THE FLORIDA STATE UNIVERSITY
COLLEGE OF ARTS AND SCIENCES

GRAPHICAL VISUALIZATION OF COMPILER OPTIMIZATIONS

By

MICKEY BOYD

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

Degree Awarded:

Summer Semester, 1993

The members of the Committee approve the thesis of Mickey Boyd defended on July 19, 1993.

David Whalley
Professor Directing Thesis

Theodore Baker
Committee Member

Gregory Riccardi
Committee Member

This work is dedicated to three very special people: My parents, Don and Toki, for their love and understanding throughout my existence, and to my beloved Sheryl, for being kind enough to share her life with me.

Acknowledgements

I would like to offer special thanks to my advisor, Dr. David Whalley, for his support and insight during this research. Also, I offer thanks to committee members Dr. Ted Baker and Dr. Greg Riccardi, for their review of and subsequent suggestions concerning this document.

Contents

List of Tables

List of Figures

Abstract

This document describes *xvpodb*, a visualization tool developed to support the analysis of optimizations performed by the *vpo* optimizer. It was designed to aid and simplify the process of retargeting *vpo* to a new machine architecture. The tool is a graphical viewer/debugger that can display each change made by the optimizer to the generated instructions during an actual compilation. The format used to display these changes is simple and intuitive. The information and insight such visualization provides can greatly simplify debugging of the optimizer. Unique features of *xvpodb* include reverse viewing (or "undoing") of changes, and the ability to stop at breakpoints associated with the generated instructions. In addition to facilitating debugging, the tool also simplifies experimentation with new optimization techniques. Lastly, *xvpodb* is a valuable teaching aid in compiler courses, as it allows students to see a precise depiction of what happens to generated instructions as they are optimized.

Chapter 1

Introduction

To increase portability compilers are often split into two parts, a front end and a back end. The front end processes a high-level language program and emits intermediate code. The back end processes the intermediate code and generates assembly instructions for the target machine architecture. Thus, the front end is dependent on the source language and the back end is dependent on the instruction set for the target machine. Retargeting such a compiler for a new machine requires creating a new back end. While the time and effort 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 at which new machines are being introduced has increased. Also, there is an increasing reliance on compilers to perform highly sophisticated optimizations that 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. Large amounts of time are spent attempting to determine why incorrect code is generated, or why optimizations cannot be applied for specific programs. Many back ends store information about the program being compiled in an encoded internal format, which exacerbates the problem. While such formats consume less memory and allow optimizations to occur more rapidly, they also increase the difficulty of analyzing specific problems within the code.

The goal of the research described in this document is to simplify the task of debugging the *vpo* optimizer. A visualization tool was developed that allows the programmer to view each optimization performed by the optimizer in a way that is both flexible and easy to understand.

One could obtain the same information using a standard symbolic debugger to examine internal data structures by hand. However, this is both labor intensive and prone to human error. The abstract, yet precise way the optimizations are presented by the viewer allows the programmer to concentrate on what the optimizer is doing to the instructions, rather than struggling with inadequate debugging tools. Also, this debugging tool recognizes the temporal element of a compilation. This is particularly important with *vpo*, as it can reapply optimization phases many times during a compilation. The viewer identifies not only what changes occurred, but also when they occurred during the compilation (relative to other changes). This greatly simplifies the eradication of bugs that only manifest with a certain cascading set of optimization phases and instructions.

Selective viewing of the optimizations performed by *vpo* is accomplished using breakpoints. The breakpoint paradigm used is simple yet powerful, allowing the programmer to quickly focus in on the desired portion of the compilation.

Finally, this debugging tool has the ability to reverse the effects of any or all optimizations made during a compilation. With this feature, the programmer need not be concerned about executing the compiler “one step too far,” and being forced to reexecute the current test.

Chapter 2

Overview of the Compiler

2.1 General Overview

The tool described in this paper supports 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 2.1 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 sub-phases of the column may be executed in an arbitrary order. IL is the Intermediate Language generated by a front end. 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. The RTLs are stored in a data structure that also contains information about the order and controlflow of the RTLs within a function. One advantage of using RTLs as the sole intermediate representation is that many phase ordering problems are eliminated. Most opti-

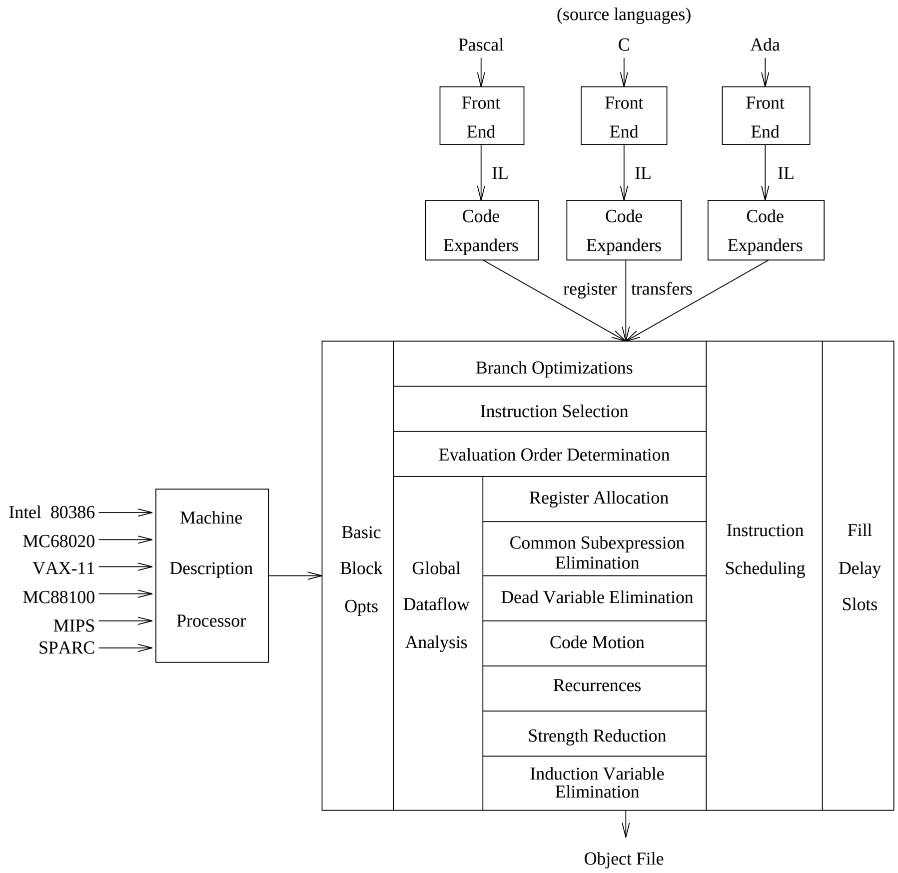

Figure 2.1: Compiler Structure

mizations can be invoked in any order and are allowed to iterate until no further improvements can be found. Another advantage is that since each RTL represents a legal machine instruction, the effect of a modification to the set of RTLs comprising a function is relatively simple to grasp. In contrast, most conventional compiler systems generate code after optimizations. In these systems the effect of a modification may not be trivial to deduce.

2.2 Enhancements for Xvpodb

The *vpo* optimizer was modified to identify any *change* to the RTL data structure. A serial sequence of changes that preserve the meaning of the compiled program is referred to as a *transformation*. This terminology is used throughout this paper. The optimizer was also modified to send messages containing descriptions of these changes, as well as notification of the optimization phase and transformation number in which the changes took place. The exclusive use of RTLs as the intermediate representation greatly simplified the design and implementation of *xvpodb*. Because there is only one type of data structure representing program information, only one algorithm needed to be developed to process modification messages and produce a view of the data structure.

2.3 Vpoiso

During the development of *xvpodb*, another tool was created to assist programmers in debugging the *vpo* optimizer. The tool depends upon some of the enhancements made to *vpo* for *xvpodb*. This tool, called *vpoiso*, will automatically isolate the first transformation in a compilation that causes the program being compiled to produce incorrect output when executed [BoW93]. After performing such an isolation, *xvpodb* is ideal for viewing the transformation that is found to determine the cause of the error. Thus, these two tools compliment each other well, and together they form a powerful environment for facilitating the retargeting of *vpo*.

Chapter 3
Functional Description of Xvpodb

3.1 Method of Use

In typical usage, *vpo* would be executed from within a source level debugger, to allow the viewing of internal structures and to control execution. It would be invoked with the command line switches needed to activate the routines that build and pass messages to *xvpodb*, which would be running concurrently. The message passing paradigm chosen provides the user with the option of running *vpo* and *xvpodb* on two different machines. Due to the use of X Windows, the user also has the option to view the output windows of these two process groups on yet another machine. This allows the user to use the resources of up to three machines, thus speeding up the debugging cycle. Figure 3.1 illustrates this relationship, with the circles representing processes and the arrows showing communication channels between processes.

3.2 Message Passing

Before performing any optimizations, *vpo* will pass a set of messages that describe the initial state of all RTLs in the function currently being compiled. After receiving these messages, *xvpodb* will display this *initial set* to the user. Subsequently, descriptions of all changes to the RTLs will be passed to *xvpodb*, which stores them for later interpretation at the request of the user.

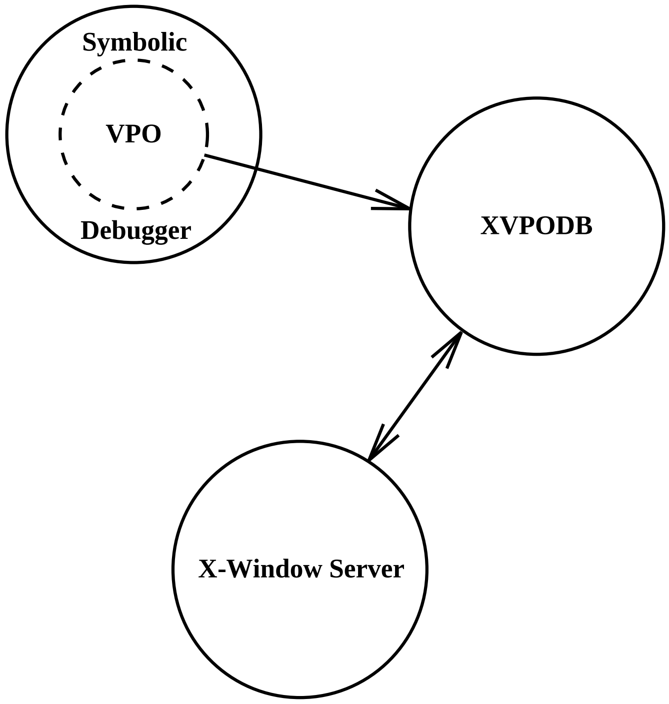

Figure 3.1: Typical Use of *xvpodb*

3.3 RTL Display Window

Figure 3.2 depicts *xvpodb* displaying an Instruction Selection transformation. The large middle section of the window displays a portion of the RTL structure (these RTLs are targeted to the SMCC SPARC architecture). The RTLs are shown contained in rectangles, which represent basic blocks. Transfers of control between basic blocks are depicted using arcs (the arrowheads on the arcs indicate direction of transfer). The basic blocks are shown in the order in which they will appear when generated as assembly instructions. The RTLs themselves are displayed in human readable form (not the encoded internal format used by *vpo*). This flowchart-like model is a common way of abstracting a program representation, and is used in many textbooks. The scrollbar to the left of the display area can be used to view

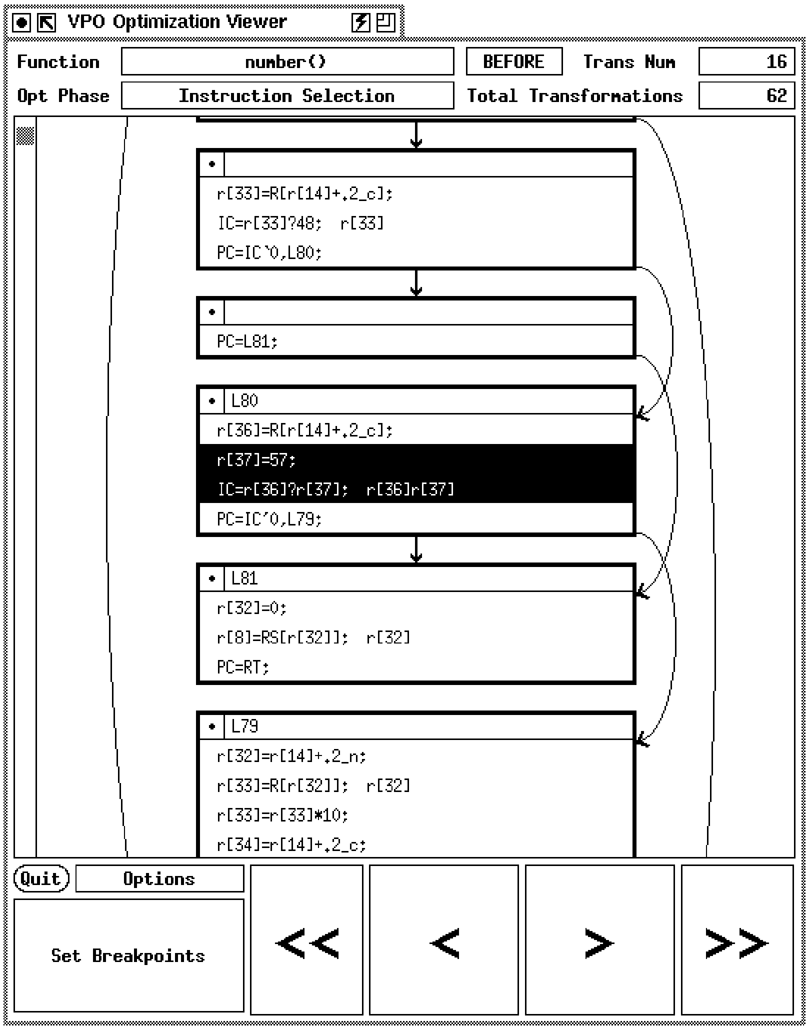

Figure 3.2: Main Display of *xvpodb*

any part of the set of RTLs currently being processed by the optimizer. Also, one can center the screen on a basic block's right or left pointer destination by clicking the right or left mouse buttons on the block header. This feature, called *magic jumping*, allows the user to follow an execution path without having to search for branch and jump destinations.

3.4 Informational Labels

The labels at the top of the window provide the name of the function being examined, the optimization phase in which the current transformation occurred, the unique number of this transformation, the total number of transformations that have been received for this function, and the state of the viewer. In Figure 3.2, the state is BEFORE transformation 16, which indicates that this is what the RTLs look like before this transformation is applied. If the state were AFTER, the middle display would show what the RTLs look like after transformation 16 is applied. These are the only two states in which the viewer can be.

3.5 Step and Continue Buttons

At the bottom of the window, there are four large buttons that resemble the controls on an audio cassette player (this is not a coincidence). The **>** and **<** buttons are, respectively, **Step Forward** and **Step Backward**. These buttons will display the effects of the next or previous transformations, again respectively. A full transformation is shown using two clicks of the left mouse button. In the case of the **Step Forward** button, the first click would show the BEFORE state of the RTLs. The RTLs to be affected are highlighted, and the screen is centered on these RTLs. Clicking again on the same button would perform the transformation, and cause the viewer to display the AFTER state of that transformation. The RTLs that were affected by the transformation would be updated and highlighted (they are not always the same RTLs seen in the BEFORE state, as in the case of an RTL insertion or deletion). The reverse case works similarly, except one views the AFTER state first, then the BEFORE. The user can apply and reverse (or vice-versa) a transformation as many times as needed, which is useful for grasping the full effect of a complicated transformation. The >> and << buttons are, respectively, **Continue Forward** and **Continue Backward**. They are similar to > and <, except they stop on breakpoints (as opposed to the next or previous transformation). Breakpoints are set with the **Set Breakpoints** button and menus, and are discussed in section 3.6.

The user can view the transformations serially or at specified breakpoints, either in the forward (showing the transformations being applied) or reverse (showing them being undone) direction. In other words, the user does not need to reexecute anything to view a previously applied transformation. The viewer can reverse the effects of any or all transformations with a few mouse clicks. This process does not affect the ability to interpret transformations in the forward direction. Thus, the user can view a transformation or set of transformations being applied and reversed as many times and in as many areas of the compilation as desired. Also, the programmer need not compile the entire function to be able to view transformations. The viewer will allow the user to see any transformations that have already been received from *vpo*. If desired, the programmer can step *vpo* through its optimization of a function one transformation at a time, thus being able both to view the graphical representation of the transformation and to study the actual data structures and source code in the optimizer that produced it. Information is provided by *xvpodb* that allows the programmer to easily locate the proper data structures. For example, the programmer can find the pointer address in *vpo* of the structure that contains an RTL or basic block simply by clicking on it in *xvpodb*.

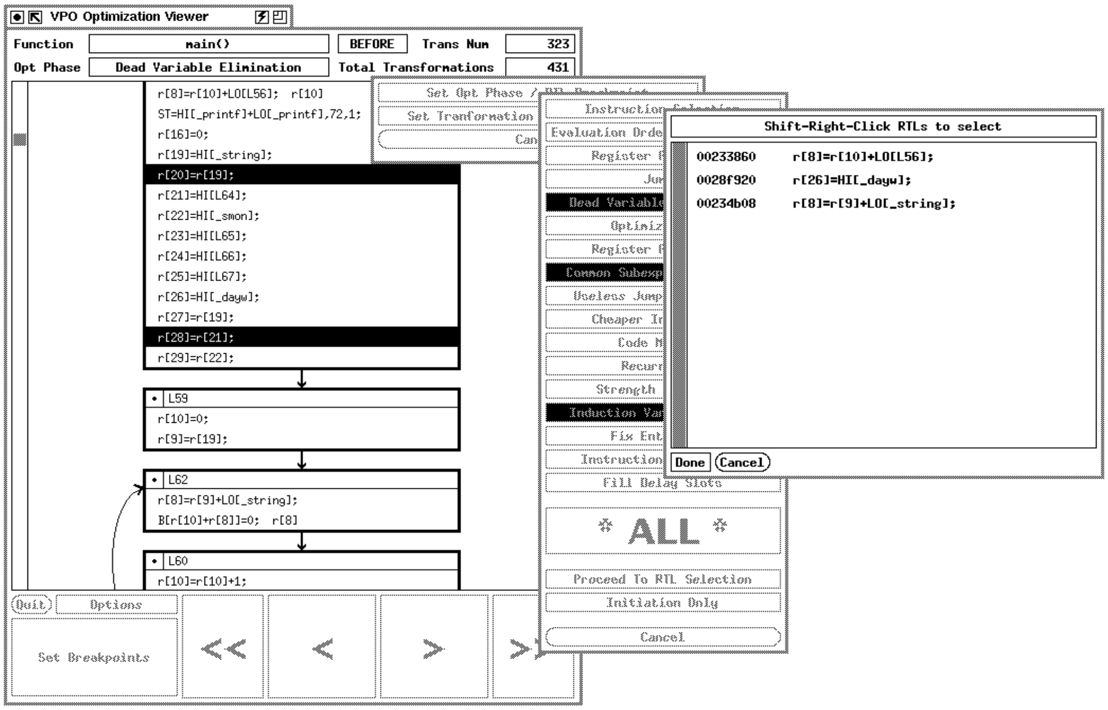

Figure 3.3: Breakpoint Menus

3.6 Setting Breakpoints

There are two main types of breakpoints in *xvpodb*. The first, and simplest, are *Transformation Number* breakpoints. The user inputs a transformation number or numbers, and *xvpodb* will break at the beginning or end of those transformations (depending on the direction of viewing). Since the transformation numbers are determined by *vpo*, this provides a convenient way to coordinate breakpoints in both the compiler and the viewer.

The second type of breakpoint is more general. Figure 3.3 shows several of the menus used to set this type of breakpoint. Basically, the user selects some number of optimization phases from a toggle menu. After completing this, a decision is

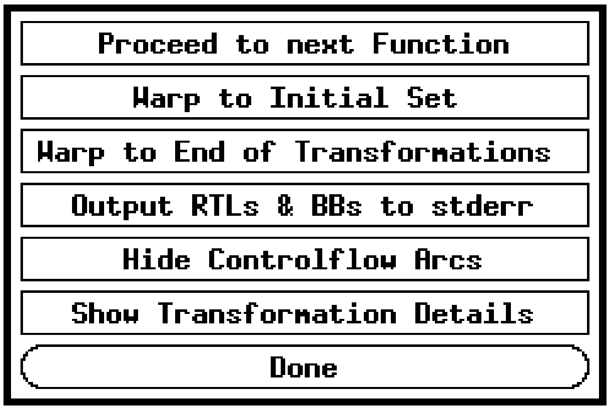

Figure 3.4: Options Menu

made to break on one of two criteria. The user can choose to break whenever one of the selected phases is encountered (at the beginning or end of the phases, depending on the direction of viewing), or the user can select a set of specific RTLs that should be recognized. In this latter case, the viewer will break whenever any of the RTLs selected is changed in any way during any of the selected optimization phases. The user can break on any change to a set of RTLs simply by selecting all optimization phases when setting the breakpoint (there is a button provided to do this with one click). Thus, breakpoints can be set on specific optimization phases, specific RTLs, or any combination of both.

3.7 Options Menu

The **Options** button pops up a menu of buttons that implement less commonly used features of the viewer, or features that that need not be available at all times (Figure 3.4). This was done to reduce screen clutter, and provides a convenient place for future developers to add minor features to the viewer. The **Proceed to**

Next Function button instructs *xvpodb* to discard the current function data and interpret the next function that was compiled. The **Warp To Initial Set** button allows the user to undo all transformations and be returned to a view of the initial set of RTLs for the current function. The **Warp to End of Transformations** button will display the completely optimized set of generated instructions, applying all transformations that have been received for the current function. Both of these functions skip all breakpoints. The **Output RTLs and BBs to stderr** button outputs a formatted textual version of the current set of RTLs to standard error, where it can easily be grabbed and inserted into a file for later study. The **Hide Controlflow Arcs** button toggles the drawing of the control flow arcs in the main display. This can be useful when using an extremely slow X server. Finally, the **Show Transformation Details** button toggles the verbosity of the step forward and step backward functions. When this toggle is selected, detailed information about each change in the current transformation being displayed is presented to the user while stepping forward or backward. This option is turned off by default, as it typically provides too much data to look at while searching for a particular transformation. However, after that transformation is found, this extra data is more likely to be of interest, and is easily accessible by activating this button.

3.8 RTL/Basic Block Information Popup

At any time, if the user clicks the middle mouse button on any RTL in the main RTL display, or on a basic block header, a small window will pop up showing extended information about the RTL or basic block. For an RTL, this would include its dead register list and exact pointer address in *vpo*. The latter piece of information allows the user to easily find and examine the RTL in the debugger

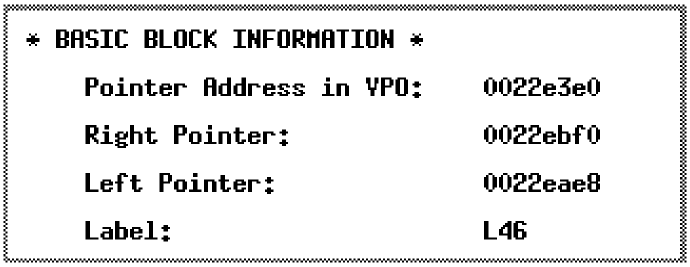

Figure 3.5: Information Popup

within which *vpo* is running. Similarly, if the middle button is clicked on a basic block header, the addresses the block itself, its left and right pointers, and the block label are displayed (Figure 3.5).

Enhancements to this popup that produce more information about basic blocks have been developed, and are discussed in section 6.1.

3.9 Debugging Example

The following example illustrates the power of both reverse viewing and breakpoint selection in *xvpodb* by showing how a programmer could view the entire genesis of a single generated instruction. After executing *vpo* and *xvpodb*, and after sending all messages associated with the function in which the RTL resides, the user clicks on the **Warp to End of Transformations** button (located in the options menu). This will display the completely optimized set of RTLs. Then, the user goes to the breakpoint menus (by clicking **Set Breakpoints**) and sets a breakpoint in the following manner. First, **Optimization Phase/RTL Breakpoint** is clicked, followed by **All Optimization Phases**, and **Proceed to RTL Selection**. Now, the user chooses the RTL of interest, and clicks on **Done**. At this point, the user can see each successive transformation to that RTL being "undone" by clicking the << button. If a transformation is encountered but not understood, the user can examine it (and other closely preceding and following transformations to gain context) as many times as necessary by clicking the < and > buttons. When the transformation is fully understood, the << button is again clicked. Eventually, the user will be informed that there is no other transformation involving that RTL (therefore, at that point the RTL is in its initial state, before any transformations). If desired, the user could click the >> button to watch the transformations be successively reapplied to *just that RTL*. Eventually, the user would again be informed that there are no more transformations involving that RTL (thus implying that it is currently in the fully optimized form in which *vpo* will output to the object file). Using this technique, a programmer can determine exactly how any specific RTL in the final output of the optimizer was generated.

Chapter 4

Implementing Xvpodb

4.1 Message Passing

To simplify the implementation, each message from *vpo* to *xvpodb* reflects at most a single change to the data structure containing the RTLs. All communications are one way, from *vpo* to *xvpodb*. Table 4.1 lists some of the types of messages that are sent.

A transformation is defined by a set of change messages bracketed by begin and end transformation messages. A set of transformations occurring in a particular optimization phase would be bracketed by begin and end phase messages. Thus, in general the message stream from *vpo* to *xvpodb* can be thought of as a set of transformation blocks (a begin transformation, some number of change messages, and an end transformation message), sometimes with either begin or end phase messages in between to show transitions to different optimization phases. A set of

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

Table 4.1: Message Types in Xvpodb

these messages representing an entire function compilation will be bracketed with begin and end function messages. There are special messages that communicate the initial set of RTLs, basic blocks, and dead register lists for a function (they are sent after the begin function message, and before any change information).

4.2 Main Data Structures

There are two main data structures in *xvpodb*, the *Optimization List* and the *Screen List*. The optimization list is a doubly linked list of nodes, each node representing a decoded message. This list is descended when performing forward transformations, and ascended when reversing them. A single pointer to this list represents the point during the compilation that is currently being displayed by the viewer. The act of stepping or continuing forward or backward merely moves this pointer in the appropriate direction, interpreting nodes along the way. All transformations above the pointer and none below will have been applied to the initial set of RTLs. When a transformation is applied, all information needed to reverse it is stored in the node. When ascending the list (undoing transformations), this information is used to restore the screen representation of the RTL or basic block to its previous state.

The screen list is a singly linked list of nodes, each node containing (among other things) one small section of the main RTL viewing area. These nodes represent the current state of all RTLs and basic blocks. These nodes are modified as transformations are applied or reversed. After the changes are complete, a simple routine copies all of the small screen sections to the actual viewer window. These nodes are created or deleted during RTL or basic block insertions or deletions (none of the optimization nodes are deleted until the user proceeds to the next function).

4.3 Controlflow Arcs and Highlighting

The arcs showing transfers of control are drawn after the middle section of the RTL display is created (from the screen list, as described above). If the length of the arc will be longer than two video pages a small stub arrow is drawn instead, as very long arcs are difficult to follow, and it is easier to use *magic jumping* (Section 3.2) to locate left and right pointer destinations. Finally, a routine highlights the RTLs pertinent to the current transformation and centers the screen on them.

4.4 Breakpoints

Breakpoints are stored in several small linked lists. The transformations are scanned for criteria that the user has indicated should be stopped upon before any changes are applied. If any of the breakpoint criteria is met, the viewer stops and displays the state of the RTLs at that point.

4.5 Sanity Check

During the development of *xvpodb*, the need was recognized to ensure that the viewer be bug free, in terms of accurately depicting the state of RTLs at any time. There can be few things in life more irritating than a buggy debugger. To aid in this effort, a sanity check module was incorporated into *xvpodb*. By issuing a function call within *vpo* (typically through the use of a symbolic debugger), the tester can invoke a routine that sends to *xvpodb* a set of messages that describe the current state of the RTLs in *vpo*. The viewer will compare this to its internal representation of the RTLs, and will report any differences found. This can be done at any time during a debugging session, and as many times as desired. Of course, the tester must ensure that all transformations that have been sent to *xvpodb* have been applied (i.e. *xvpdob* must be displaying the AFTER state of the last transformation that was sent), or differences will surely be found. The sanity check module was relatively simple to implement, as it reuses many of the routines needed to send the initial set messages.

The sanity check module was primarily intended to aid implementation and to provide a method for testing the accuracy of the viewer. However, it is conceivable that users might also like the ability to perform an occasional sanity check simply to ensure that the problem being examined really represents a bug in the optimizer, and not a debugger error.

4.6 Implementation Summary

Primary goals in the implementation of the viewer included ease of portability, simplicity of extension or upgrade, and robustness. The KISS principle was applied whenever possible, even when it implied some efficiency loss. The resulting application is satisfyingly fast on a Sun IPC running over a stock MIT X11R5 server. Interactive response time is quite adequate, enabling the user to click buttons and see transformations as rapidly as desired. Graphic drawing time is minimal due to the use of the the small screen sections in the screen list nodes. On the *vpo* end, virtually all modifications are localized to the machine independent portion of the source. The modifications required were relatively minor, and should be easy to incorporate when adding or modifying optimization phases in the future. Message passing is currently implemented using UNIX sockets, but could easily be redone using RPCs. Internet stream sockets were used to guarantee reliable communications and to allow long haul operation of the viewer.

Chapter 5

Comparison with Related Work

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 are 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.

Chapter 6

Existing and Future Enhancements to Xvpodb

6.1 Existing Enhancements

The viewer has been enhanced by members of an advanced compiler course as a semester project. The enhancement allows the user to see live registers information (i.e. which registers are live coming into and leaving a basic block) and see dominator information for the basic blocks. In addition, a special Fat Boy edition of *xvpodb* (in honor of the Harley motorcycle of similar name) has been provided for students working with a version of *vpo* targeted to an VLIW (Very Large Instruction Word) architecture. The Fat Boy can display RTLs that are over one hundred characters in length.

6.2 Future Enhancements

Many other enhancements are planned for *xvpodb*. Its modular organization and convenient **Options** menu aids the task of adding minor features. One interesting enhancement currently planned to be implemented is to allow the user to select a *motion picture* mode. This would display multiple transformations occurring to RTLs in real time (i.e. animation of the optimization process). It is thought that this may give students some insight into how various optimization phases perform their tasks. There are also plans to further enhance the availability of live variable range information. For example, the viewer could display all the live ranges of a particular variable when clicked by the user. The ability to click on an RTL and be presented with the actual assembly represented by that RTL has also been discussed. Finally, there is interest in producing a version of *xvpodb* for the GNU gcc compiler.

Chapter 7
Conclusions

An optimization viewer, such as *xvpodb*, can be very useful when retargeting a back end of a compiler. 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.

Another use of *xvpodb* is as a teaching aid for advanced compiler courses. 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. This tool, which allows students to interactively visualize the effect of each transformation, would be quite useful in illustrating these optimizations.

Bibliography

[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).

[BoW93] M. R. Boyd and D. B. Whalley, "Isolation and Analysis of Optimization Errors," *Proceedings of the SIGPLAN '93 Symposium on Programming Language Design and Implementation*, pp. 26-35 (June 1993).

[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).

[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).

Vita

Mickey Boyd was born on March 15, 1968, the son of Donald and Toki Boyd. He received his Bachelor of Science in Computer Science from Florida State University during the summer of 1990. He also received his Master of Science in Computer Science from Florida State University in the summer of 1993. While working on his Master's degree he was employed by the Department of Computer Science, first as a system manager, then as the system administrator. He is currently the system administrator of the Department of Mathematics at Florida State University.

He co-authored a paper, "Isolation and Analysis of Optimization Errors," which was based on the same research contained within this document. It was accepted by SIGPLAN PLD&I '93 and was published in *Proceedings of the SIGPLAN '93 Symposium on Programming Language Design and Implementation.* He presented the paper in Albuquerque during the conference.