THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

COMPILER MODIFICATIONS TO SUPPORT INTERACTIVE COMPILATION

By

BAOSHENG CAI

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

Degree Awarded:
Spring Semester, 2001

The members of the Committee approve the thesis of Baosheng Cai defended on March 30 2001.

David B. Whalley
Professor Directing Thesis

Xin Yuan
Committee Member

Robert van Engelen
Committee Member

Approved:

Theodore P. Baker, Chair
Department of Computer Science

ACKNOWLEDGEMENTS

I am deeply indebted in gratitude to my major professor, Dr. David Whalley, for his guidance, support, patience, and promptness during my research. He was always available to discuss new problems and exchange ideas. I cannot complete this thesis without his excellent teaching and mentoring. I thank my committee members Dr. Yuan and Dr. van Engelen for reviewing this thesis and subsequent valuable suggestions. I also thank Wankang Zhao for his assistance. The interface described in this thesis is implemented by Wankang Zhao.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

This thesis describes the modifications to a compiler to support an interactive compilation paradigm. Unlike traditional compiler systems where the smallest unit of compilation is typically a function and the programmer has no control over the optimization process other than what optimizations to apply, an interactive compilation system opens the optimization process and gives the application programmer, when necessary, the ability to finely control it. In particular, we allow the application developer to (1) direct the order and scope in which the optimization phases are applied, (2) specify code-improving transformations by hand, (3) undo previously applied transformations, and (4) support user-specified queries.

CHAPTER 1

INTRODUCTION

The problem of automatically generating acceptable code for embedded microprocessors is much more complicated than for general-purpose processors. First, embedded applications are optimized for a number of conflicting constraints, such as speed, code size, and power consumption. In fact, in many applications, the conflicting constraints of speed, code density, and power consumption are managed by the software designer writing and tuning assembly code. Unfortunately, the resulting software is less portable, less robust, and more costly to develop and maintain. Second, embedded microprocessors often have specialized architectural features that make optimization and code generation difficult [13]. While some progress has been made in developing high-level language compilers and embedded software development tools, most embedded applications are still coded in assembly language because current compiler technology cannot produce code that meets the cost and performance goals for the application.

This thesis presents a new compilation paradigm that we believe can achieve the cost/performance tradeoffs (i.e., size vs. power vs. speed vs. cost) demanded for embedded applications. The traditional compilation framework has a fixed order in which the optimization phases are executed and there is no control over individual transformations, except for compilation flags to turn code-improving phases on or off. In contrast, our compilation framework, called *vista* (Vpo Interactive System for Tuning Applications) gives the application user the ability to finely control the code-improvement process.

We had the following goals when developing the *vista* compilation framework. First, the user should be able to direct the order of the compilation phases that are to be performed. The order of the compilation phases in a typical compiler is fixed, which is unlikely to be the best order for all types of applications. Second, hand-specified transformations should be possible. For instance, the user may provide a sequence of instructions that *vista* inserts and integrates into the program. We are not aware of any compiler that allows a user such direct and fine control over the optimization process. Third, the user should be able to undo code-improving transformations previously applied since a user may wish to experiment with other alternative phase orderings or types of transformations. Finally, the low-level program representation should appear in an easily readable display. The use of dynamically allocated structures by optimizing compilers and the inadequate debugging facilities of conventional source-level symbolic debuggers makes it difficult for a typical user to visualize the low-level program representation of an application during the compilation process. To assist the user when interacting with the optimization engine, *vista* should provide the ability for the user to view the current program representation and any relevant compilation state information (i.e., live registers, available registers, def-use information, etc.).

Beyond *vista*'s primary purpose of supporting development of embedded systems, it has several other uses. First, *vista* can assist a compiler writer to develop new low-level code-improving transformations. The ability to specify transformations by hand can help a compiler writer to prototype new low-level code-improving transformations. The ability of viewing low-level representations can help a compiler writer diagnose problems when developing new transformations. In *vista*, when a transformation is applied, the exact portion of the representation that is altered will be indicated in the viewer. This feature can assist the compiler writer in identifying the problem. Second, it can help compiler writers in understanding the interaction and interplay of different optimizations. Finally, an instructor or educator

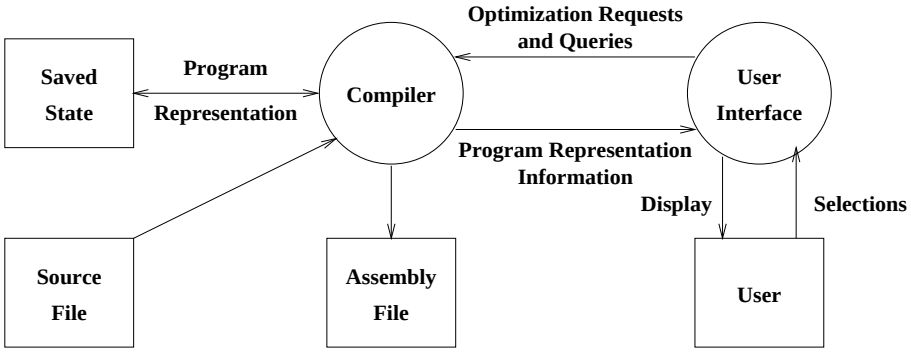

Figure 1.1. Overview of the Interactive Compilation Process

teaching compilation techniques can use the system to illustrate code-improving transformations to students.

Fig. 1.1 depicts a high-level overview of the flow of information in the interactive compilation system. The user would initially specify a source file to be compiled. The user retains control over the code-improvement process by specifying requests to the compiler, which includes the order of the optimization phases and actual transformations that are performed. The compiler responds to optimization requests by performing the specified actions and sends information about the changes to the program representation back to the user interface. Likewise, the compiler sends information regarding queries of the program representation state to the user interface. At some point a user session may be terminated and the current state of the program representation is saved to a file to enable future updates. The user may also wish to save multiple optimized versions to contrast their performance. Eventually, the user may be satisfied with the generated code and can produce the assembly. In addition, the user may wish to collect some preliminary results about the performance of the generated code, which can be accomplished by producing assembly that is instrumented with additional instructions that collect a variety of measurements during the program's execution.

The remainder of this thesis is structured as follows. Chapter 2 reviews related work regarding alternative compilation paradigms and user interfaces for compilers.

Chapter 3 presents the functional capabilities of *vista*. Chapters 4 to 9 discuss implementation issues that were involved in achieving these functionalities. Chapter 10 presents some techniques used in diagnosing problems during the implementation. Chapter 11 discusses future work in this topic and Chapter 12 gives the conclusions for the thesis.

CHAPTER 2

RELATED WORK

Some systems have been developed that are used for simple visualization of the compilation process. The UW Illustrated Compiler [1], also known as *icomp*, has been used in undergraduate compiler classes to illustrate the compilation process. The *xvpodb* system [6, 7] has been used to illustrate low-level code-improving transformations in the vpo compiler system [4]. *Xvpodb* has also been used when teaching compiler classes and to help ease the process of retargeting the compiler to a new machine or diagnosing problems when developing new code-improving transformations. Unlike these visualization systems, *vista* allows a user to interactively control the optimization process.

There have also been several systems that provide some visualization support for the parallelization of programs. These systems include the *pat* toolkit [2], the *parafrase-2* environment [16], the *e/sp* system [8], and a visualization system developed at the University of Pittsburgh [11]. All of these systems provide support for a programmer by illustrating the possible dependencies that may prevent parallelzing transformations from occurring. A user can inspect these dependencies and assist the compilation system by verifying whether a dependency is valid or can be removed.

In contrast, *vista* supports interactive compilation on a low-level representation (machine instruction) as opposed to a high-level representation (source code). Because of the difficulty of producing code for embedded processors that meets the conflicting constraints of space, speed, and power consumption, there is a wide body

of research that has advanced the state of the art [13, 14, 12, 10, 17]. There has also been some work on experimenting with optimization phase ordering and other techniques to produce better code. Coagulating code generators have been developed that use run-time profiles to perform optimizations phases on the most frequent sections of the code before the less frequently executed sections [15]. Genetic algorithms have been used to experiment with different orders of applying optimization phases in attempt to reduce code size [9]. *vista* allows a user to interactively experiment with the order of optimizations and the region in which the optimizations are performed.

CHAPTER 3

VISTA'S OPTIMIZATION ENGINE

Vista's optimization engine is based on *vpo*, the Very Portable Optimizer [3, 5]. *Vpo* has several properties that make it an ideal starting point for realizing the *vista* compilation framework.

First, *vpo* performs all code improvements on a single low-level representation called RTLs (register transfer lists). RTL is a low-level, machine and language independent representation that encodes machine-specific instructions. The comprehensive use of RTLs in *vpo* has several important consequences. One advantage of using RTLs as the sole intermediate representation is that many phase ordering problems are eliminated. In contrast, a more conventional compiler system will perform optimizations on various different representations. For instance, machine-independent transformations are often performed on intermediate code and machine-dependent transformations are often performed on assembly code. Local transformations (within a basic block) are often performed on DAG representations and global transformations (across basic blocks) are often performed on three-address codes. Thus, the order in which optimizations are performed is fixed. By only using RTLs, most optimizations can be invoked in any order and allowed to iterate until no further improvements can be found, which is one of the goals that we want to achieve in *vista*. In addition, the use of RTLs allows *vpo* to be largely machine-independent, yet efficiently handle machine-specific aspects such as register allocation, instruction scheduling, memory latencies, multiple condition code registers, etc. *Vpo*, in effect, improves object code. Machine-specific optimization is important because it is a

viable approach for realizing high-level language compilers that produce code that effectively balances target-specific constraints such as code density, power consumption, and execution speed. Another advantage of using RTLs is that the effect of an optimization can be easily understood since each RTL represents an instruction on the machine.

A second important property of *vpo* is that it is easily retargeted to a new machine. Retargetability is key for embedded microprocessors where chip manufacturers provide many different variants of the same base architecture and some chips are custom designed for a particular application. To retarget *vpo* to a new machine, one must write a description of the architecture's instruction set, which consists of a grammar and semantic actions. It is easier to write a machine description for an instruction set than it is to write a grammar for a programming language. The task is further simplified by the similarity of RTLs across machines, which permits a grammar for one machine to be used as the model for a description of another machine. Since the general RTL form is machine independent, the algorithms that manipulate RTLs are also machine independent, which makes most optimization code machine independent. So the bulk of *vpo* is machine- and language- independent. Overall, no more than 15 percent of the code of *vpo* may require modification to handle a new machine or language.

A third property of *vpo* is that it is easily extended to handle new architectural features as they appear. Extensibility is also important for embedded chips where cost, performance, and power consumption considerations often mandate development of specialized features centered around a core architecture.

A fourth and final property of *vpo* is that *vpo*'s analysis phases (e.g. data flow analysis, control flow analysis, etc.) were designed so that information is easily extracted and updated. This property makes writing new optimizations easier and it allows the information collected by the analyzers to be obtained for display.

CHAPTER 4

FUNCTIONALITY OF THE INTERACTIVE COMPILATION SYSTEM

In this section we describe the functionality of *vista* from a user's viewpoint. This functionality includes viewing the low-level representation, controlling when and where optimization phases are applied, specifying code-improving transformations by hand, and undoing previously applied transformations.

4.1 Viewing the Low-Level Representation

Fig. 4.1 depicts the user interface that was developed to support low-level interactive compilation. The programmer is presented with a view of the program representation on the right portion of the user interface. This representation is shown in basic blocks, which is a common method for displaying low-level control flow. Within the basic blocks are machine instructions. The programmer has the option to display these instructions as RTLs or assembly code. Displaying the representation in RTLs may be preferred by compiler writers, while assembly may be preferred by embedded systems application developers who are familiar with the assembly language for a particular machine. In addition, options are provided to display additional information about the program representation that a compiler writer may find useful.

The left side of Fig. 4.1 can vary depending upon the specific action requested. This figure shows the default type of display. Other types will be shown in subsequent sections. The function being compiled and the current transformation are indicated

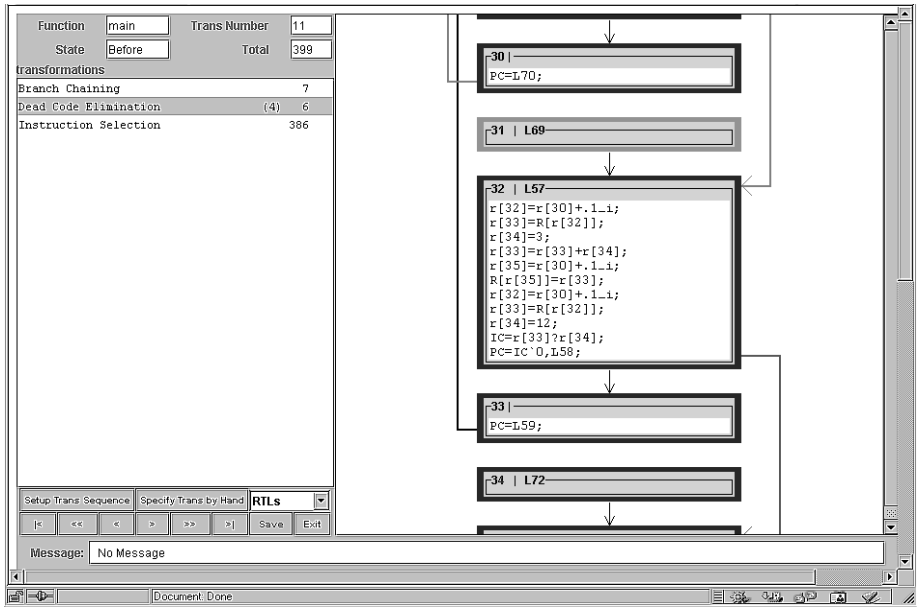

Figure 4.1. User Interface Depicting a History of the Compilation Phases Performed

at the top of the left portion of the user interface. Each transformation consists of a set of changes, where the program representation before and after the transformation are semantically equivalent. A transformation is displayed in a before or after state. In the before state, the transformation has not yet been applied. However, the instructions that are about to be modified or deleted are highlighted. In the after state, the transformation has been applied. At this point, the instructions that have been modified or inserted are highlighted. This highlighting allows a user to quickly grasp the effects of each individual transformation. The use of before and after states has been used in previous graphical compilation viewers [6, 7].

The bottom portion of the left side of Fig. 4.1 contains a variety of buttons representing selections that the user can make. The >, >>, >| buttons allow a user to advance through the transformations that were performed. The actions associated with the |<, <<, < buttons are described in Section 4. The > button allows a user to display the next transformation. A user can display an entire transformation (before and after states) with two clicks of this button. The >> button allows a user to

advance to the next phase. A phase is a sequence of transformations applying the same type of transformation. The >| button allows the user to view the program representation after advancing through all of the transformations that have been performed.

The middle portion of the left side of Fig. 4.1 shows the history of optimization phases that have been performed, which includes phases that have been applied in the user interface and the phases yet to be applied. For instance, the state represented in the figure is in the before state of the fourth transformation of the second phase. We have found that displaying a history of optimization phases in this manner helps to give a user some context to the current state being represented. Such visualization of the low-level representation provides the information and insight that can simplify the debugging of problems with the optimizer.

4.2 Directing the Order and Scope of the Optimization Phases

A programmer typically has little control over the order in which optimization phases are applied by a compiler. Usually the only control a programmer has is the ability to turn a compiler optimization on or off for the entire compilation of a file or function. For some compilation units, one phase ordering may produce the most suitable code, while a different phase ordering may be best for other compilation units. *Vista* provides the ability to specify what optimizations to apply to a program region and the order to apply them. A knowledgeable embedded systems application developer can use this capability for critical program regions to specify that the most appropriate transformations are applied in the most advantageous order.

The left portion of the window shown in Fig. 4.2 depicts the user selecting optimization phases. The user can make selections from a number of different required and code-improving transformation phases that are applied in the back end of a compiler. As each phase is selected, it is added to a numbered list of

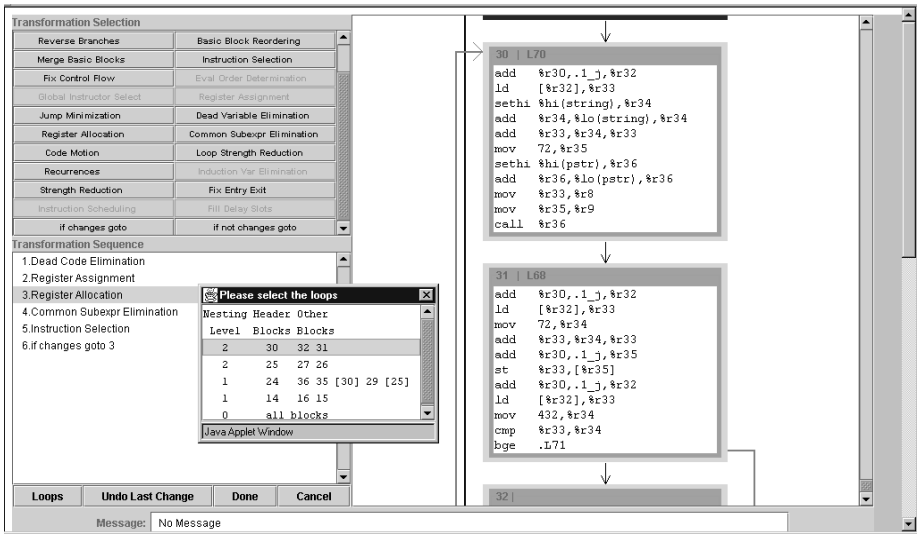

Figure 4.2. User Interface for Specifying a Sequence of Optimization Phases

optimization phases. In addition, the user is allowed to specify control over which selected phase is specified to be performed next. For instance, the figure shows that as long as changes to the representation are detected, the compiler is directed to repeatedly perform register allocation, common sub-expression elimination and instruction selection. Thus, we are in essence providing the programmer with the ability to program the optimizer in an optimization phase language. Once the user confirms the selection of the sequence of phases to be performed, this sequence is sent to the compiler, which performs the phases in the specified order and sends a series of messages back to the user interface describing the resulting program representation changes.

As shown in the left side of the figure, the user cannot select some of the phases. The reason for this is due to restrictions in the compiler concerning the order in which the phases can be performed. For instance, the compiler does not allow the register allocation phase (allocating variables to registers) to be selected until the register assignment phase (assigning pseudo registers to hardware registers) has been completed. Rather than allowing a user to make selections and later informing the

user that the selections were invalid, we decided to prevent the user from making an invalid selection.

In addition to specifying the order of the optimization phases, a user can also restrict the scope of the region of the program representation in which an optimization phase is applied. The user can select a number of basic blocks that the optimization phases are to be applied by just clicking within the basic blocks. For loop optimization phases, such as *minimize loop jumps*, *loop invariant code motion*, *loop strength reduction*, *recurrence elimination*, and *induction variable elimination*, it only makes sense to select a complete loop inside the scope. So we provide the user the ability to query for loops information. Fig. 4.2 shows a loops information window which displays all the loops in the function being optimized. The brackets around a basic block indicates that it is the header of an inner loop. A user can select a complete loop just by clicking the corresponding loop information in this window. This has the effect of selecting all of the basic blocks within the loop. Some phases cannot have their scope restricted due to the method in which they were implemented in the compiler or how they interact with other phases (e.g. fill delay slots). Note that by default the scope in which a phase is applied is unrestricted (i.e. the entire function).

4.3 Specifying Code-Improving Transformations by Hand

Many embedded architectures have special features (e.g., zero overhead loop buffers, modulo address arithmetic, etc) not commonly available on general-purpose processors. Automatically exploiting these features is difficult due to the high rate at which these architectures are introduced and the time required for a highly optimizing compiler to be produced. Yet generating an application entirely in assembly code by hand is not an attractive alternative due to the labor involved. It would be desirable to have a system that supports traditional compiler optimization phases and the

ability to hand-specify transformations that are not automatically performed by a compiler.

Fig. 4.3 shows the interface used to support specifying code-improving transformations by hand. The user selects an instruction with a GUI pointing device (e.g. mouse) and a list of possible types of hand-specified changes that can be performed associated with that instruction is displayed. As each change is selected, the change is sent to the compiler, which checks it for validity. For instance, if an instruction is inserted, then the syntax is checked to make sure it is valid. A number of semantic checks are also necessary. For instance, if the target of a branch is modified, then the compiler checks to ensure that the target label in the branch is actually a label of a basic block. The compiler responds to each change by indicating if the change was valid and sending the appropriate change messages to the user interface so the presentation of the program representation can be updated. The approach of immediately querying the compiler for the validity of each change at the point the change was specified allows the compiler to ensure that the user never causes the program representation to be placed in an inconsistent state. We also plan to eventually integrate *vista* with the ability to validate entire transformations, where the semantic effects of a region of code before and after a hand-specified transformation are determined to be equivalent [19].

Fig. 4.4 shows an example of the improvement that can be obtained by specifying a transformation by hand. Fig. 4.4(a) depicts a loop in the whetstone benchmark and Fig. 4.4(b) shows the SPARC assembly code generated for the call to the *sqrt* function. The SPARC calling sequence specifies that the first six words of arguments be passed in integer registers. Moving a double-precision value from a pair of floating-point registers to integer registers on the SPARC requires accesses to memory since a floating-point register and an integer register cannot be referenced in the same instruction. A knowledgeable programmer can determine that the *sqrt* function can be accomplished with the *fsqrtd* instruction that is available on the SPARC.

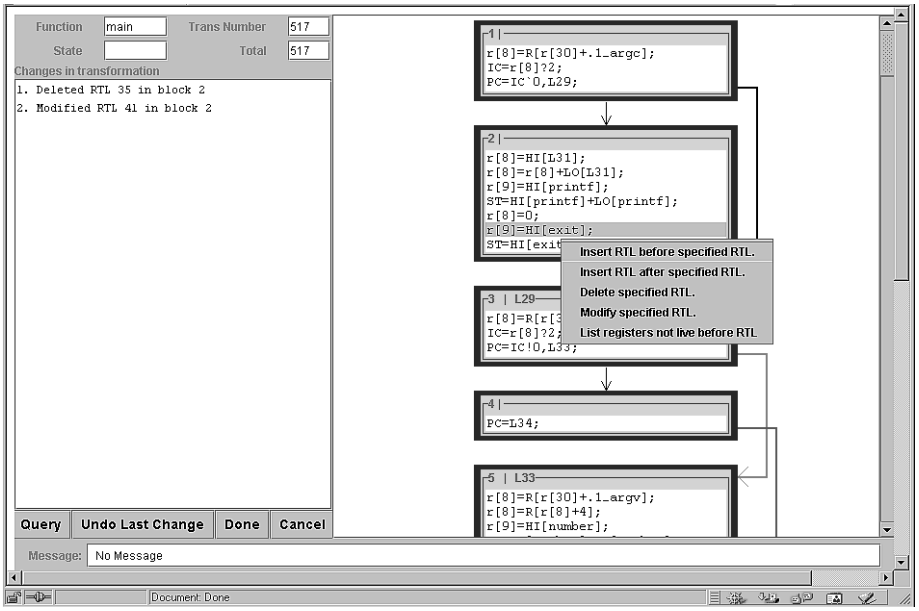

Figure 4.3. User Interface for Specifying a Transformation by Hand

Fig. 4.4(c) shows that the call and four preceding instructions can be replaced with a single instruction by performing this hand-specified transformation.

The user also can query the compiler for information that can be helpful when specifying a transformation by hand. For instance, a user may wish to know which registers are live at a particular point in the control flow. The query is sent to the compiler, the compiler obtains the requested information (calculating it on demand) and sends it back to the user interface to be displayed. Thus, the compiler can be used to help ensure that the changes associated with hand-specified transformations are properly made or to guide the user in generating valid and more efficient code.

The ability to specify low-level code-improving transformations by hand has another interesting application. Unlike high-level code-improving transformations, it is difficult to prototype the effectiveness of low-level code-improving transformations. One cannot simply modify the source code of an application to gauge the effectiveness of most low-level transformations. Often architectural features may need to be exploited and these features can only be accessed after the low-level representation

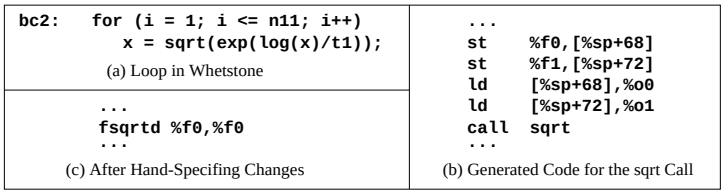

Figure 4.4. Example of a Hand-Specified Transformation

has been generated and other code-improving transformations have been applied. However, *vista* allows a compiler writer to specify the proposed transformation by hand at the appropriate point in the compilation process, perform additional optimization phases in the compiler, generate assembly code, and gather measurements. Thus, one can easily prototype low level code-improving transformations using *vista*.

4.4 Undoing Previously Applied Transformations

One issue with an interactive compilation system is how to allow an embedded system application developer to experiment with different orderings of phases and/or hand-specified transformations in an attempt to improve the generated code. In order to support such experimentation, we provide the ability for the user to reverse previously made decisions regarding phases and transformations that have been specified.

This undoing of transformations is accomplished using the (|<, <<, <) buttons, as previously shown in Fig. 4.1. The (|<, <<, <) buttons allow a user to undo the transformations that were previously applied. The < button allows a user to display the previous transformation. The << button allows a user to back up to the previous optimization phase. Likewise, the |< button allows the user to view the program representation before any transformations have been applied. The ability to back up and view previously applied transformations is very useful for understanding how code was generated or to grasp the effects of individual transformations.

If the user invokes an optimization phase or hand-specified transformation while viewing a prior state of the program representation, then the subsequent transformations will be removed and the state of the program representation in the compiler will be adjusted to reflect the currently viewed state. Thus, the user has the ability to permanently undo previously applied phases and transformations.

The ability to undo transformations can also be useful in a non-interactive compilation environment. A traditional compiler could use this feature to exhaustively attempt a variety of optimizations and select the phase ordering that produces the most effective code. In addition, it is sometimes easier to perform a portion of a transformation before completely determining whether the transformation is legal or worthwhile. Being able to undo changes to the program representation will facilitate the development of such transformations.

CHAPTER 5

SUPPORTING VIEWING OF TRANSFORMATIONS

Xvpodb, a X-Window based visualization tool, had previously been developed to support the analysis of optimizations performed by the *vpo* optimizer. It is a graphical optimization viewer that can display the state of the program representation before and after transformations. The *xvpodb* viewer is a separate program that can execute concurrently with the *vpo* optimizer.

In order to view the program representation, the *vpo* optimizer first passed a set of messages that describes the initial state of all RTLs in the function currently being compiled before performing optimizations. After receiving these messages, *xvpodb* displayed this initial set to the user. Subsequently, messages containing descriptions of all changes to the RTLs as they occur are passed to *xvpodb*, which stores them for later interpretation at the request of the user. All of these messages passed from *vpo* to *xvpodb* were accomplished via system calls using Unix sockets.

The implementation details of *xvpodb* were described in [7]. In *vista*, we used the similar techniques as in *xvpodb*, that is, using Unix sockets to pass program representation and changes from *vpo* to the viewer. The existing message-passing framework of *xvpodb* made this task easier. The following sections in this chapter describe some changes that we made to support our new viewer, as well as new features in viewing.

5.1 Supporting Viewing of Transformations in Assembly Mode

It is likely that most users would prefer viewing program representations in assembly language as opposed to viewing them in RTLs. Thus, we decided to add the functionality of viewing the program representation in assembly language. We have two modes of viewing: assembly and RTLs. The user would be given the option to select which mode he/she prefers. There are two things that we did to support this functionality.

First we provided the ability to translate an encoded RTL into an assembly instruction. In *vpo*, we have a machine description, which will parse the encoded RTL. The machine description is used to check if the encoded RTL is a legal instruction and to print the RTL to standard output as an assembly instruction. So we modified the machine description to output the assembly instruction to a global character string instead of standard output. This string can then either be used in passing messages or being printed to standard output.

Second, in order to view the instructions in the assembly language for the machine, we augmented the existing protocol to send the assembly instruction after the RTL in the messages that contain RTLs. The user interface keeps both types of representation for each instruction.

The right side of Fig. 4.1 shows the program representation in RTL mode, while the right side of Fig. 4.2 shows the program representation in assembly mode. So the user is offered the flexibility to view the program representation in his/her favorite format.

5.2 Inter-Process Communication between *Vpo* and the User Interface

The user interface was implemented using Java to enhance its portability. The version used was Java 1.2, which includes the Java Swing user interface toolkit to

create graphical user interfaces. Java is often interpreted rather than compiled, which can result in slower execution. We found that the aspects of the interface that limit its speed are the displaying of information and the communication with the compiler. The performance of the interface was satisfyingly fast, despite having not been implemented in a traditionally compiled language.

We separated the compiler and the user interface into different processes for several reasons. First, we were concerned that the amount of memory used by the compiler and the user interface may be excessive for a single process. Second, the use of separate processes provides additional flexibility. For instance, the sequence of change messages sent from the compiler to the user interface can be saved and a simple simulator has been used instead of the compiler to facilitate demonstrations of the interface. Likewise, a set of user commands can be read from a file by a simple simulator that replaces the user interface, which can be used to support batch mode experimentation with different phase orderings. Finally, separating the compiler and the user interface into separate processes allows users to access the interactive compilation system on a different machine from which the compiler executes. The communication between the compiler and the user interface was accomplished using UNIX sockets.

5.2.1 Framework of the Main Function

We developed a main function to support communication with the user interface. Fig. 5.1 contains pseudocode that depicts the logic for the main function.

```
main()
{
    Create a socket;
    Bind this socket with a certain port number;
    Listen to this socket;
    WHILE (1) DO
        Accept a listening socket;
        Fork a child process to handle new connection;
        IF (Child Process) THEN
            Perform User request;
            Exit;
        END IF
    END WHILE
}
```

Figure 5.1. Main Function Framework

A port number is assigned to our *vpo* program. The optimizer runs as a server, listening to the socket. If there is a request from the user interface requesting for connection of this server, our *vpo* will accept this connection, and *fork* a child process to serve the user interface. The parent process will continue to listen for other connections. With this typical server framework, *vpo* could even serve multiple user requests in parallel.

5.2.2 Protocols from *Vpo* to the Viewer

As a client/server system, an important aspects to ensure reliable communication between client and server is defining communication protocols. The protocol must be defined carefully and precisely. In the *vista* system, there are several issues that affect the definition of the protocols.

First, the protocols that we defined must be robust. That is, when an error occurs during the connection, the server cannot be crashed. Because of this reason, we considered many error conditions that might occur, and defined error message protocols between the client and the server. The error message is defined as below:

$$< error\_msg > ::= \text{ERROR\_MSG error\_type.}$$

The *error_type* is defined both in the user interface and the compiler, so that both of them can give a correct error message.

Second, the messages must be short to acquire reasonable speed. The aspects of the interface that limits its speed are the displaying of information and the communications with the compiler. To shorten the messages, each type of messages begins with a single character, which identifies the type of this message. The delimiter of each message component is just a single blank. For example, the "delete RTL" message is defined as:

$$< del\_rtl\_msg > ::= DELRTL < rtl\_id >< block\_id >,$$

in which DELRTL is defined as a single character '-'. When the user interface sees this character, it knows that this is a "delete RTL" message, so that it can read in rtl_id and block_id accordingly.

Finally, to simplify the passing of messages and diagnosing problems, all the communication between the compiler and the user interface were accomplished using character strings. Binary number must be converted into character strings before being transferred. There are several problems related with passing binary values.

1. Different implementations store binary numbers in different formats. For example, some implementations store binary numbers in big endian byte order, while some implementations store binary numbers in little endian byte order.

2. Different languages store the same data type differently. For example, the size of an integer in Java is always 32 bits, while that in C is different among different architectures.

3. Different implementations pack structures differently.

Although we can explicitly define the binary formats of the supported data types(number of bits, big or little endian) and pass all binary data across sockets in this format, it is quite complicated. By only allowing text strings passed through sockets, the difficulties of passing binary numbers are eliminated.

Appendix A defines the protocol of the messages that are transferred from *vpo* to the user interface. The protocol defines the initial program representation messages, transformations, and changes that occur in transformations. After the communication is established, the compiler first sends the initial set to the user interface. Then it reads commands from the user interface, and do the specified transformation phases. Finally it sends the phases to the user interface. Each phase consists of one or more transformations, while each transformation consists of one or more changes. To reduce message traffic, empty transformations (transformations which contain no changes) and empty phases (phases which contain no transformations) are not allowed.

In this protocol, each RTL has to be uniquely identified to support messages related with RTLs during transformations. A global variable *max_rtl_id* was used to record the maximum number of RTLs currently used. When an RTL is allocated, *max_rtl_id* is increased by 1, and then assigned to the RTL. We used a similar strategy to uniquely identify each basic block.

CHAPTER 6

DIRECTING THE ORDER AND SCOPE OF THE OPTIMIZATION PHASES

One advantage of using RTLs as the sole intermediate representation is that many phase ordering problems are eliminated. By only using RTLs, most optimizations can be invoked in any order and allowed to iterate until no further improvements can be found. In other words, using RTLs as the only intermediate representation facilitates the implementation of the ability to interactively direct the order of the optimization phases. The ability to limit the scope of the optimization phases also gives the user more control over the optimization process. To implement these capabilities, we defined the protocol for the commands being passed from the viewer to the compiler. Numerous modifications were also made to *vpo* to support these functions.

6.1 Protocol from the Viewer to *Vpo*

Besides the protocol from the compiler to the viewer, we also defined a protocol from the user interface to the compiler. This protocol define sequences of commands that a user can select, including phase order command sequences, hand-specified transformation sequences, undo command sequences, loops query sequences, and so on. Each sequence is terminated by QUIT_TRANS. A phase order command sequence is defined as below:

normal_sequence ::= numblocks ids commands,

where *numblocks* indicates the number of blocks to which the sequence of optimizations is applied, *ids* indicates the unique identifiers of these blocks, and *commands* indicates the sequence of optimization phases, which will be performed in order. A zero value of *numblocks* means to apply the transformation to all the basic blocks in the function. After receiving the commands, *vpo* will perform this sequence of optimization phases, send information back to the user interface, and wait for another sequence. No more sequences of commands will be issued from the user interface for the optimization of a function when a STOP_TRANS is encountered. The complete protocol from the user interface to *vpo* is defined in Appendix B.

6.2 Modifications to *Vpo*

There were numerous modifications to *vpo* that were required to support interactive specification of the order of optimization phases. First, a function called *getcmds()* is used to read in a sequence of commands from the viewer. Then the commands are parsed and stored in a data structure to be used during the interactive optimization. Fig. 6.1 shows the pseudo code of *getcmds* function. It implements the protocol from the viewer to the compiler.

Second, the high-level functions in *vpo* to perform the optimization phases for a function had to be rewritten. *Vpo* had a fixed order, depending upon the compilation flags selected, in which optimization phases were attempted. Fig. 6.2 shows the revised logic used for responding to user requests. It is used to support directing the order of the optimization phases, and limiting the scope of the optimization phases, which are described in this chapter. It is also used to support queries, hand-specified transformations and reversing previous applied transformation, which will be discussed in later chapters.

After composing a sequence of optimization phase commands, this sequence is sent by the user interface to *vpo* and the compiler interprets these commands until an *exit* command is encountered. The compiler first sets the *inscope* field in each basic

```
//Read the first part of the commands, e.g.  number of blocks
//followed by block IDs
Read in the first element E.
IF E is STOP_TRANS or QUIT_TRANS THEN
    RETURN.
ELSE IF E is UNDO_TRANS OR UNDO_CHANGE
    Read in the number of transformations (or changes) to reverse.
ELSE IF E is the number of blocks that the sequence will apply
    Read in each block ID.
ELSE
    ERROR.
END IF

//Read in the sequence of commands that follows block IDs
DO
    Read in an element E.
    IF E is IF_TRUE_GOTO_TRANS OR IF_FALSE_GOTO_TRANS
        Read in the destination transformation number.
    END IF
WHILE E is not STOP_TRANS AND E is not QUIT_TRANS
```

Figure 6.1. Algorithm for Reading User Commands

block to indicate that it is in the scope that the transformations are applied. Then the command sequence is applied to this scope. When applying the transformations, only the basic blocks in the scope can be changed. The *branch* command allows a user to specify a transfer of control based on whether or not changes to the program representation were encountered. Before each optimization phase, *vpo* performs the analysis needed for the phase that is not already marked as valid. After performing the optimization phase, *vpo* marks which analysis could possibly be invalidated by the current phase. Identifying the analysis needed for each optimization and the analysis invalidated by each optimization was accomplished in a table-driven fashion to facilitate maintenance of the compiler.

We had to also identify which optimization phases that were required during the compilation (e.g. *fixing the entry and exit to conform to calling sequence*

conventions), which optimization phases could only be performed once (e.g. *register assignment*), and the restrictions on the order in which optimization phases could be applied. Fortunately, many ordering restrictions required by other compilers are not required in *vpo* since all optimization phases are applied on a single program representation (RTLs). The ordering restrictions that were required were accomplished in a table-driven fashion.

```
Read request from the user.
WHILE (user selects additional requests to be performed) DO
    IF user selected a sequence of phases to be performed THEN
        FOR Each basic block in the scope specified
                Set its inscope field to TRUE.
        pc = 0.
        WHILE commands[pc].oper != EXIT DO
                IF commands[pc].oper == BRANCH THEN
                    Adjust pc according to branch command.
                    CONTINUE.
                END IF
                Perform analysis needed for current phase.
                SWITCH (commands[pc].oper)
                CASE BRANCH_CHAINING:
                    remove branch chains.
                    BREAK.
                ...
                END SWITCH
                Mark analysis invalidated by current phase.
                pc += 1.
        END WHILE
    ELSE IF user selected a hand-specified change THEN
        Perform hand-specified change selected by the user.
    ELSE IF user selected to reverse transformations THEN
        Reverse transformations as specified by the user.
    ELSE IF user requested a query THEN
        Calculate query information and send to user interface.
    Read request from the user.
END WHILE
```

Figure 6.2. Algorithm for Performing User Requests on a Function

CHAPTER 7

SUPPORTING LOOP INFORMATION QUERIES

To specify the scope that the transformations are applied, the user may need information about the loops in the function being optimized. For example, it does not make sense to apply a loop transformation to only part of a loop. This chapter will discuss the implementation to support loop information queries.

The command for a query is defined as a unique character. When the compiler receives this command, it will send loops information back to the user interface. The protocols for the loops informationpassed from the compiler to the user interface is defined below:

$$
\begin{aligned}
< loops\_msg > \quad &::= \quad \text{BEGINLOOPS} < loop\_msg > + \quad \text{ENDLOOPS} \\
< loop\_msg > \quad &::= \quad \text{LOOP} < nesting\_level >< header\_id > \\
&\qquad\qquad\qquad\qquad < inner\_id > * \\
&\qquad\qquad\quad \text{ENDLOOP} \\
< header\_id > \quad &::= \quad \text{block\_id} \\
< inner\_id > \quad &::= \quad \text{block\_id} \mid \text{-header\_id}
\end{aligned}
$$

The loops information consists of all the loops in the current function. The information for each loop includes its nesting level, header ID, and the IDs of other basic blocks in this loop. If there are other nested loops in this loop, only their header IDs are included in the loop information. The negated number of the header ID is used to distinguish the header of a nested loop and a normal basic block. For

example, for the basic block structure in Fig. 7.1, the loop message is

BEGINLOOPS
    LOOP 1 2 -3 7 ENDLOOP
    LOOP 2 3 -4 -5 6 ENDLOOP
    LOOP 3 4 ENDLOOP
    LOOP 3 5 ENDLOOP
ENDLOOPS.

The loop message will be passed to the user interface, which displays the message in the format shown in Fig. 4.2.

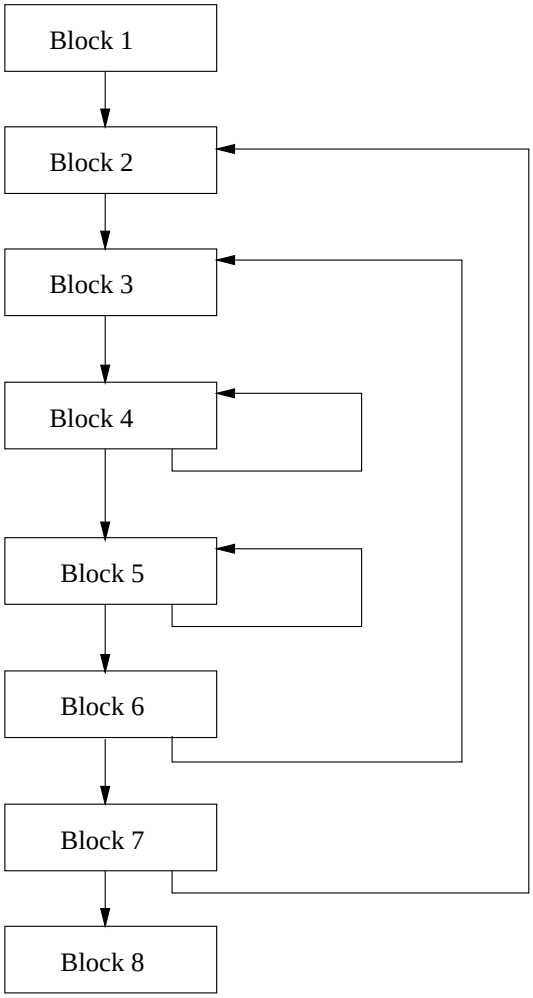

Figure 7.1. An Example with Nested Loops

CHAPTER 8

SUPPORTING HAND-SPECIFIED TRANSFORMATIONS

As previously mentioned, the feature of supporting hand-specified transformations was added due to the difficulty of automatically exploiting special features of different architectures in embedded systems by a compiler. A hand-specified change, which can be a basic block level change or an instruction level change, can be specified via the user interface. The change is then sent to the compiler, which will check its validity. If the change is valid, the compiler will update the program representation and send back actual changes occurred in the compiler to the user interface. Otherwise, the compiler will just send back an error message. The program representation used by the compiler for an instruction is an encoded RTL. The user enters either human readable RTL or assembly instructions. Translators were developed to convert the human readable instructions to encoded RTLs.

8.1 Related Messages

The message related hand-specified transformations is defined as below:

hand_trans ::= HAND_TRANS < *change* > QUIT_TRANS.

Each hand-specified transformation consists of one change, and ends with a QUIT_TRANS. Changes can be classified as block level changes and instruction level changes. Block level changes include inserting a basic block, deleting a basic block, and modifying the label of a basic block. Instruction level changes include

inserting/deleting/modifying/moving an instruction, while the instruction can be either in assembly format or in RTL format. The following sections describe the implementation related with basic block level changes and instruction level changes, respectively.

8.2 Hand-Specified Changes at the Basic Block Level

Since we do not need any translators, it was relatively easy to implement hand-specified changes at the basic block level. First, a hand-specified change received from the user interface has to be checked by the compiler to ensure that it is valid. For example, deleting a non-empty basic block is illegal. If the change is illegal, the compiler will just send an error message. Otherwise, the compiler will update the program representation corresponding to the required change, and then send back the actual changes to the user interface. One change in a command from the user interface can result in several changes. For example, inserting or deleting a basic block can cause changes in the control flow.

8.3 Hand-Specified Changes at the Instruction Level

After the compiler receives an instruction level change, it will first translate the specified instruction, whether in RTL or assembly, into an encoded RTL, which is the format used in the compiler. We developed two translators for this purpose. The first translator converts a human readable RTL into an encoded RTL. The second translator converts an assembly instruction into an encoded RTL. The translated encoded RTL will then be checked by the compiler to verify its validity. This check includes ensuring not only that the syntax of an instruction is valid, but also that its semantics are valid with regard to the rest of the program representation. If the change is valid, the compiler will update the program representation and send back

actual changes to the user interface. Otherwise, the compiler will just send back an error message.

8.3.1 Translating an Assembly Instructions to an Encoded RTL

We developed a translator to convert a single assembly instruction into an encoded RTL. It is used when the user modifies or inserts an assembly instruction using the user interface. The new assembly instruction will be sent to the compiler, which calls the translator to translate it into an encoded RTL before updating the program representation. Each assembly instruction corresponds to one RTL. The translator was implemented by checking the operator first, and then the operands based on the operator. After that, a simple syntax check is performed to check if the number and types of operands match the operator. Finally an encoded RTL will be generated according to the RTL standard.

Registers in RTL are encoded as two characters, based on the register type and register number. First, the translator reads in the register type and number in assembly. The translator then encodes this register based on its type and number. For a variable, the translator must search the global variable list and the local variable list to determine the variable type (global or local) and its identifier number. It then encodes this variable based on these two attributes of this variable. Constant are not encoded at all.

8.3.2 Translating a Human-Readable RTL into an Encoded RTL

We developed a translator to convert a human readable RTL to an encoded RTL. This translator is needed when a user modifies or inserts an RTL using the user interface. The new RTL will be transferred in a human-readable form. It must be encoded by the compiler before updating the program representation. This translator is quite similar to the translator from assembly into encoded RTL. However, since the format of a human-readable RTL is quite similar to that of its corresponding encoded

RTL, it was relatively simple to implement. The translator simply reads in each word, and encoded registers, local variables and global variables while scanning the RTL. Since it is difficult to distinguish between global variable, local variables and labels, we restricted the user to use a special form to identify a global variable or a local variable. A global variable must be represented as *global(variable name)*. Similarly, a local variable must be represented as *local(variable name)*. For example, *L31* in *r[8]=HI[L31+4]* represents a label. The user can indicate that it represents a global variable in *r[8]=HI[global(L31)+4]*, or a local variable in *r[8]=HI[local(L31)+4]*.

8.3.3 Syntax and Semantic Check

The above two translators do not completely check the syntax of a user specified instruction before encoding it. The compiler will check the syntax of the encoded RTL after the translation. It uses the machine description in the compiler to check if the generated encoded RTL was legal for the machine.

Semantic checks related with instruction level changes are required. The semantic check of a non-branch instruction is quite straightforward. For a register, the compiler will check whether this register is defined in the system. For a variable, the compiler will search the global or local variable list to find out whether it has been previously defined. This has to be done before translation. The semantic check of a branch instruction is more complicated since extra checks are required. The compiler must ensure that a branch instruction is at the end of a basic block. Furthermore, if the branch instruction contains a label, then the label must exist. Semantic checks were performed before the compiler committed to insert, modify, or move an instruction.

8.3.4 Update Program Representation

After the compiler determines that the semantics of the instruction is valid, the program representation is updated and actual changes are sent back to the user interface. For a non-branch instruction, one change specified in a command sequence

will only result in one actual change in the compiler. But for a branch instruction, one change specified in a command sequence may result in several actual changes in the compiler. For example, inserting a branch at the end of a basic block not only results in inserting an instruction, it can also modify the control flow of the program representation. These changes will be grouped together as a transformation and sent back to the user interface for consistency.

CHAPTER 9

REVERSING PREVIOUSLY APPLIED TRANSFORMATIONS OR CHANGES

Vista provides the ability to undo previously applied transformations or changes for two purposes. First, this ability can help the user to experiment with different ordering of phases and/or hand-specified transformations in an attempt to improve the generated code. Second, this feature gives the user the ability to change his/her mind. For normal transformations, the user can request to undo one or more previously applied transformations. For hand-specified transformations, the user can request to undo one or more previously applied changes. These undo commands are sent to the compiler, which reverses the effects of specified number of transformations or changes.

There are several difficulties related with reversing transformations or changes. First, the compiler must have enough and correct information required for reversing a transformation or a change. This requires that each time a change occurs, the compiler must record the required information correctly. Second, the control flow of the "reversed" program representation must be consistent. For example, when undoing the effect of inserting a basic block, not only should the compiler delete this basic block from its program representation, it must also update the control flow information of the basic blocks to which it falls through or jumps.

9.1 Related Messages

The messages related with reversing commands from the user interface to the compiler are defined as below:

undo_command ::= UNDO_TRANS < *number* >
undo_change ::= UNDO_CHANGE < *number* >.

The definition of a transformation is the same between the user interface and the compiler. However, the definition of a change is a little different between the user interface and the compiler. A change specified by a user can result in several changes in the compiler. When the user requests to undo a change, the user interface will identify the number of actual changes in the compiler corresponding to this change, and send an *undo_change* message to the compiler.

9.2 Data Structure

In order to undo previously applied transformations, a linked list structure was used to keep the history of changes that occurred in the compiler. Fig. 9.1 depicts this linked list. All changes (additions and deletions) to the linked list occur at the tail.

When a phase of transformations is applied, the compiler inserts a BEGINPHASE node at the beginning. There can be several transformations in one phase. Each time a transformation is applied, a BEGINTRANS node is inserted. Usually there are multiple changes in a transformation. Each time a change occurs, the change type, as well as the information needed to reverse this change, are stored in the linked list. The end of a transformation is indicated by an ENDTRANS node, while the end of the whole phase is indicated by an ENDPHASE node. When the compiler is required to undo a transformation, it will search the linked list from the tail and reverse the effect of each node until it finds a BEGINTRANS node. When the compiler is

required to undo several changes, it will just reverse the specified number of nodes in the linked list.

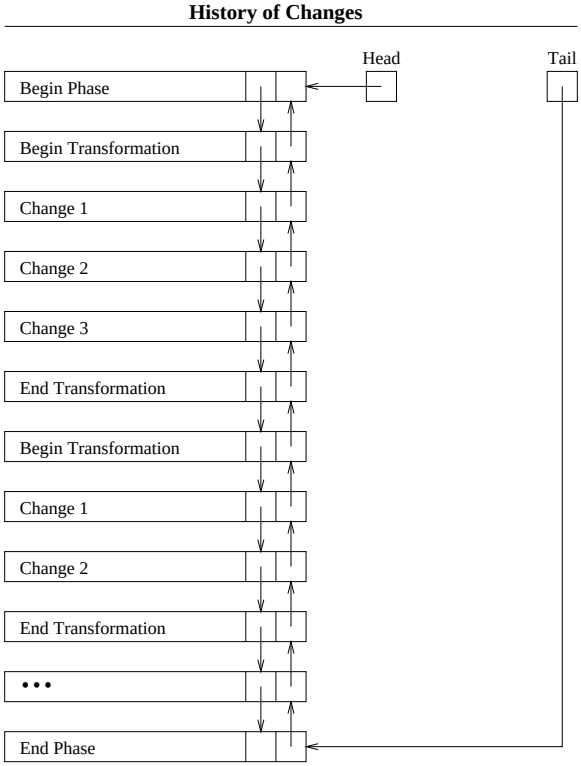

Figure 9.1. Data Structures Used for Undoing Transformations

Enough information regarding each change must be saved so its effect can be reversed if requested by the user. For instance, if a change reflects a modification to an instruction, then the compiler must save the previous representation of the instruction before the modification. If a change reflects inserting or deleting a basic block, the previous control flow information before the change must be saved. So a node in the undo linked list includes the change type, basic block ID, previous label for the basic block, previous RTL instruction, previous control flow information, and so on.

9.3 Modifications to *Vpo*

As described in the previous section, new data structures must be added to the *vpo* to store changes needed for reversing purpose. Besides the data structures, there are two major modifications in *vpo*.

First, in each place where a change of the program presentation occurs, code was added to record this change. A new node is created first. The information needed to undo this change is then stored in this node. Finally this node is added to the tail of the linked list used. Since there are a lot of changes of program representation in the *vpo*, modifications were required to many parts of the compiler.

Second, functions were added to implement the reversing ability. The compiler first determines the type of the change being reversed. It reverses the effect of this change based on its type and the information stored in the linked list. Finally, if the change results in other changes in the control flow, the control flow information has to be recalculated. Fig. 9.2 shows the algorithm to undo a change.

Because of the numerous modifications in the *vpo*, the regression test becomes very important. Each time a new change was added to the compiler, the regression test had to be done to ensure that the new change did not introduce errors to the compiler.

```
SWITCH(type of modification)
CASE UPDATEBB:
    Reorder the RTLs in this basic block to its original order;
    BREAK;
CASE LABELBB:
    Restore the label;
    BREAK;
CASE MOD_LEFT:
CASE MOD_RIGHT:
CASE MOD_DOWN:
    Restore the corresponding pointer;
    Restore the predecessors of the blocks that it jumps to or falls through;
    BREAK;
CASE NEWRTL:
    Delete the inserted RTL from RTL list;
    BREAK;
CASE DELRTL:
    Insert the deleted RTL to original location;
    BREAK;
CASE MOVERTL:
    Move RTL back to its original location;
    BREAK;
CASE MODRTL:
    Modify the RTL back to its original value;
    BREAK;
...
END SWITCH
```

Figure 9.2. Algorithm to Undo a Change

CHAPTER 10

DIAGNOSING PROBLEMS

One difficulty in this project was diagnosing problems. Because the user interface and the compiler were developed separately, it was always difficult to locate where a problem occurs. I needed to work together with the developer of the user interface in order to find out the location of the problematic code. One change in the compiler may affect some other unknown parts of the code. So special techniques were developed for diagnosing purposes.

10.1 Client Simulator

As described above, it is always difficult to find out whether a certain problem is in the compiler or in the user interface. Even worse, sometimes the progress of the user interface development is not the same as that of the compiler. For example, the compiler may have been updated to implement the functions to support hand-specified transformations, while this feature in the user interface was still under development. Instead of wasting time to wait until the user interface has been fully implemented, we developed simulators to simulate the user interface for debugging purposes.

The first simulator we developed just reads from a file. The compiler reads in commands from a file instead of the user interface. The communication is disabled in this mode and the commands in the file should also follow the protocol defined in Appendix B. When the compiler needs to send information to the user interface, the information is also written to a file. However, the communication between the

user interface and the compiler cannot be simulated by this simulator. So we soon moved to another real client simulator, which is also a separate program written in C. When starting the client simulator, it first establishes a communication socket with the compiler. Then the simulator reads commands from a file, sends the commands to the compiler, and reads information back from the compiler. This simulator simulates most of the features of the user interface that is needed to debug the compiler.

The simulator of the user interface is illustrated in Fig. 10.1

```
Establish a connection with the compiler;
Read in initial set from the compiler;
Write the initial set to an output file;
IF the compiler is accepting commands THEN
    WHILE not end of the input file
        Read in a command sequence from the input file;
        Write this command sequence to the compiler;
        Read information back from the compiler;
    END WHILE
END IF
Close the connection with the compiler.
```

Figure 10.1. Algorithm to Simulate the User Interface

10.2 Diagnosing Problems when Undoing Transformations

To diagnose problems when undoing transformations or changes, we defined the following commands:

write_command ::= WRITE_RTLS_TRANS filename

cmp_files_command ::= CMP_FILES_TRANS filename1 filename2

When testing a certain transformation, the typical sequence of commands is as below:

...
WRITE_RTLS_TRANS filename1
Transformation
Undo this transformation
WRITE_RTLS_TRANS filename2
CMP_FILES_TRANS filename1 filename2
...

A *write_command* is inserted before the transformation to write the program representation to a file. The transformation is performed, and then reversed. The reversed result is written to another file. Finally these two files are compared to see whether they are identical. If so, the reversing of this transformation is successful. Otherwise, the difference between these two files is inspected to debug the problem.

10.3 Check the Consistency of the Control Flow

Because of the numerous modifications in *vpo*, it is difficult to maintain the control flow consistency of the program representation. For example, it is very easy to delete a basic block without updating the predecessor list or successor list of the related basic block. The inconsistency of the control flow is difficult to find. So a *check_cf* function was developed to check the consistency of the control flow. Each time the compiler performs a change to the program representation, the compiler writer can insert a *check_cf* call to check the consistency of the control flow.

CHAPTER 11

FUTURE WORK

Vista is still undergoing development. However, the core features of the system have been implemented. This includes the ability to view the low-level program representation, interactively direct the order of the optimization phases, specify code-improving transformations by hand, undo previously applied transformations in the compiler, limit the scope in which an optimization phase will be applied, and support queries about loop information.

A user can currently request queries about loop information. We envision more queries that are useful, such as live variable information and symbolic expansion of expressions at given points in the control flow. Such requests will be useful when specifying transformations by hand or deciding which optimization phases to perform next. Besides these static queries, we also plan to allow a user to query for profile information, such as call edge counts, basic block counts, flow edge counts, and so on. This information will be helpful for a user to determine the scope of the program representation on which transformations should be applied. These queries will be sent to the compiler, which will obtain the desired information and send it back to the user interface to be displayed.

Currently, a user has to perform a compilation of a file in a single session. We plan to eventually allow a user to end a session at any time. The sequence of changes applied to the program representation will be saved in a configuration file. When the user wishes to resume a session later, the corresponding configuration file will be

read in. The past history of the transformations will still be valid as if it were never interrupted. This capability will give the user more flexibility when using *vista*.

We also plan to allow a user to achieve more abstract goals. For example, a user may wish a certain loop to be executed in a given amount of time, or to be compiled in no more than a specified number of bytes, or to consume a maximum amount of power. The compiler will try different combinations of optimizations in an attempt to find the combination that satisfies the user's goals.

CHAPTER 12

CONCLUSIONS

This thesis describes the modifications made to *vpo* to support a new interactive compilation paradigm to give the user the ability to finely control an optimization process. This modified *vpo* is used in a compilation framework called *vista*. Its features include the ability to graphically display the low-level representation of a program, allow a user to interactively direct the order and scope of the optimization phases, support the hand specification of code-improving transformations, undo previously applied transformations, and allow a user to query the state of the program representation. This system can be used by embedded systems developers to tune application code, by compiler writers to debug errors and/or prototype proposed code-improving transformations, and by instructors or educators to illustrate code-improving transformations to students learning compilation techniques.

APPENDIX A

PROTOCOLS OF MESSAGES FROM *VPO* TO THE USER INTERFACE

Below is the definition of the syntax of the messages sent from *vpo* to the user interface. The definition of many of the defined constants (words in capital letters) can be found in view.h.

$< compilation >$ ::= $< function > *$
$< end\_compilation\_msg >$

$< function >$ ::= $< begin\_func\_msg >$
$< initialset >$
$(< phase > | < valid\_response\_msg > | < loops\_msg >)*$
$< end\_func\_msg >$

$< initialset >$ ::= $< initbasicblock > *$
$< end\_init\_set\_msg >$

$< initbasicblock >$ ::= $< new\_bb\_msg >$
$[< new\_bbleft\_msg >]$
$[< new\_bbright\_msg >]$
$< rtl\_init\_msg > *$

$< phase >$ ::= $< begin\_phase\_msg >$
$< transformation > +$
$< end\_phase\_msg >$
$[< end\_seq\_msg >]$

$< transformation >$ ::= $< begin\_trans\_msg >$
$< change > +$
$< end\_trans\_msg >$

$< rtl\_init\_msg >$ ::= VET_RTL $< rtl\_id >< rtl\_type >< rtl\_val > [< assem\_val >]$

[VET_RTLLINKS < *rtl_link* > ∗0]
[VET_DEADS < *rtl_deads* >]
[VET_SIDE < *rtl_side* >]

< *valid_response_msg* >::= VALID | INVALID

< *loops_msg* > ::= BEGINLOOPS < *loop_msg* > + ENDLOOPS

< *loop_msg* > ::= LOOP < *nesting_level* >< *header_id* >
< *inner_id* > ∗
ENDLOOP

< *change* > ::= < *new_bb_msg* >
| < *update_bb_msg* >
| < *label_bb_msg* >
| < *del_bb_msg* >
| < *mod_bb_ptrs_msg* >
| < *insert_rtl_msg* >
| < *del_rtl_msg* >
| < *move_rtl_msg* >
| < *mod_rtl_msg* >
| < *mod_links_msg* >
| < *mod_side_msg* >
| < *mod_res_msg* >
| < *mod_deads_msg* >
| < *move_rtls_msg* >

< *end_init_set_msg* > ::= ENDINITSET

< *begin_phase_msg* > ::= BEGINPHASE < *phasenum* >

< *end_phase_msg* > ::= ENDPHASE

< *begin_trans_msg* > ::= BEGINTRANS

< *end_trans_msg* > ::= ENDTRANS

< *end_seq_msg* > ::= ENDSEQ

< *new_bb_msg* > ::= NEWBB < *blk_id* >< *label* >
//id for new block, label for new block

$< new\_bb\_left\_msg >$ ::= BBLEFT $< blk\_id >$

$< new\_bb\_right\_msg >$ ::= BBRIGHT $< blk\_id >$

$< update\_bb\_msg >$ ::= UPDATEBB
$< blk\_id >$\{VET_RTL $< rtl\_id >$\}*
ENDUPDATE
// id for block being update, ids of rtls in block

$< label\_bb\_msg >$ ::= LABELBB $< blk\_id >< label >$
// block id with label being updated, new label

$< del\_bb\_msg >$ ::= DELBB $< blk\_id >$
// block id of block being deleted

$< mod\_bb\_ptrs\_msg >$ ::= (MOD_DOWN | MOD_LEFT | MOD_RIGHT)
$< blk\_id >< blk\_id >$
// block id of block being updated, new value of ptr

$< insert\_rtl\_msg >$ ::= NEWRTL $< rtl\_id >< rtl\_type >< rtl\_val >< assem\_val >$
$< blk\_id >< rtl\_id >$
// id of new rtl, type of new rtl, val of new rtl,
// assembly of new rtl,
// block in which new rtl is inserted,
// old rtl before which new rtl precedes

$< del\_rtl\_msg >$ ::= DELRTL $< rtl\_id >< blk\_id >$
// rtl to be deleted, block containing deleted rtl

$< move\_rtl\_msg >$ ::= MOVERTL $< rtl\_id >< blk\_id >< rtl\_id >< blk\_id >$
// rtl to be moved, block where rtl was,
// rtl before moved rtl precedes,
// block where rtl is now

$< move\_rtls\_msg >$ ::= MOVERTL $< blk\_id >< blk\_id >$
// block where rtls to be moved currently reside
// block where rtls are to be appended at the end

$< mod\_rtl\_msg >$ ::= MOD_RTL $< rtl\_id >< rtl\_type >< rtl\_val >$
$< assem\_val >< blk\_id >$

// rtl to be modified, rtl type, new rtl val,
// assembly of new rtl,
// block containing rtl

< *mod_links_msg* > ::= MOD_RTLLINKS < *rtl_id* >< *rtl_type* >
< *rtl_id* > ∗0 < *blk_id* >
// rtl containing new links, rtl type,
// new links of rtl, block containing rtl

< *mod_side_msg* > ::= MOD_SIDE < *rtl_id* >< *rtl_type* >< *rtl_side* >< *blk_id* >
// rtl containing new reserve line, type of rtl,
// new side effect, block containing rtl

< *mod_res_msg* > ::= MOD_RES < *rtl_id* >< *rtl_type* >< *rtl_val* >< *blk_id* >
// rtl containing new reserve line, type of rtl,
// new reserve line, block containing rtl

< *mod_deads_msg* > ::= MOD_DEADS < *rtl_id* >< *rtl_type* >< *rtl_deads* >< *blk_id* >
// rtl containing modified deads, type of rtl,
// new deads for rtl, block containing rtl

//fields
< *rtl_id* > ::= INTEGER
< *rtl_type* > ::= INTEGER
< *rtl_val* > ::= CHARACTER_STRING
< *assem_val* > ::= CHARACTER_STRING
< *rtl_link* > ::= INTEGER
< *rtl_deads* > ::= CHARACTER_STRING
< *rtl_side* > ::= CHARACTER_STRING
< *funcname* > ::= CHARACTER_STRING
< *phasenum* > ::= INTEGER
< *header_id* > ::= < *blk_id* >
< *inner_id* > ::= < *blk_id* > | − < *header_id* >
< *blk_id* > ::= INTEGER
< *label* > ::= CHARACTER_STRING
< *nesting_level* > ::= INTEGER

APPENDIX B

PROTOCOLS FROM USER INTERFACE TO THE COMPILER

Below is the current protocol of messages that will be passed from the user interface to *vpo*. This protocol defines a sequence of commands that a user can select. *Vpo* will perform this sequence, send information back to the user interface, and wait for another sequence. No more sequences of commands will be issued from the user interface for the optimization of a function when a STOP_TRANS is encountered.

func_msgs	::=	requests STOP_TRANS
requests	::=	requests request | request
request	::=	sequence QUIT_TRANS
sequence	::=	hand_trans | loops_query
	|	undo_command | write_command | cmp_files_command
	|	normal_sequence
hand_trans	::=	HAND_TRANS < *change* > QUIT_TRANS
loops_query	::=	LOOPS_QUERY
normal_sequence	::=	numblocks ids commands
commands	::=	commands command | command
command	::=	branch_command | phase_command | other_command
	|	trans_command
branch_command	::=	(IF_TRUE_GOTO_TRANS | IF_FALSE_GOTO_TRANS) offset

phase_command ::= phase_oper

phase_oper ::= (DEAD_CODE_ELIM_TRANS
| FIX_CONTROL_FLOW_TRANS
| REG_ASSIGNMENT_TRANS
| DEAD_ASG_ELIM_TRANS
| COMMON_SUBEXPR_ELIM_TRANS
| FIX_ENTRY_EXIT_TRANS
| INST_SCHED_TRANS
| FILL_DELAY_SLOTS_TRANS
| MINIMIZE_LOOP_JUMPS_TRANS
| CODE_MOTION_TRANS
| LOOP_STRENGTH_REDUCT_TRANS
| RECURRENCE_ELIM_TRANS
| INDUCT_VAR_ELIM_TRANS
| REG_ALLOCATION_TRANS
| BRANCH_CHAINING_TRANS
| ELIM_EMPTY_BLOCKS_TRANS
| USELESS_JUMP_ELIM_TRANS
| REVERSE_BRANCHES_TRANS
| BLOCK_REORDERING_TRANS
| MERGE_BLOCKS_TRANS
| EVAL_ORDER_DETER_TRANS
| INST_SELECT_TRANS
| GLOBAL_INST_SELECT_TRANS
| STRENGTH_ REDUCT_TRAN
)

undo_command ::= UNDO_TRANS < *number* >

write_command ::= WRITE_RTLS_TRANS filename

cmp_files_command ::= CMP_FILES_TRANS filename filename

rtl ::= < *characterstring* >

filename ::= < *characterstring* >

offset ::= < *number* >

numblocks ::= < *number* >

ids ::= ids id | id

id ::= < *number* >

change ::= mod_rtl_msg
| del_rtl_msg
| insert_rtl_msg
| mod_assem_msg
| insert_assem_msg
| del_bb_msg
| insert_bb_msg
| END_TRANS
| BEGIN_TRANS

mod_rtl_msg ::= MOD_RTL rtl_id rtl_type rtl_val blk_id

del_rtl_msg ::= DELRTL rtl_id blk_id

insert_rtl_msg ::= NEWRTL rtl_id rtl_type rtl_val
blk_id rtl_id

mod_assem_msg ::= MOD_ASSEM rtl_id rtl_type assem_val
END_ASSEM blk_id

insert_assem_msg ::= NEW_ASSEM rtl_id rtl_type assem_val
END_ASSEM blk_id rtl_id

rtl_id ::= < *number* >

rtl_type ::= < *number* >

rtl_val ::= < *characterstring* >

assem_val ::= < *characterstring* >

blk_id ::= < *number* >

REFERENCES

[1] K. Andrews, R. Henry, and W. Yamamoto, "Design and implementation of the UW Illustrated Compiler," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 105–114 (June 1988).

[2] B. Appelbe, K. Smith, and C. McDowell, "Start/pat: a ParallelProgramming Toolkit," *IEEE Software*, (6), pp.29–40, (1988).

[3] M. E. Benitez, "Retargetable Register Allocation," *PhD Dissertation, University of Virginia*, (1994).

[4] 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*, Atlanta, GA, pp. 329–338 (June 1988).

[5] M. E. Benitez and J. W. Davidson, "The Advantages of Machine-Dependent Global Optimization," *Proceedings of the 1994 International Conference on Programming Languages and Architectures*, pp. 105–124 (1995).

[6] M. Boyd and D. Whalley, "Isolation and Analysis of Optimization Errors," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 26–35 (June 1993).

[7] M. Boyd and D. Whalley, "Graphical Visualization of Compiler Optimizations," *Journal of Programming Languages*, (3), pp. 69–94 (1995).

[8] J. Browne, K. Sridharan, J. Kiall, C. Denton, and W. Eventoff, "Parallel Structuring of RealTime Simulation Programs," *COMPCON Spring '90: ThirtyFifth IEEE Computer Society International Conference*, pp. 580–584 (1990).

[9] K. Cooper, P. Schielke, and D. Subramanian, "Optimizing for Reduced Code Space Using Genetic Algorithms," *Proceedings of the ACM SIGPLAN Workshop on Language, Compilers, and Tools for Embedded Systems*, pp. 1–9 (May 1999).

[10] Guido Costa Souza de Araujo, "Code Generation Algorithms for Digital Signal Processors," *PhD Dissertation, Princeton University, Princeton, NJ*, (June 1997).

[11] C. Dow, S. Chang, and M. Soffa, "A Visualization System for Parallelizing Programs," *Proceedings of Supercomputing*, pp. 194–203 (1992).

[12] Christine Eisenbeis and Sylvain Lelait, "LoRA: a Package for Loop Optimal Register Allocation," *3rd International Workshop on Code Generation for Embedded Processors, Witten, Germany*, (March 1998).

[13] Rainer Luepers, "Retargetable Code Generation for Digital Signal Processors," *Kluwer Academic Publishers, Boston*, (1997).

[14] Peter Marwedel and Gert Goossens, "Code Generation for Embedded Processors," *Kluver Academic Publishers, Boston*, (1995).

[15] W. G. Morris, "CCG: A Prototype Coagulating Code Generator," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 45–58 (June 1991).

[16] D. Polychronopoulos, M. Girkar, M. Haghighat, C. Lee, B. Leung, and D. Schouten, "Parafrase2: an Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors," *Proceedings of the 1989 International Conference on Parallel Processing*, pp. 39–48 (1989).

[17] Ashok Sudarsanam, "Code Optimization Libraries for Retargetable Compilation for Embedded Digital Signal Processors," *PhD Dissertation, Princeton University Princeton, NJ*, (November 1998).

[18] 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).

[19] R. van Engelen, D. Whalley, and Xin Yuan, "Automatic Validation of Code-Improving Transformations," *Proceedings of the ACM SIGPLAN Workshop on Language, Compilers, and Tools for Embeded Systems*, June 2000.

[20] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, "Compilers: Principles, Techniques, and Tools," *Addison-Wesley, U.S.A.*, 1986.

[21] Richard Stevens, "Unix Network Programming Volume 1, Networking APIs: Sockets and XTI," *Prentice Hall, U.S.A.*, 1998.

BIOGRAPHICAL SKETCH

Baosheng Cai

Baosheng Cai was born in Hebei, China in May, 1974. He received his B.E. in Automatic Control in 1997 from Tsinghua University. At Florida State University, he received his M.S. in Computer Science in 2001.