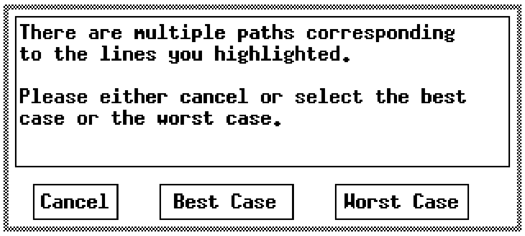

Figure 8: Popup Window after Selecting Lines with Multiple Paths

Figures 9 and 10 show the best and worst case set of source lines that were displayed associated with the options to be selected in Figure 8, respectively. Figure 9 shows the highlighted lines after the user selects to view the best case

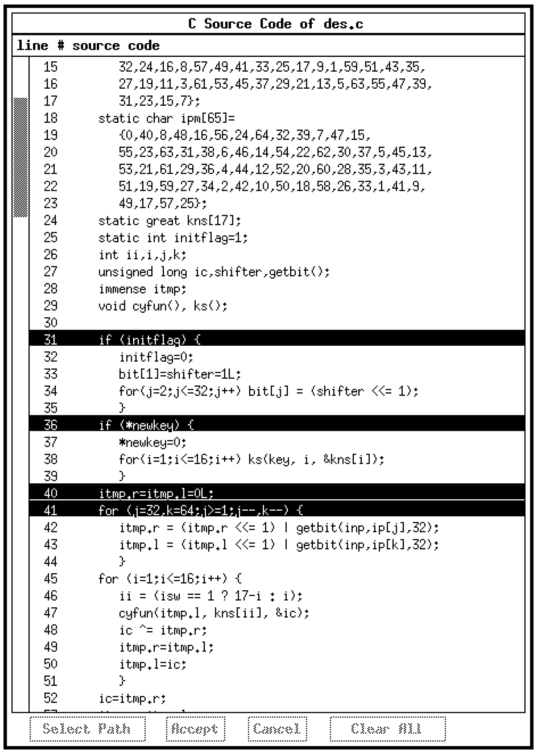

Figure 9: Best Case Path from Source Lines Selected in Figure 7

timing of the path selected. Note that instructions associated with other source lines may have to be executed as well. 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 interface restricts the user 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 of the source lines within that block are selected.

Figure 10 shows the highlighted lines after the user selects to view the worst case timing of the path selected. Note this time both if statements are entered. The popup shown in Figure 8 will remain on the screen until the user selects the *cancel* option. This allows the user to repeatedly view both options without reselecting the path.

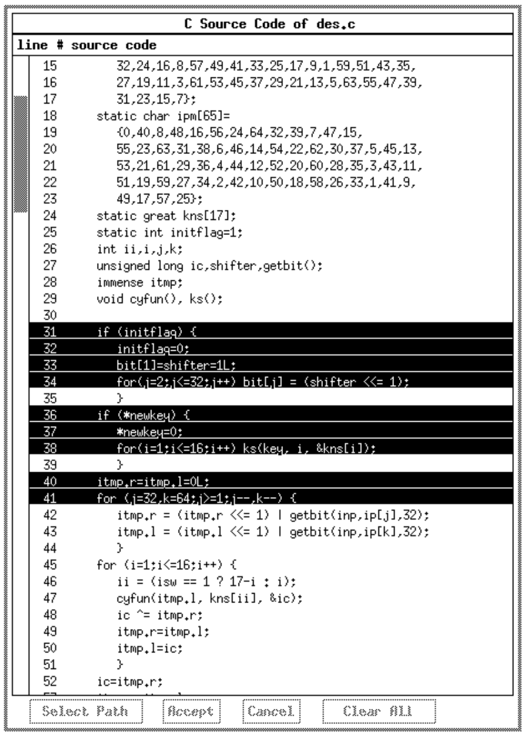

Figure 10: Worst Case Path from Source Lines Selected in Figure 7

Figure 11 shows the selection of an `if-then-else` construct. As the popup indicates in Figure 12, 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.

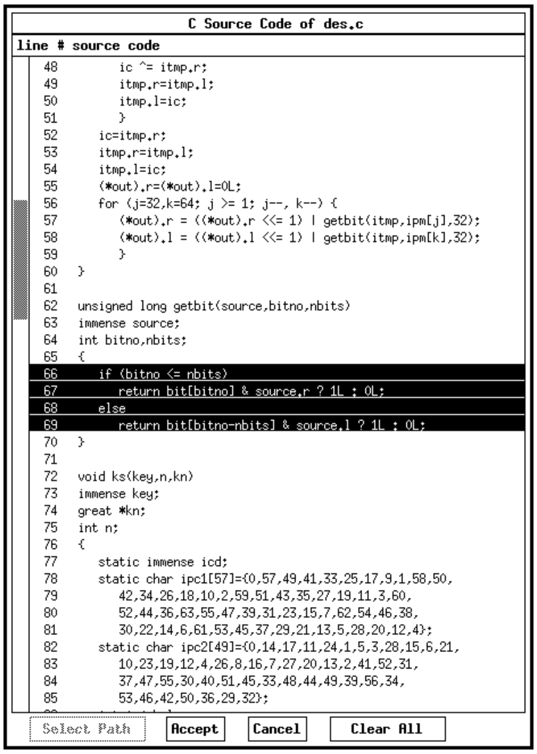

Figure 11: Selecting an Infeasible Path

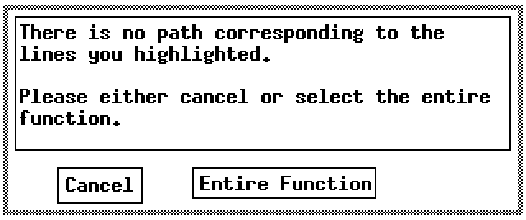

Figure 12: Popup Window after Selecting an Infeasible Path

Figure 13 shows the selection of source lines that correspond to a single path. Figure 14 shows the popup window that appears after selecting the path. The user has the option of viewing that path or the entire function. Notice that the user has selected lines partially within a loop and outside of the loop. This selection illustrates another of the limitations of the timing analysis. Any path that enters a loop is assumed to execute the entire loop. Figure 15 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 7 and 9, 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.

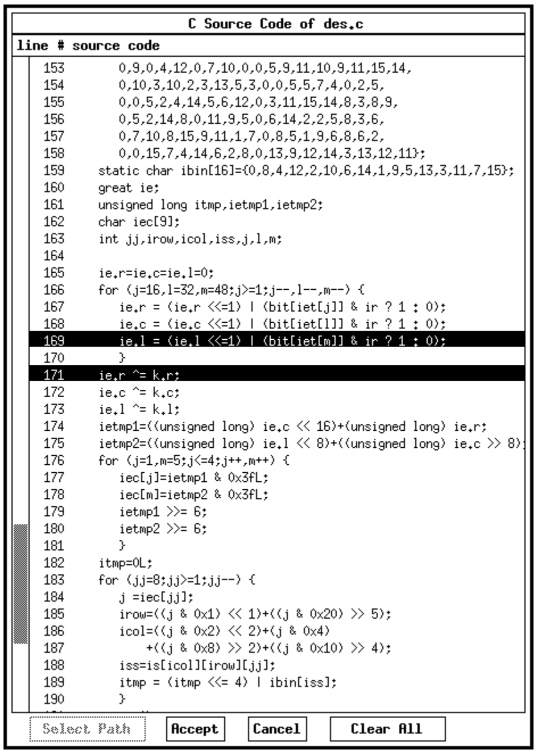

Figure 13: Selecting a Single Path

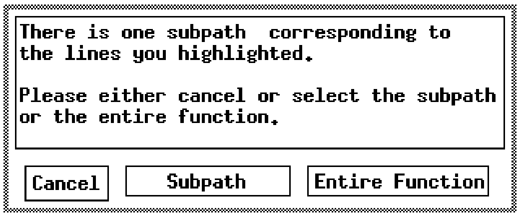

Figure 14: Popup Window after Selecting Lines Associated with a Single Path

IMPLEMENTATION

The implementation of the timing interface includes obtaining information from the timing analysis tree and graphically presenting timing predictions to a user.

Obtaining Information from the Timing Analysis Tree

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 are 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 or a loop, then this information can be obtained directly from the timing tree. If a function or loop 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.

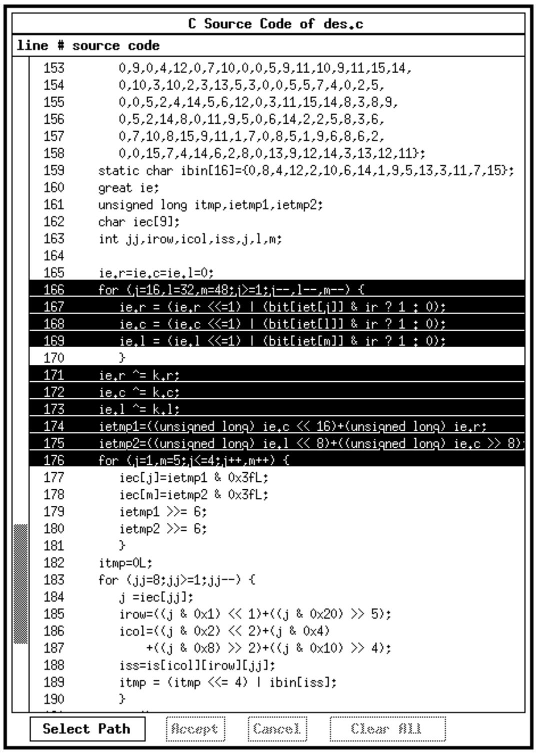

Figure 15: Expanded Selected Path

Timing predictions for paths are not stored in the timing analysis tree. A path may be used multiple times during the analysis of a loop. On each iteration of its loop, the timing prediction of a path may change [AMW94]. If a user requests information for a path or a subpath, then the appropriate function within the timing analyzer is reinvoked for each instance of the loop or function in which the path or subpath is contained.

Graphical Presentation of Timing Predictions

The user interface was implemented using the X Toolkit (Xt) Intrinsics [NyO90] and Xlib [Nye90] libraries. Both of these 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.

FUTURE WORK

There are several areas in which the user interface could be enhanced. First, only portions of a line could be highlighted at appropriate times. For instance, Figure 9 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. Yet both paths would be highlighted identically 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.

Another area in which the user interface could be enhanced is to display information that would describe how the timing prediction was obtained. Work is currently being performed on the timing analyzer to include analysis of pipeline performance. The authors intend to extend the user interface to provide the ability to display a scrollable pipeline diagram for paths and subpaths that do not contain nested loops or calls to functions. Such diagrams would help a user understand when pipeline hazards could cause stalls and when cache miss delays could occur.

An additional enhancement would be to allow a user to request another level of detail of timing predictions. Rather than limiting the user to the level of timing a sequence of blocks, the user could instead indicate the initial instruction within the first block of a subpath and the last instruction within the last block. Such an enhancement would allow a knowledgeable user to avoid the limitation of multiple high-level language statements always having to be grouped together within the same subpath. However, the user would have to be informed that the corresponding highlighted source lines would probably not be accurate.

At this time the only information the timing analyzer requests from the user is the minimum and maximum iterations for loops that the compiler could not automatically determine. These requests occur each time the program is analyzed. A better interface would be to allow assertions to be placed within the source code. Besides specifying the maximum and minimum iterations of a loop, the user could also specify timing constraints. The timing analyzer could be invoked as part of the compilation process and could inform the user when a timing constraint might be violated.

CONCLUSIONS

The user interface described in this paper provides two methods to allow a user to quickly select a portion of a program for timing prediction. The first 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 basic block level. The second method allows a user to directly select paths from the source window. While a user may find this method faster, some selections of paths or subpaths may not be possible using this method when a single source line has multiple basic blocks. Futhermore, selections with this 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. Real-time programmers may have to deal with the problem of relating timing constraints associated with source code lines to sequences of machine instructions. A user-friendly interface was presented that assists real-time programmers in relating the analysis of timing constraints associated with source code lines to sequences of machine instructions. Thus, presenting timing predictions at a high (source code) level can be achieved while retaining the accuracy of low-level (machine code) analysis.

ACKNOWLEDGEMENTS

Robert Arnold provided assistance for obtaining timing predictions on code portions from the timing analyzer. Mickey Boyd answered many questions about interfacing with the X Toolkit (Xt) Intrinsics and Xlib libraries. Robert Arnold, Chris Healy, Frank Mueller, Emily Ratliff and Randy White offered helpful suggestions that resulted in a friendlier user interface.

REFERENCES

[AdG93] 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*, pp. 13-25 (June 1993).

[AMW94] 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).

[BHS92] 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*, pp. 1-11 (June 1992).

[CMR88] 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*, pp. 125-134 (June 1988).

[Hen82] J. L. Hennessy, “Symbolic Debugging of Optimized Code,” *ACM Transactions on Programming Languages and Systems* **4**(3) pp. 323-344 (July 1982).

[Nye90] A. Nye, *Xlib Programming Manual,* O’Reilly & Associates, Inc. (1990).

[NyO90] A. Nye and T. O’Reilly, *X Toolkit Intrinsics Programming Manual,* O’Reilly & Associates, Inc. (1990).