Isolation and Analysis of Optimization Errors

MICKEY R. BOYD AND DAVID B. WHALLEY
Department of Computer Science B-173, Florida State University, Tallahassee, FL 32306, U.S.A.
e-mail: whalley@cs.fsu.edu   phone: (904) 644-3506

SUMMARY

This paper describes two related tools developed to support the isolation and analysis of optimization errors in the *vpo* optimizer. Both tools rely on *vpo* identifying sequences of changes, referred to as transformations, that result in semantically equivalent (and usually improved) code. One tool determines the first transformation that causes incorrect output of the execution of the compiled program. This tool not only automatically isolates the illegal transformation, but also identifies the location and instant the transformation is performed in *vpo*. To assist in the analysis of an optimization error, a graphical optimization viewer was also implemented that can display the state of the generated instructions before and after each transformation performed by *vpo*. Unique features of the optimization viewer include reverse viewing (or undoing) of transformations and the ability to stop at breakpoints associated with the generated instructions. Both tools are useful independently. Together these tools form a powerful environment for facilitating the retargeting of *vpo* to a new machine and supporting experimentation with new optimizations. In addition, the optimization viewer can be used as a teaching aid in compiler classes.

INTRODUCTION

While the time required to retarget a back end of a compiler to a new machine has decreased over the years, performing this task in an expeditious manner still remains a problem. One reason is that the rate of new machines being introduced has increased. In addition, there is an increasing reliance on compilers to perform more sophisticated optimizations to exploit architectural features. Usually these optimizations can be applied most effectively in the back ends of compilers [BeD88].

Much of the effort required to retarget a back end occurs during testing. Often much time is spent determining why incorrect code is generated or optimizations cannot be applied for specific programs. Most back ends store information about the program that is being compiled in an encoded internal format, which exacerbates these problems. While such formats require less space and allow optimizations to occur more rapidly, they also increase the difficulty of analyzing a specific problem.

This paper describes two tools that assist a compiler writer in isolating and analyzing optimization errors. First, an optimization error isolator is presented that can automatically determine the first transformation during the optimization of a program that causes the output of the execution to be incorrect. Second, an optimization viewer is described that can graphically depict the state of the generated instructions before and after each transformation performed by the optimizer. One can easily examine the invalid transformation discovered by the optimization error isolator with the optimization viewer and quickly access the point in the compiler when the transformation is performed.

OVERVIEW OF THE COMPILER

The tools described in this paper support isolating and analyzing optimization errors for the compiler technology known as *vpo* (Very Portable Optimizer) [BeD88, Dav86, DaF84]. The optimizer, *vpo*, replaces the traditional code generator used in many compilers and has been used to build C, Pascal, and Ada compilers. The back end is retargeted by supplying a description of the target machine. Using the diagrammatic notation of Wulf [WJW75], Figure 1 shows the overall structure of a set of compilers constructed using *vpo*. Vertical columns within a box represent logical phases which operate serially. Columns divided horizontally into rows indicate that the subphases of the column may be executed in an arbitrary order. IL is the Intermediate Language generated by a front end. Register transfers or register transfer lists (RTLs) describe the effects of machine instructions and have the form of conventional expressions and assignments over the hardware's storage cells. For example, the RTL

```
r[1] = r[1] + r[2]; cc = r[1] + r[2] ? 0;
```

represents a register-to-register integer add on many machines. While any particular RTL is machine-specific, the *form* of the RTL is machine-independent.

All phases of the optimizer manipulate RTLs. An advantage of using RTLs as the sole intermediate representation is that many phase ordering problems are eliminated. Most optimizations can be invoked in any order and are allowed to iterate until no more improvements can be found.

The RTLs are stored in a data structure in *vpo* that also contains information about the order and controlflow of the RTLs within a function. The *vpo* optimizer was

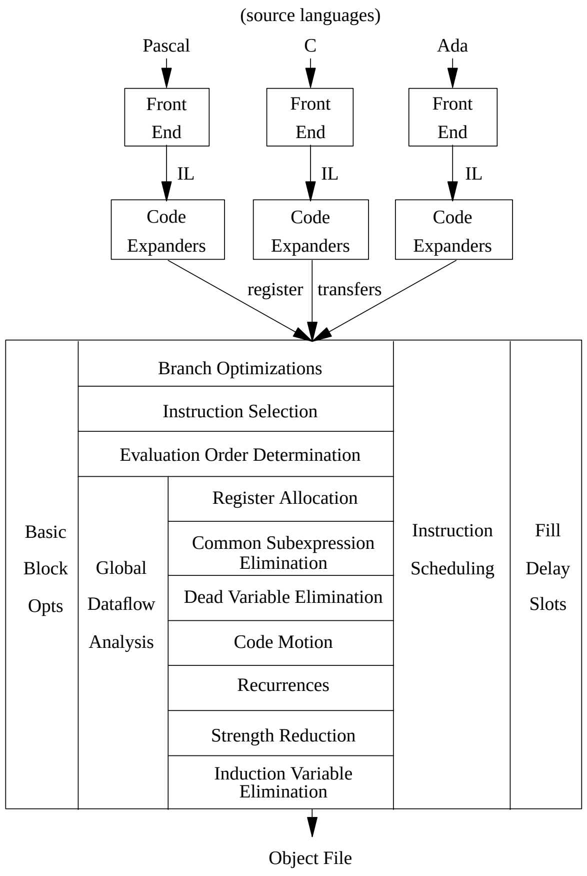


Figure 1: Compiler Structure

modified to identify each *change* to this data structure and to denote each serial sequence of changes that preserves the meaning of the compiled program. In this paper these sequences of changes are referred to as *transformations*.

ISOLATING OPTIMIZATION ERRORS

Testing is often the most time-consuming component of retargeting a back end of an optimizing compiler to a new machine. Much of the time spent during testing involves isolating errors in an optimizer to determine why specific programs do not execute correctly. The compiler writer must not only determine what was produced incorrectly in the erroneous program, but also at what point it was produced within the compiler.

Traditional Isolation of Optimization Errors

Traditionally, the compiler writer initially attempts to determine the specific instruction (or instructions) generated by the compiler that causes the compiled program to execute incorrectly. One could first isolate a function that contains incorrect instructions. This is accomplished by compiling some functions with optimizations and other functions without optimizations and executing the program. If the program executes correctly, then the compiler writer knows the problem is in the set of functions that were not compiled with optimizations. Otherwise, the compiler writer assumes the problem is in the set of functions that were compiled with optimizations. The compiler writer continues to narrow down the set of functions that could contain an error until the function with incorrect code is isolated.

The compiler writer can then compile the isolated function with and without various optimizations until finding the additional optimization being applied to the function that causes the compiled program to execute incorrectly. At this point the compiler writer can visually inspect the differences between the two assembly versions of the functions in an attempt to determine the instruction or instructions that appear to cause incorrect behavior.

Given that the compiler writer is able to conclude that a specific instruction within a function causes the compiled program to produce incorrect results, finding the reason why the compiler produced this instruction is the next task. Identifying the optimization that produces the problem may be difficult since the instruction may only be produced when a specific combination of optimizations are performed. Even if the compiler writer happens to correctly identify the optimization that produces the problem, the point in the compiler when the incorrect transformation occurs still has to be found. A specific optimization may be applied to hundreds of RTLs in *vpo* when compiling a function.

While these techniques may sometimes be effective, they are also quite tedious. Furthermore, some compiler optimizations that reduce execution time while increasing code size are becoming more popular. These optimizations include subprogram inlining [DaH88], loop unrolling [HeP90], and replicating code to avoid unconditional jumps [MuW92]. When these types of optimizations are applied, a single function may expand into several thousands lines of assembly code. Visual inspection of such functions to discover incorrect instructions is impractical. Using traditional methods to identify the point in the compiler that causes an invalid instruction to be produced in these functions may also be unrealistic.

Automatic Isolation of Optimization Errors

A tool, called *vpoiso*, has been developed to automatically isolate the first transformation that causes incorrect output from the execution of the compiled program. First, the optimization phases applied by *vpo* were classified as one of two types, *necessary* or *improving*. A *necessary* phase is required to produce code that can be compiled and executed. These phases include assigning pseudo registers to hardware registers and fixing the entry and exit points of a function to manage the run-time stack. All phases that are not required are referred to as *improving*. Only *improving* transformations that cause incorrect output can be isolated by *vpoiso*.

The *vpoiso* tool performs a binary search that relies on the ability to limit the number of *improving* transformations applied to a specified function. In the routine that is invoked when the end of a transformation is identified, *vpo* checks a counter to determine if the specified limit to the number of *improving* transformations has been reached. Unfortunately, *vpo* can be in quite deeply nested routines and logic at a point when a transformation has been completed. To check a status flag at each of the points after completing a transformation to prevent further *improving* transformations would have required significant modifications to *vpo*. Therefore, *vpo* was modified using the `setjmp` and `longjmp` functions. The `setjmp` function saves the values of all registers, including the stack pointer and program counter, into a environment buffer. Immediately before each call to a highest level optimization routine, a call to `setjmp` is performed. The `longjmp` function uses the environment buffer to restore the values of the registers, which has the effect of transferring control back to the point immediately following the call to `setjmp`.[1] A call to `longjmp` is executed at the point when the specified limit of the number of transformations was reached. Execution then resumes after the call to `setjmp` and only the remaining *necessary* transformations are applied. The following code illustrates these modifications to *vpo*.

```
/* Within a high level routine in VPO. */
...
/* Save current environment. */
setjmp(my_env);

/* If more optimizations allowed then
   perform register coloring. */
if (moreopts)
   color();
...
/* Within the routine that is invoked when
   the end of a transformation is identified. */
...
/* If reached limit, then set flag to not allow any
   more optimizations and restore environment. */
if (maxtrans == opttransnum) {
   moreopts = FALSE;
   longjmp(my_env, 1);
   }
...
```

[1] One has to ensure that no local variable is modified between invoking `setjmp` and the call to the optimization routine. If the variable was allocated to a register, then the variable's value at the point of invoking `setjmp` would be restored by the `longjmp` call.

The *vpoiso* tool is a C program which uses the C `system()` function to invoke various UNIX shell commands. First, *vpoiso* reads in a file of information indicating how to isolate an error within a program. This information includes the basenames of the files that are output from the code expander (or input to *vpo*), link and execute commands, maximum cpu time in seconds allowed for execution (i.e. in case an error causes the program to not terminate), desired and actual output filenames, compilation flags (user can specify some or all optimizations to be performed), and strings indicating lines to disregard (i.e. the output contains information dependent on time). For instance, a manufactured error was inserted during the compilation of the program *yacc*. To isolate the error, the following information was input to *vpoiso*.

```
cexfiles: y1 y2 y3 y4 #
link command: cc -o yacc y1.o y2.o y3.o y4.o
execute command: yacc cgram.y
maximum time: 15
desired output file: yacc.out
actual output file: y.tab.c
compilation flags: LVGOCMSFA
disregard strings:
```

After reading this information *vpoiso* has to determine if an incorrect transformation can be isolated. Thus, *vpoiso* invokes *vpo* for each file to be compiled with an option to record the number of *improving* transformations required for each function, the function name, and the basename of the file in which the function resides. The *vpoiso* tool then links and executes the program using the specified commands. If the actual output is the same as the desired output, then *vpoiso* quits after informing the user that it could find no error when all optimizations were applied to each function in the program. Otherwise, *vpoiso* reads the information generated during the previous compilation and invokes *vpo* for each file to be compiled indicating that no *improving* transformations are to be performed. Again, *vpoiso* issues commands to link and execute the program. If the actual output differs from the desired output, then *vpoiso* exits after informing the user that the error must be caused by the front end, code expander, or a *necessary* transformation in *vpo*.

If *vpoiso* determined that the error can be isolated, then it performs a binary search to isolate the first incorrect transformation. The search is depicted in the following pseudocode.

```
lastmin = 0;
lastmax = total number of improving transformations
while (lastmax - lastmin > 0) {
   midnum = (lastmin + lastmax)/2;
   recompile program with only the first midnum
     transformations performed
   remove actual output file
   link and execute program
   if (actual output file == desired output file)
      lastmin = midnum+1;
   else
      lastmax = midnum;
   }
if (last result was incorrect)
   badtrans = midnum;
else
   badtrans = midnum+1;
```

At this point *vpoiso* prints the name of the function containing the first incorrect transformation and the incorrect transformation number within that function.[2] The user can then set a breakpoint in the source-level debugger that is executing *vpo* that will stop when the transformation with that number is encountered. The routine in *vpo* that is invoked when the start of a transformation is identified contains the following portion of code.

```
 ...
 if (opttransnum == breakopttransnum)
    fprintf(stderr,
            "improving trans breakpoint\n");
 ...
```

The user assigns the displayed transformation number to the `breakopttransnum` variable, sets a breakpoint at the line where the message is printed, and reexecutes *vpo*. Thus, using this feature, the compiler writer can quickly access the point during the compilation that precedes the incorrect *improving* transformation.

The *vpoiso* tool avoids unnecessary recompilation to reduce the isolation time. If the transformations on functions in a file are not within the current search range that could contain the first incorrect transformation, then this file is not recompiled. Recompilation of a file is also unnecessary when all the functions in the file would be compiled with the same number of transformations as in the previous compilation. In addition, if a function is in a file that needs to be compiled and it is not within the current search range, then the function is compiled with no optimizations to decrease the compilation time.

[2] The *vpoiso* tool is only guaranteed to find the first transformation that causes incorrect output. It is possible that a previous transformation was invalid and the isolated transformation was the first transformation that moves invalid instructions into a path that was executed. This situation has not occurred when testing *vpoiso* with manufactured or actual errors.

To illustrate the performance of *vpoiso*, the results for finding a manufactured error inserted into the compilation of the *yacc* program is described. There were a total of 13,955 *improving* transformations applied with the complete optimization of *yacc*. The *vpoiso* tool required 16 compilations/executions of *yacc* and a little under 10 minutes of wall-clock time to correctly isolate the erroneous transformation on a Sun SPARC IPC.[3] A log of the actions performed by *vpoiso* when isolating this error is given in Appendix I.

SUPPORTING THE ANALYSIS OF OPTIMIZATION ERRORS

After isolating the incorrect transformation, the compiler writer still has to determine why the transformation was produced. A graphical optimization viewer, called *xvpodb*, was developed to assist in the analysis of optimization errors in *vpo*. Figure 2 depicts how viewing optimizations will typically be accomplished. One process is *vpo* executing under the control of a source-level debugger and the other is *xvpodb*. By invoking *vpo* with a source-level debugger, the user can control the execution of the optimizer and perform such tasks as setting breakpoints and printing the values of variables that are local to routines within the compiler. Before performing any optimizations for the current function, *vpo* will pass a set of messages to *xvpodb* that describe the initial set of RTLs produced by the code expander. After receiving these messages, *xvpodb* will display this initial set to the user. Subsequently, information about each transformation to the RTLs will be passed to *xvpodb*. The user can view these transformations serially or at specified breakpoints, either in the forward (showing the transformations being applied) or reverse (showing them being undone) direction.

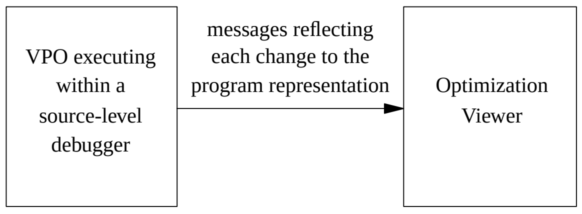
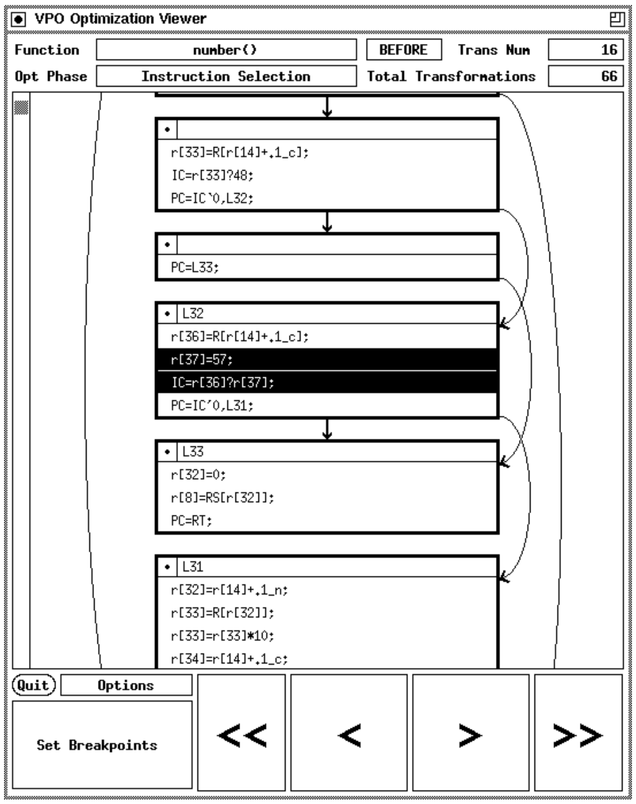

VPO executing within a source-level debugger
messages reflecting each change to the program representation
Optimization Viewer

Figure 2: Typical Use of the Optimization Viewer

[3] Note that the first 2 executions were only performed to verify that an incorrect transformation could be isolated.

XVPODB User Interface

Figure 3 depicts the interface for the main window of the optimization viewer. The large display in the center of the window contains a scrollable view of the RTLs (describing SPARC instructions) in the current function being compiled. A portion of the RTL structure is displayed at a given point during the compilation. Each basic block is represented by enclosing its RTLs within a rectangle. The basic blocks are displayed in the order that they would appear when generated as assembly instructions. Transfers of control between basic blocks are depicted using arcs. Forward transfers of control are depicted to the right of the blocks and backward transfers of control are depicted on the left. Arcs are used to enable a user to distinguish overlapping control transitions. The RTLs are displayed in human readable form (not the encoded internal format used by *vpo*). Highlighted RTLs are those that are affected by the current transformation being viewed.



```
Function   number()                 BEFORE   Trans Num              16
Opt Phase  Instruction Selection    Total Transformations           66

•
r[33]=R[r[14]+.1_c];
IC=r[33]?48;
PC=IC`0,L32;

•
PC=L33;

• L32
r[36]=R[r[14]+.1_c];
r[37]=57;
IC=r[36]?r[37];
PC=IC´0,L31;

• L33
r[32]=0;
r[8]=RS[r[32]];
PC=RT;

• L31
r[32]=r[14]+.1_n;
r[33]=R[r[32]];
r[33]=r[33]*10;
r[34]=r[14]+.1_c;
```

Figure 3: Main Window for the Optimization Viewer

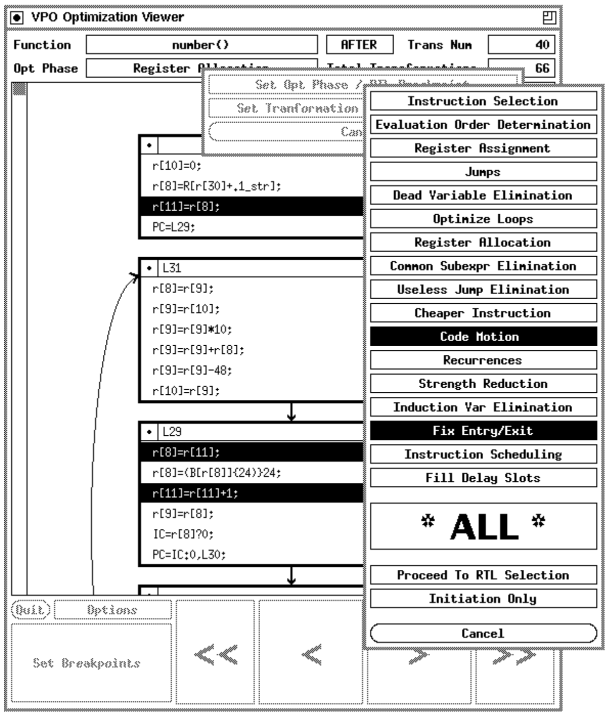
The labels at the top of the window display the name of the function being examined, the optimization phase in which the current transformation is performed, the unique number of this transformation, and the total number of transformations that have been received for this function. Also shown is the current state, which in this example is BEFORE. The BEFORE state indicates that this transformation has not yet been applied to the RTLs.

The buttons at the bottom of the window represent the different options available to the user of *xvpodb* via mouse clicks. Four of these buttons resemble the controls on an audio tape player (including auto-reverse). When the **Step Forward** (>) and **Step Backward** (<) buttons are clicked, *xvpodb* displays the next or previous transformation, respectively. The user can view a full transformation with two clicks of the mouse. In the case of >, the first click causes the BEFORE state of the RTLs to be displayed. The RTLs that will be deleted or modified are highlighted. Figure 3 shows two RTLs about to be combined during an instruction selection transformation. A second click of > causes the viewer to display the AFTER state. The highlighted RTLs will be those that were just modified or inserted. Selecting > again will display the BEFORE state of the next transformation. After each selection the RTL display is automatically scrolled to a position where highlighted RTLs are visible. < works similarly, except the display of the AFTER state precedes the BEFORE. The user can apply and reverse (or vice-versa) a transformation as many times as desired, which is very useful for grasping the full effect of a complicated transformation. The **Continue Forward** (>>) and **Continue Backward** (<<) buttons are similar to > and <, except they continue to apply transformations until a breakpoint is reached.

There are two main types of breakpoints in *xvpodb*. The first and simplest is a *transformation number* breakpoint. The user enters a transformation number or numbers, and *xvpodb* will break when encountering the beginning or end of those transformations (depending on whether >> or << was selected). Since *vpo* knows the number of each transformation it sends to *xvpodb*, this provides a convenient way to coordinate breakpoints in the compiler and the viewer. In addition, this type of breakpoint allows a user to quickly view an invalid transformation identified by *vpoiso*.

The second type of breakpoint is more general. Initially, the user selects some or all optimization phases from a toggle menu. After completing this selection the user has two choices. One choice is to have *xvpodb* stop whenever one of the selected phases is encountered (at the beginning or end of the phase, depending on the direction of the viewing). Alternately, the user can choose a set of RTLs to associate with this breakpoint. The viewer will stop whenever any of the selected RTLs is changed during any of the selected phases. Thus, breakpoints can be set on specific optimization phases, specific RTLs, or any combination of both. Figure 4 shows an example of setting this type of breakpoint. The user has selected the optimization phases and can now choose **Initiation Only** or **Proceed to RTL Selection**.

The options menu allows selection of less commonly used features of the viewer. Currently, the options menu includes buttons to return the RTLs to the inital state (before any transformations) and to apply all transformations (thus showing the fully optimized set of RTLs). Both of these functions skip all breakpoints. In addition, there is a button that allows one to select to proceed to the next function within the file being compiled.



VPO Optimization Viewer

Function: number() AFTER Trans Num: 40

Opt Phase: Register Allocation Total Transformations: 66

```
•
r[10]=0;
r[8]=R[r[30]+.1_str];
r[11]=r[8];
PC=L29;

• L31
r[8]=r[9];
r[9]=r[10];
r[9]=r[9]*10;
r[9]=r[9]+r[8];
r[9]=r[9]-48;
r[10]=r[9];

• L29
r[8]=r[11];
r[8]=(B[r[8]]{24)}24;
r[11]=r[11]+1;
r[9]=r[8];
IC=r[8]?0;
PC=IC:0,L30;
```

Figure 4: XVPODB Breakpoint Selections

To illustrate the power of *xvpodb*, the process of using the viewer to understand how a particular RTL was generated is described. Initially, the user selects the **Apply All Transformations** button (located in the options menu), which will cause the viewer to display the completely optimized set of RTLs. Next, the user sets a breakpoint to cause the viewer to stop on any change to the desired RTL. At this point, by successively clicking << the user can view each transformation involving this RTL being undone. The transformation can be analyzed by clicking < and > as many times as necessary. The user can click << until the transformations that produced the particular RTL are understood. If desired, the user could click >> to view each transformation being reapplied to the RTL. If an invalid instruction has been identified and a tool like *vpoiso* is unavailable, then this technique can also be used to visually isolate the incorrect transformation.

Implementing XVPODB

Each message from *vpo* to *xvpodb* reflects at most a single change to the data structure containing the RTLs. The list of message types from *vpo* to *xvpodb* include:

begin function	modify basic block label
end function	insert RTL
start optimization phase	delete RTL
end optimization phase	move RTL
start transformation	modify RTL
end transformation	modify RTL dead register list
create new basic block	modify controlflow successor
free up basic block	modify output position successor

Separately displaying this level of detail to the user would be excessive. For instance, in Figure 3 the two highlighted RTLs are about to be combined together as a result of an instruction selection transformation. The first highlighted RTL will be deleted and the second highlighted RTL will be modified. In fact, showing these two changes as separate steps would be confusing to the user since the function would not appear equivalent after a single change. Also, the order in which these two changes are applied to the RTL data structure is unimportant. Therefore, the effects of all changes that are enclosed between *start* and *end transformation* messages are displayed simultaneously.

Providing reverse viewing affected the design of the optimization viewer. First, the information within each message has to be retained after being processed. Whenever a new message is received from the compiler, a new node containing the information about the message is added to the end of a doubly linked list. When the user chooses to proceed to the next function, the nodes in the list that comprise the function are freed. A pointer to this list indicates the current state of the displayed RTLs. All transformations preceding and none following this pointer have been applied to the initial set of RTLs. If the user steps or continues in either direction (<, >, <<, >>), then the nodes of the list are traversed (and the changes are applied) in the appropriate direction until the desired transformation is reached. Another implication of reverse viewing is that all information needed to reverse a change must be retained. For example, when reversing a *delete RTL* change, *xvpodb* must regenerate the deleted RTL and insert it into the RTL display structure.

The optimization viewer is quite easy to retarget to versions of *vpo* for other architectures. The code comprising *xvpodb* itself is machine-independent. The messages passed from *vpo* to *xvpodb* are accomplished via system calls using UNIX sockets. The optimization viewer was developed in X-Windows. As UNIX has become the most popular and portable operating system, X-Windows appears to be achieving the same goals as a graphical environment. A final feature that enhances portability is that the general form of RTLs is machine-independent. This allows algorithms that perform transformations on the RTLs to be implemented in machine-independent code. Since most of the transformations on RTLs in *vpo* are accomplished in a machine-independent fashion, there are few additional changes required due to the addition of *xvpodb* when retargeting *vpo*.

The message passing paradigm provides the user with the option of executing *vpo* and *xvpodb* on two different machines. Due to the use of X-Windows, the user has the option to view the output windows of the two programs on yet another machine. Thus, one can use the resources of up to three machines to speed up the debugging process.

APPLYING THE TECHNIQUES TO OTHER OPTIMIZERS

There are certain features of *vpo* that simplified the development of the tools to isolate and analyze optimization errors. Performing code generation before all optimizations allows *vpoiso* to accurately determine that a code generation error was not caused by the optimizer. If code generation was performed after optimizations, then a code generation error may only occur when the intermediate representation is in a specific form (e.g. a particular instance of a dag). When the number of optimizations performed is reduced, this specific form may not appear. In this situation it would be difficult to have a tool automatically determine that the error was not caused by the optimizer. The structure of the *vpo* optimizer also made it easy to stop performing improving transformations at any point during a compilation. This ability may not be as straightforward to implement in other optimizers.

Certain features of *vpo* also simplified the development of *xvpodb*. One such feature is that all phases of the optimizer manipulate RTLs. Because there is only one type of data structure to represent program information, only one algorithm needs to be developed to accept changes and produce a view of the data structure. Since each RTL represents a legal machine instruction (and can be decoded into a very readable format), the effect of a modification to the set of RTLs comprising a function is simple to grasp. In contrast, most conventional compiler systems generate code after optimizations. In these systems the effect of a modification on the final code that will be generated may not be obvious. Finally, RTLs can easily be displayed in a linear fashion at any point during the optimization. Displaying the representation of a program being optimized that has a dag or tree intermediate form would be much more difficult.

COMPARISON WITH RELATED WORK

A tool known as *bugfind* [CaD90] was developed to assist in the debugging of optimizing compilers. The *bugfind* tool attempts to determine the highest optimization level at which each file within a program can be compiled and produce correct output. To isolate a function that was not optimized correctly, one has to place each function within the program in a separate file. The *bugfind* tool uses the *make* facility in Unix and is generalized enough to work with different compilers.

While *bugfind* and *vpoiso* share some similar ideas, there are also considerable differences. Both *bugfind* and *vpoiso* use a binary search technique to isolate optimization errors. The *vpoiso* tool finds not only the failing module, but also the first transformation within a function that causes incorrect results. The transformation number can be used to view the transformation in *xvpodb* and access the point in *vpo* when the transformation is about to be applied. This finer level of isolating errors is important when optimization errors occur in large functions or code size increasing transformations are performed. Unlike *bugfind*, *vpoiso* can only isolate errors within *vpo*. However, the techniques *vpoiso* uses can probably be applied with other optimizers.

The UW Illustrated Compiler [AHY88], also known as *icomp*, graphically displays its control and data structures during the compilation of a program. A feature called hookpoints is used to specify points in the compiler to update the windows that have changed since the last hookpoint was executed. By specifying hookpoints and breakpoints in the compiler a user can control the rate at which views are displayed during a compilation. The *icomp* compiler has been used by undergraduate compiler classes to illustrate the compilation process.

There are many differences between *icomp* and *xvpodb*. The purpose for developing the *icomp* compiler was for use as a teaching tool in an undergraduate compiler class. The main purpose for constructing *xvpodb* is to assist a compiler writer when retargeting the *vpo* compiler to a new machine. The *xvpodb* tool can also be used as a teaching tool in a compiler class to illustrate various compiler optimizations. The source programs compiled by *icomp* are written in a subset of Pascal called PL/0. The *vpo* back end currently interfaces with a front end called *vpcc* (Very Portable C compiler) that supports the complete C language. The *icomp* compiler shows views of different portions of the compilation process which includes lexical analysis, parsing, semantic analysis, and code generation. No optimizations are performed by the compiler. In contrast, *xvpodb* displays the effects of optimizations exclusively. The *icomp* compiler allows breakpoints and hookpoints to be set at different locations in the source code of the compiler. It does not have the ability to stop when a user-specified portion of a view is updated. The *xvpodb* tool allows breakpoints to be set associated with updates to a specific portion of the information representing a function. The *icomp* compiler was written in Interlisp-D to access facilities in the language for implementing hookpoints and producing graphical displays. Both the *vpo* compiler and *xvpodb* are written in C. Thus, optimization viewers could be developed for other existing compilers written in conventional programming languages using the techniques to implement *xvpodb*. Finally, *icomp* does not allow reverse viewing of transformations. It was stated, "*icomp* cannot be run in reverse because of the complexity of implementing such a feature." Reverse viewing was feasible in *xvpodb* since the information about a function is represented in only a single type of data structure. By retaining information about each change to this data structure the ability to undo transformations was accomplished without excessive complexity.

CONCLUSIONS

The tools described in this paper have several important benefits when retargeting the back end of a compiler. A tool with the ability to isolate incorrect transformations automatically, such as *vpoiso*, may prove to be invaluable, particularly when employing code size increasing optimizations. An optimization viewer, such as *xvpodb*, is also quite useful. Displaying the program representation at any given point during the optimization of a function, stopping at breakpoints associated with the generated code, and reverse viewing of transformations are all helpful features for analyzing problems with an optimizer. Compilers can also be used to guide instruction set design to determine if proposed architectural features can be exploited [DaW91]. Decreasing the time to retarget a compiler to a proposed architecture would also decrease the time required to design and develop a new machine.

Additionally, *xvpodb* can be used as a teaching aid for advanced compiler classes. Many recently introduced machines require sophisticated compiler optimizations to exploit their architectural features. Advanced compiler courses that present techniques to perform these types of optimizations may soon become more common. A tool that would allow a student to interactively visualize the effect of each transformation would be quite useful in illustrating these optimizations.

ACKNOWLEDGEMENTS

The authors thank Jack Davidson for allowing *vpo* to be used for this research. The identification of each change and the sequences of changes that comprised the transformations in *vpo* was simplified by the high quality of coding of *vpo*, which in a very large part is due to the efforts of Manuel Benitez. Indrakshi Ray tested *vpoiso* and made several suggestions that resulted in an improved tool. Frank Mueller provided many helpful comments on an earlier draft of this paper.

REFERENCES

[AHY88] K. Andrews, R. R. Henry, and W. K. Yamamoto, "Design and Implementation of the UW Illustrated Compiler," *Proceedings of the SIGPLAN '88 Symposium on Programming Language Design and Implementation*, pp. 105-114 (June 1988).

[BeD88] 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).

[CaD90] J. M. Caron and P. A. Darnell, "Bugfind: A Tool for Debugging Optimizing Compilers," *Sigplan Notices* **25**(1) pp. 17-22 (January 1990).

[DaH88] J. Davidson and A. Holler, "A Study of a C Function Inliner," *Software—Practice & Experience* **18**(8) pp. 775-790 (August 1988).

[Dav86] J. W. Davidson, "A Retargetable Instruction Reorganizer," *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pp. 234-241 (June 1986).

[DaF84] J. W. Davidson and C. W. Fraser, "Code Selection through Object Code Optimization," *Transactions on Programming Languages and Systems* **6**(4) pp. 7-32 (October 1984).

[DaW91] J. W. Davidson and D. B. Whalley, "A Design Environment for Addressing Architecture and Compiler Interactions," *Microprocessors and Microsystems* **15**(9) pp. 459-472 (November 1991).

[HeP90] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Mateo, CA (1990).

[MuW92] F. Mueller and D. B. Whalley, "Avoiding Unconditional Jumps by Code Replication," *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 322-330 (June 1992).

[WJW75] W. Wulf, R. K. Johnsson, C. B. Weinstock, S. O. Hobbs, and C. M. Geschke, *The Design of an Optimizing Compiler*, American Elsevier, New York, NY (1975).

APPENDIX I

Below is the log from *vpoiso* when isolating the manufactured error inserted during transformation 500 in the routine *setup* in the *yacc* program.

```
START TIME: 18:02:22

compiling program to check if works with all improving transformations
compiling y1.cex
compiling y2.cex
compiling y3.cex
compiling y4.cex
linking program
executing program
As expected, the output was incorrect.

13955 total improving transformations for program
  y1: main: 1-42                 y2: fdtype: 6913-6966
  y1: others: 43-782             y2: chfind: 6967-7098
  y1: chcopy: 783-807            y2: cpyunion: 7099-7409
  y1: writem: 808-945            y2: cpycode: 7410-7848
  y1: symnam: 946-980            y2: skipcom: 7849-7982
  y1: summary: 981-1160          y2: cpyact: 7983-9734
  y1: error: 1161-1196           y3: output: 9735-10169
  y1: aryfil: 1197-1237          y3: apack: 10170-10473
  y1: setunion: 1238-1287        y3: go2out: 10474-10762
  y1: prlook: 1288-1368          y3: go2gen: 10763-11128
  y1: cpres: 1369-1559           y3: precftn: 11129-11257
  y1: cpfir: 1560-2016           y3: wract: 11258-11647
  y1: state: 2017-2449           y3: wrstate: 11648-12006
  y1: putitem: 2450-2523         y3: wdef: 12007-12019
  y1: cempty: 2524-2904          y3: warray: 12020-12121
  y1: stagen: 2905-3291          y3: hideprod: 12122-12241
  y1: closure: 3292-3873         y4: callopt: 12242-12911
  y1: flset: 3874-3989           y4: gin: 12912-13126
  y2: setup: 3990-5413           y4: stin: 13127-13491
  y2: finact: 5414-5429          y4: nxti: 13492-13648
  y2: defin: 5430-5689           y4: osummary: 13649-13724
  y2: defout: 5690-5877          y4: aoutput: 13725-13760
  y2: cstash: 5878-5922          y4: arout: 13761-13862
  y2: gettok: 5923-6912          y4: gtnm: 13863-13955

compiling program to check if works with no improving transformations
compiling y1.cex
compiling y2.cex
compiling y3.cex
compiling y4.cex
linking program
executing program
As expected, the output was correct.

starting binary search to isolate error within 13955 transformations

error within main to gtnm (transformation 1 to 13955)
compiling program: applying transformations 1 to 6978
compiling y1.cex
compiling y2.cex
stopped optimization of chfind after 11 improving transformations
linking program
executing program
execution was incorrect

error within main to chfind (transformation 1 to 6978)
compiling program: applying transformations 1 to 3489
compiling y1.cex
stopped optimization of closure after 197 improving transformations
compiling y2.cex
linking program
executing program
execution was correct

error within closure to chfind (transformation 3490 to 6978)
compiling program: applying transformations 3490 to 5234
compiling y1.cex
compiling y2.cex
stopped optimization of setup after 1244 improving transformations
linking program
executing program
execution was incorrect

error within closure to setup (transformation 3490 to 5234)
compiling program: applying transformations 3490 to 4362
compiling y2.cex
stopped optimization of setup after 372 improving transformations
linking program
executing program
execution was correct

incorrect transformation isolated to function setup

error within setup (transformation 4363 to 5234)
compiling program: applying transformations 4363 to 4798
compiling y2.cex
stopped optimization of setup after 808 improving transformations
linking program
executing program
execution was incorrect

error within setup (transformation 4363 to 4798)
compiling program: applying transformations 4363 to 4580
compiling y2.cex
stopped optimization of setup after 590 improving transformations
linking program
executing program
execution was incorrect

error within setup (transformation 4363 to 4580)
compiling program: applying transformations 4363 to 4471
compiling y2.cex
stopped optimization of setup after 481 improving transformations
linking program
executing program
execution was correct

error within setup (transformation 4472 to 4580)
compiling program: applying transformations 4472 to 4526
compiling y2.cex
stopped optimization of setup after 536 improving transformations
linking program
executing program
execution was incorrect
```

```
error within setup (transformation 4472 to 4526)
compiling program: applying transformations 4472 to 4499
compiling y2.cex
stopped optimization of setup after 509 improving transformations
linking program
executing program
execution was incorrect

error within setup (transformation 4472 to 4499)
compiling program: applying transformations 4472 to 4485
compiling y2.cex
stopped optimization of setup after 495 improving transformations
linking program
executing program
execution was correct

error within setup (transformation 4486 to 4499)
compiling program: applying transformations 4486 to 4492
compiling y2.cex
stopped optimization of setup after 502 improving transformations
linking program
executing program
execution was incorrect

error within setup (transformation 4486 to 4492)
compiling program: applying transformations 4486 to 4489
compiling y2.cex
stopped optimization of setup after 499 improving transformations
linking program
executing program
execution was correct

error within setup (transformation 4490 to 4492)
compiling program: applying transformations 4490 to 4491
compiling y2.cex
stopped optimization of setup after 501 improving transformations
linking program
executing program
execution was incorrect

error within setup (transformation 4490 to 4491)
compiling program: applying transformations 4490 to 4490
compiling y2.cex
stopped optimization of setup after 500 improving transformations
linking program
executing program
execution was incorrect

incorrect transformation isolated to optimization 500 in function setup

STOP TIME: 18:12:18
```